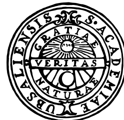
ACTA UNIVERSITATIS UPSALIENSIS
Uppsala Dissertations from the Faculty of Science and Technology
1214

Ahmed Rezine

Parameterized Systems: Generalizing and Simplifying Automatic Verification



UPPSALA
UNIVERSITET

Dissertation at Uppsala University to be publicly examined in MIC 2446, Lägerhyddsvägen 2, Tuesday, April 15, 2008 at 10:15 for the Degree of Doctor of Philosophy. The examination will be conducted in English

Abstract

Rezine, A. 2008. Parameterized Systems: Generalizing and Simplifying Automatic Verification. Acta Universitatis Upsaliensis. *Uppsala Dissertations from the Faculty of Science and Technology* 1214. 190 pp. Uppsala. ISBN 91-554-5436-4

In this thesis, we propose general and simple methods for automatic verification of parameterized systems. These are systems consisting of an arbitrary number of identical processes or components. The number of processes defines the size of the system. A parameterized system may be regarded as an infinite family of instances, namely one for each size. The aim is to perform a parameterized verification, i.e. to verify that behaviors produced by all instances, regardless of their size, comply with some safety or liveness property. In this work, we describe three approaches to parameterized verification.

First, we extend the Regular Model Checking framework to systems where components are organized in tree-like structures. For such systems, we propose a methodology for computing the set of reachable configurations (used to verify safety properties) and the transitive closure (used to verify liveness properties).

Next, we introduce a methodology allowing the verification of safety properties for a large class of parameterized systems. We focus on systems where components are organized in linear arrays and manipulate variables or arrays of variables ranging over bounded or numerical domains. We perform backwards analysis on a uniform over-approximation of the parameterized system at hand.

Finally, we suggest a new approach that enables us to reduce the verification of termination under weak fairness to reachability analysis for systems with simple commutativity properties. The idea is that reachability calculations (associated with safety) are usually less expensive than termination (associated with liveness). This last idea can also be used for other transition systems and not only those induced by parameterized systems.

Keywords: Parameterized systems, Automatic verification, Approximation, Regular model checking, Safety, Termination

Ahmed Rezine, Department of Electronic Publishing, Uppsala University, Lägerhyddsvägen 2, SE-752 37 Uppsala, Sweden

© Ahmed Rezine 2008

ISSN 1651-6214
ISBN 91-554-5436-4

urn:nbn:se:uu:diva-3344 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-3344)

à Salah ould-sidi rached, à Magali

Contents

1 Acknowledgments 15
2 Swedish Abstract 17
3 Arabic Abstract 21
4 Introduction 25
4.1 Background 25
4.1.1 Ever Running Programs 25
4.1.2 Testing versus Verification 26
4.1.3 Safety versus Liveness 28
4.1.4 Finite versus Infinite Model Checking 29
4.2 Parameterized Systems 31
4.2.1 Definition 31
4.2.2 Classes of Parameterized Systems 31
4.3 Related Work 33
4.3.1 Decidable Subclasses 34
4.3.2 Incomplete Subclasses 34
4.3.3 Fair Termination for Infinite-State Systems 35
4.4 Contributions 35
4.4.1 Methods 36
4.4.2 Implementations 37
4.5 Organization of the Thesis 38
4.6 Publications 38

Part I: Extending Regular Model Checking

5 Introduction 43
6 Regular Representation 45
6.1 Representing Linear Systems 46
6.1.1 Sets of Configurations and Automata 46
6.1.2 Transitions and Transducers 48
6.1.3 Verifying Linear Parameterized Systems 49
6.2 Representing Tree-Like Systems 52
6.2.1 Sets of Configurations and Tree Automata 54
6.2.2 Transitions and Tree Transducers 55
6.2.3 Examples of Tree-Like Systems 56
6.3 Conclusions 62
7 Transitive Closure of Tree Transducers 65
7.1 Computing the Transitive Closure 66

7.2 Soundness, Completeness, and Computability 68
7.2.1 Downward and Upward Simulation 68
7.2.2 Induced Equivalence Relation 71
7.2.3 Sufficient Conditions for Soundness and Completeness . . 72
7.2.4 Sufficient Condition for Computability 75
7.3 Good Equivalence Relation 75
7.4 Computing Reachable Configurations 79
7.5 Experimental Results 79
7.6 Conclusions 80
8 Conclusions 81

Part II: Monotonic Over-Approximation for Safety
9 Introduction 85
10 Reachability for Monotonic Systems 91
11 Systems with Boolean and Numerical Variables 95
11.1 Transition System 97
11.2 Ordering on Configurations 99
11.3 Approximation 101
11.4 Constraints, Operations and Termination 103
11.5 Additional Features 111
11.6 Conclusions 113
12 Extension to Components with Array Variables 115
12.1 Extended Transition System 117
12.2 Ordering on Configurations 118
12.3 Approximation 119
12.4 Constraints and Operations 120
12.5 Towards Distribution 124
12.6 Conclusions 126
13 Examples of Linear Parameterized Systems 127
13.1 Bounded Parameterized Mutex Algorithms 127
13.1.1 Burns Algorithm 128
13.1.2 Dijkstra Algorithm 129
13.1.3 Szymanski Algorithm 130
13.1.4 The Java Meta-Locking Algorithm 131
13.2 Bounded Parameterized Coherence Protocols 133
13.2.1 Illinois Coherence Protocol 134
13.2.2 DEC Firefly Coherence Protocol 135
13.2.3 Futurebus+ Coherence Protocol 136
13.2.4 German Coherence Protocol 139
13.3 Unbounded Parameterized Mutex Algorithms 142
13.3.1 Bakery with Races 142
13.3.2 Bakery with a Bug 143
13.3.3 Bakery with all Interleavings 144
13.4 Originally Distributed Mutex Algorithms 145

13.4.1 Distributed Mutex by Lamport 146
13.4.2 Distributed Mutex by Ricart-Agrawala 149
13.5 Experimental Results 151
14 Conclusions 153

Part III: Reducing Termination to Reachability

15 Fair Termination as Reachability 157
15.1 Fair Transition Systems 157
15.2 Intuition of the Main Proof Method 158
15.3 Main Proof Method 160
15.4 A Complementary Termination Method 162
15.5 Conclusions 164
16 Applications on Infinite Systems 165
16.1 Parameterized systems 165
16.2 Other Applications 167
16.2.1 Unbounded Variables: an Integer Program 167
16.2.2 Unbounded Channels: Alternating Bit Protocol 168
16.3 Conclusions 172
17 Conclusions 173

Part IV: Conclusions

Bibliography 177

List of Tables

7.1 Experiments with Trees 80

13.1 Burns Mutex Algorithm 128
13.2 Dijkstra Mutex Algorithm 129
13.3 Szymanski Mutex Algorithm. 131
13.4 Java Meta-Locking Algorithm. 132
13.5 Illinois Coherence Protocol 134
13.6 Dec Firefly Coherence Protocol 135
13.7 Futurebus Coherence Protocol 137
13.8 German Coherence Protocol 139
13.9 Bakery Mutex 143
13.10Bakery Mutex with a Bug 144
13.11Bakery Mutex with all Interleavings 145
13.12Lamport Distributed Mutex 148
13.13Ricart-Agrawala Distributed Mutex 149
13.14Experiments with Monotonic Abstraction 152

16.1 State Based Algorithm of Szymanski. 165

List of Figures

6.1 Instance of a simple linear parameterized system 46
6.2 Examples of word automata 47
6.3 Example of word transducers 48
6.4 Transitive closure 50
6.5 Transitive closure augmented 51
6.6 Closure is not always regular 52
6.7 Trees accepted by a tree automaton 53
6.8 Example of a tree configuration 54

10.1 Violation of mutual exclusion is regular 92

11.1 A transition rule in a linear parameterized system 95
11.2 Larger configurations can do more 100

13.1 Pseudo-code for Bakery algorithm 142

15.1 Left moving configurations 161
15.2 The Helpful set for an action 162

16.1 The Alternating Bit protocol 169
16.2 Calculation of $\Diamond F$ for the Alternating Bit protocol 171

1. Acknowledgments

It would be impossible to thank all the teachers, professors and friends who made this work possible.

I am undeniably in debt to my supervisor Parosh Abdulla. While your expertise and guidance introduced me to the realm of scientific research; your open door and heart flattened numerous difficulties during these last years.

I would like to thank my second supervisor Bengt Jonsson. Your scientific enthusiasm and availability are an example for me.

Working with Giorgio Delzanno on multiple occasions has been a great pleasure and an opportunity to learn and make possible important parts of this thesis. Thank you for your help, patience, and scientific generosity.

I would like to thank my dear friend and excellent colleague Noomene Ben Henda for the fruitful work and the numerous discussions in this mixture of English, Arabic, French and Swedish.

Working with Julien d'Orso, Mayanks Saksena, Axel Legay, Jonathan Cederberg and Frédéric Haziza has been an opportunity for practicing and enjoying scientific work. Thank you for sharing those long rewarding times. I want to express my gratitude to the members of the algorithmic program verification group Lisa Kaati, Johann Deneux, Pritha Mahata, Marcus Nilsson and Sven Sandberg. You made this PhD an incredibly enriching and pleasant experience.

Many thanks to Björn Victor, Therese Berg, Per Gustavsson, Tobias Lindhal, Magnus Ågren and all the other members of the "fika circle", for making of these sessions both a relaxing and a scientifically enriching experience. I would like to thank all those who helped with the Swedish abstract.

I am grateful to Ahmed Bouajjani and Peter Habermehl for receiving me in Paris during two months at the end of this thesis.

I can not forget to thank the employees of the department of computer science in Uppsala for their help in the flattening of the administrative issues, and for the enjoyable atmosphere of work.

Finally, I would like to express my deepest love and gratitude to my family and to my fiancee. Without you, nothing of this would have been possible. My parents Salah and Magali for your incomparable example of dedication and generosity; my siblings Rafik, Othmane and Lella Zhor for your constant support and care; and my fiancee Therese Egnor for your love, patience, and smile that lit the long deadline nights.

2. Sammanfattning på Svenska

Datorsystem har skapat möjligheter som för bara ett årtionde sedan ansågs vara ren science fiction. Denna utveckling verkar fortgå i oförminskad takt och påverkar numera nästan varje aspekt av våra liv, allt från bokningen av en tvättid till kontrollen av kylsystemet på kärnkraftverk.

Denna utveckling kommer dock med ett pris, nämligen ett ökat beroende av datorsystem rent generellt och specifikt av programmen som kontrollerar dem. Det är inte ovanligt att dessa program innehåller fel, så kallade "buggar". Konsekvenserna av dessa buggar kan vara allt från frusterande, som när din tvättid inte blir registrerad, till katastrofala, som när kylsystemet på ett kärnkraftverk stängs av.

Varje program finns till för att utföra en specifik uppgift. Huruvida ett program fungerar som förväntat eller inte kan beskrivas utifrån huruvida programmet besitter ett antal önskvärda egenskaper. Genom att säkerställa att de önskvärda egenskaperna möts är det möjligt att konstruera system utan "buggar".

Tillförlitlighet kontra liveness

De flesta egenskaper faller inom en utav följande två kategorier. Den första kategorin är egenskaper som säger att det aldrig uppstår fel, dvs egenskaper som säger att något dåligt aldrig kommer att hända. Dessa egenskaper kallas för *säkerhetsegenskaper*. Den andra kategorin av egenskaper beskriver funktioner där bra konfigurationer garanterat uppnås, dvs att någonting bra kommer slutligen att hända. Dessa egenskaper kallas för *liveness-egenskaper*.

Testning kontra verifiering

En stor del av mjukvaruindustrins arbete består i att felsöka programvara, det vill säga att påvisa funktioner som strider emot de önskvärda egenskaperna. Denna del av utvecklingen kan ibland vara mer än hälften av det totala arbetet, och är därmed en stor kostnad för industrin. Detta sker för att kunna försäkra sig om att programmen verkligen fungerar så som förväntat. De metoder som används främst inom industrin är simulering och testning. Fallstudier och tester skapas, ibland med hjälp av speciellt framställda verktyg och tekniker, för att kunna påvisa så många oönskade beteenden som möjligt. Varje sådant beteende som identifieras gör att felet (eller buggen), kan åtgärdas. Fallstudier och tester körs ett flertal gånger innan produkten till slut anses ha uppnått önskvärd kvalité. Arbetsmetoden är effektiv, speciellt under de första iterationerna för att identifiera flertalet felkällor, men simulering och testning är långt ifrån fullständiga som

metoder när det gäller att påvisa fel. De kan nämligen enbart användas för att påvisa existensen av ett fel, men aldrig bevisa att de inte finns. Den här situationen är oacceptabel om man ser till den betydelsefulla rollen vissa program spelar i samhället. I motsats till tillvägagångssättet med fallstudier och testning, som alltså är en inkomplett metod för att försäkra sig om ett programs felfrihet, gör användning av tekniken *formell verifiering* att man kan lämna garantier om felfrihet. Vid användande av denna metod är alltså de förväntade egenskaperna verifierade mot alla möjliga beteenden. Man utför verifieringen av ett specifikt programmet genom att använda sig av en matematisk modell av programmet. Modellen kan bestå av programmet själv eller en modell av detsamma där de irrelevanta detaljerna har utelämnats. Genom att manipulera den matematiska modellen av programmet kan man använda sig av formella verifieringstekniker redan under ett tidigt skede av utvecklingen av programmet. Att upptäcka och ta bort "buggar" i ett tidigt stadium kan medföra stora kostnadsmässiga besparingar i jämförelse med att felen upptäcks när produkten är redo för försäljning. Att använda sig av matematiska modeller underlättar verifieringen av program eftersom irrelevanta detaljer som i vissa fall skulle omöjliggöra verifieringen kan elimineras. Man skapar alltså en modell genom att abstrahera bort detaljer som inte är relevanta för själva verifieringen. Modellen analyseras för att verifiera att programmet uppfyller de krav som ställts på programmet. Analysen av modellen utförs med hjälp av *modellkontroll* (model checking).

Den stora utmaningen ligger i att automatiskt kunna verifiera att de krav man ställt på programmet uppfylls i modellens alla olika beteenden. Ett verktyg för modellkontroll använder sig av en modell och en egenskap och verktygets uppgift är att undersöka om egenskapen är uppfylld i modellen. Det här innebär att vissa egenskaper enbart kan kontrolleras i vissa typer av modeller. I bästa fall ger verktyget ett svar på några sekunder eller några minuter. I dessa fall, om svaret är positivt går det att vara säker att egenskapen är verifierad i alla situationer. I de fall svaret är negativt skapar verktyget ett motexempel som man kan använda sig av för att se var i programmet misstaget finns, vilket gör det lättare att åtgärda felet. Vanligtvis används modellkontroll för att verifiera system som har ett ändligt antal konfigurationer, men alla system är inte möjliga att modellera med ett ändligt antal konfigurationer.

Den här avhandlingen visar nya metoder för att verifiera den specifika familj av system som består av system med oändligt antal konfigurationer. Dessa system kallas för parametriserade system. Ett parametriserat system består av ett ändligt antal parallella processer. Vanligtvis antar man att processerna är finita tillståndsmaskiner med ett ändligt antal (finita) variabler Exempel på parametriserade system är: telekommunikationsprotokoll, protokoll för cachekoherens och sensorsystem. Systemet sägs uppfylla vissa tillförlitlighetsegenskaper eller liveness-egenskaper om systemets alla beteenden uppfyller egenskaperna. En av de stora utmaningarna ligger i att automatiskt verifiera att ett system är korrekt oberoende av hur många processer systemet består av. För

att kunna verifiera ett system som består av ett godtyckligt antal processer måste man verifiera ett oändligt antal system där var och ett av systemen består av ett unikt antal processer. Det går att bevisa att den här uppgiften är omöjlig att genomföra. Vårt arbete har resulterat i metoder för att verifiera tillförlitlighets- och liveness-egenskaper för parametriserade system. Vi har även implementerat våra metoder och tillämpat dem på några parametriserade system.

Arbetet har utförts i tre delar:

1. Metoder för att tillämpa reguljära modellkontroll-tekniker ("Regular Model Checking") på parametriserade system där processerna är organiserade i trädstrukturer och inte bara som linjära strukturer.
2. Metoder för skapa en abstrakt oändlig modell av ett parametriserat system. Detta för att kunna generera andra typer av oändliga system som går att verifiera. De här metoderna används för att verifiera tillförlitlighetsegenskaper.
3. Metoder för att verifiera livness-egenskaper med användning av nåbarhet ("reachability"). Dessa metoder kan även användas för att verifiera tillförlitlighetsegenskaper.

3.

اَلتَحقِيق الالي مِن سَلَامَةِ وَ حَيَوِيَةِ الأَنظِمَة المُوسِطَة[1]

يَقتَرِحُ هَذَا العَمَلُ طُرُقًا جَدِيدَةً لِلتَحَقُقِ آلِيَا مِن مُطَابَقَةِ أَصنَاف مِنَ الأَنظمَة المُوسِطَة لِمُواصَفَات تَضُمُ وَ تُحَدِّدُ سُلُوكَهَا. أَمكَنَتِ هَذِهِ الطرُقُ مِنَ التَحَقُقِ آلِيَا، لأَوَّلِ مَرَّة أَحيَانَا، مِنَ العَدِيدِ مِنَ الأَنظِمَة المُوسِطَة.

نَظَرَا لِدَرَجَةِ التَعَقِيدِ وَ التَفَاعُلِ الَّذي وَصَلَت إِلَيهِ الأَنظِمَةُ المَعلُومَاتِيَّة، أَصبَحَ مِنَ الضَّرُورِي التَثَبُت مِن مُطَابَقَتِهَا لِمُواصَفَات تَضمُنُ سَلَامَتِهَا، أَي أَنَّهُ لَا يُمكِنُ لِلبَرنَامِجِ الوُصُول إِلَى حَالَةِ سَيِّئَة، وَ حَيَوِيَتِهَا، أَي أَنَّ البَرنَامِج سَيُتِمُ مُهِمَتُهُ بِنَاءًا عَلَى أَنوَاعِ مِنَ الإِنصَاف. تَتَجَلَى هَذِهِ الضَرُورَة فِي حَالَةِ تَحَكُمِ الأَنظِمَةُ فِي أَجهِزَةُ تَكُونُ عَوَاقِبُ فَشَلِهَا عَن أَدَاءِ مُهِمَّتِهَا وَخِيمَةُ.

تُبنَى عَمَلِيَةُ التَثَبُتِ هَذِهِ عَادَةَ عَلَى الإِختِبَار أَو التَحقِيق. اَلإِختِبَارُ هُوَ الأَكثَرُ إِستِعمَالَا، وَ يَتَمَّثَلُ فِي الإِختِبَار اليَدَوِيَ أَو الآلِي لِلبَرَامِجِ فِي مُحَاوَلَة لِلتَثَبُتِ مِن أَنَّ سُلُوكَهَا مُطَابِقُ لِلمُواصفَات. مِنَ البَدِيهِي أَنَّ الإِختِبَار يَزِيدُ مِنَ الثِّقَة فِي صَلَاحِيَة البَرنَامِج، لَكِنَهُ لَا يُمكِنُ إِلَا أَن يَكُونَ مَحدُودَا بِاعتِبَارِ لَا نِهَائِيَة سُلُوكِ بَعض البَرَامِجِ. يَختَلِفُ التَحَقِيق عَن الإِختِبَار فِي أَنَّهُ يَرمِي إِلَى تَغطِيَة جَمِيعَ السُلُوكَاتِ المُمكِنَة. لِلوُصُول إِلَى هَذَا الهَدَف، تُستَعمَلُ خَوَارِزمِيَات مَبنِية عَلَى مَفَاهِيم رِيَاضِيَّة وَ مَنطِقِيَة.

[1] We use the ArabTEX package developped by Klaus Lagally.

مِنَ المُمكِن بَرهَنَة عَدَم إمكَانِيَة بِنَاء خَوَارِزمِيَات تُحَقِقُ جَمِيعَ الأَنظِمَة. لِذَلِكَ تُبنَى خَوَارِزمِيَات وَ كَسبِيَات تُمَكِنُ مِنَ التَحقُقِ مِن أَصنَافِ أَعّم كُل مَرَّة مِنَ الأَنظِمَة. مِنَ المَرغُوبِ الرَّفعُ مِن دَرَجَةِ آلِيَةِ هَاتِهِ الخَوَارِزمِيَات وَ الكَسّبِيَات، حَيثُ أَنَّ دَرَجَة الآلِية عَكسُ مُتَنَاسِبَة مَعَ عَدَدُ الأَخطَاء الإِنسَانِيَة المُرتَكَبَة عِندَ التَحقُقِ.

يَتَكَوَّنُ نَمُوذَجُ الأَنظِمَة المُوسِطَة الذِي نَتَبَنَاه مِن عَدَدُ مِنَ العَملِيَّاتِ المُتَنَافِسَة. هَذَا العَدَد مُوسِط أَي انَّهُ مُتَنَافِي لَكِن غَير مَحدَّد. تُغَيِّرُ كُل عَملِيَّة فِي النِظَام مِن حَالَتِهَا طِبقًا لِقَوَاعِد مُسَبَقَة. تَصِفُ كُلُ وَاحِدَة مِن هَاتِهِ القَوَاعِد الحَالَة الجَدِيدَة لِلعَملِيَّة بِنَاءًا عَلَى الحَالَة السَابِقَة وَ تَبَعًا لِنَوعِ القَاعِدَة، عَلَى حَالَة كُل أَو بَعض مِنَ العَملِيَّات الأُخرَى. يُمكِنُ إِضَافَة مُتَغَيِرَات مُشتَرَكَة، قَوَاعِد لِانشَاء وَ حَذف العَملِيَّات، قَوَاعِد بِالمَوَاعِد ... إِلخ.

يَتَمَثَلُ التَحَدِي فِي التَحقُق آلِيًا، وَ مِن غَير إِعتِبَار لِعَدَدِ العَملِيَّات، مِن مُطَابَقَةِ نِظَام مَعلُومَاتِي مُوسِط لِمُواصَفَات تَضُمُ وَ تُحَدِّدُ سَلاَمَتُهُ وَ حَيَوِيَتُهُ.

رزين أحمد
اوپسَالَا، اَلسُوَيد
١٤ جُويلِيَة ٢٠٠٧.

4. Introduction

Today you can download this thesis and read it on your cellular phone, while taking a high speed train. This would have been impossible without computer systems being involved in functionalities ranging from the files transmission, to the control of the dense railway traffic. Computer systems have created possibilities that would have seemed fantasy some decades ago. The trend appears unstoppable, and such systems are invading almost every aspect of our lives. This comes with a price; namely an ever increasing dependency on computer systems in general, and on the programs controlling them in particular.

4.1 Background

Programs controlling computer systems are rarely free of errors. Program errors manifest themselves in malfunctions with more or less severe consequences. A phone program that starts downloading but never finishes is an example of such a malfunction. A railway traffic control program that may allow on the same railway two trains heading towards each other is another one. While a user may get irritated and buy a new phone, disasters can occur because of a malfunctioning railway control program. It is therefore important, and sometimes vital, to ensure that programs do not exhibit such malfunctions and behave as expected. The expected behavior of each program can be formulated in terms of a number of desired properties. A desired property of the phone program is that it eventually finishes the downloads. A desired property for the railway control program is that it never allows the presence of two trains heading towards each other on the same railway. Excluding malfunctions of programs amounts to ensuring that programs respect such desired properties.

4.1.1 Ever Running Programs

Some programs are designed to perform calculations on inputs and to terminate after returning the result. The behaviors of such programs can be captured by relations between the input and the output. Programs like those controlling railway traffic or phone communication are not of that sort. Instead, such software is meant to continuously run while *reacting* to other programs or to its environment. These are reactive systems [MP95]. Without interaction, the phone is useless and the railway traffic dangerous. The behaviors of reactive programs can not be captured by some input/output relations. A more suitable

description uses structures that manipulate configurations and transitions of the program. Such structures are called *transition systems*.

Transition Systems

Later, we give a more formal definition of transition systems. Intuitively, a transition system involves three sets. The first one is the set of *configurations* of the system. For a program this would typically correspond to a snapshot of the values of all the manipulated variables at a particular time. The second set in a transition system consists of the set of *initial configurations*. These are configurations from which the system may start. For example values of the variables when the phone is turned on. The third and last set is the *transition relation*. This is the set of *transitions* between the configurations. Each transition is defined by the configuration before the transition and the configuration after the transition. A reactive program is meant to continuously run and to interact with its environment. In the transition system capturing its behaviors, this will correspond to sequences of configurations. The configurations will depict the values of the variables that are manipulated by the program or by the environment. These sequences, or *computations* of the transition system conform to the following two points; (i) the first configuration of the computation is an initial configuration; and (ii) two successive configurations belonging to the computation are related by a transition in the transition relation. Transition systems can be finite or infinite depending on the number of their configurations. Typically, even finite transition systems will exhibit infinite computations. These computations define the behaviors of the continuously running program.

4.1.2 Testing versus Verification

A major issue in software industry is to track down program malfunctions, i.e. to unveil program computations violating desired properties. A considerable amounts of effort and resources, sometimes comparable to those allocated to the projects themselves, are conceded in order to ensure that produced programs indeed behave as expected.

Testing and Simulation

The prevailing approaches in industry are the ones of simulation and testing. Simulation is usually carried out at an early stage of the software life cycle. In simulation, *models* (i.e. abstractions) of the different portions of the program being produced are run according to some scenarios. The testing approach is carried out on actual parts of the code. Testing comes therefore later in the process. In testing, the program (or portions of it) is run against test cases. Scenarios and test cases are conceived, sometimes with the help of specialized tools and techniques, in order to unveil the maximum number of malfunctioning behaviors. Once such a behavior is identified, the errors, i.e bugs, causing the malfunction are fixed. Scenarios and test cases are run for a number of iter-

ations until the product is judged to be of satisfying quality. These approaches are efficient, especially during the first iterations, to uncover many sources of malfunction. Simulation and testing are however inherently incomplete. They can only be used to show the presence of bugs, but never to show their absence. This situation is unacceptable considering the critical importance of some programs.

Formal Verification

Unlike the testing and the simulation approaches, this approach does not restrict the verification of a desired property to a subset of the possible behaviors. Instead, the considered property is verified against all possible behaviors. The verification is performed on a mathematical model of the targeted program. The model can be the program itself, or some representation where irrelevant details have been abstracted away. Similarly to the simulation approach, the manipulation of models allows the use of formal techniques at an early stage of the program conception. The detection and the removal of bugs at such stages is usually much cheaper than discovering the errors after deployment of the product. Models are also useful when the programs include irrelevant details that only make the verification heavier. For example, the actual content of the file to be downloaded for the phone program, or the number of seats per wagon for the railway control program are irrelevant details. The models are then obtained by abstracting away such details. The challenge is to verify that all behaviors of the model verify the desired property. There are roughly two families of formal verification techniques, namely deductive formal methods and model checking techniques.

Deductive Methods

Deductive, or logical inference, methods define the first family of formal verification techniques. In these techniques, the behaviors of the model are shown to verify the property by means of a mathematical proof. The proof is written by experts with the help of theorem provers. These [NPW02, CH88, HKPM04, ORS92] are programs that automate proving lemmas and theorems. Ultimately, the resulting proof is to be mechanically validated. This excludes shortcuts that may hide unexpected sources of errors. The expert describes the model and the properties to be proven.The complexity of the system usually means that the expert is required to supply intermediate lemmas. This dependency on the expertise of the user limits the use of deductive techniques to situations where the need for formal verification justifies the entailed resources and delays. Deductive methods are used, for instance, to infer the correctness of the floating point units in modern processors. The other formal verification approach is the one of model checking.

Model Checking

Model checking [CE82, QS82] techniques provide automatic tools for verification. Similarly to the deductive techniques, the user supplies both the model and the property to be verified. Unlike theorem provers, model checking tools automatically enumerate all possible behaviors of the model. The desired property to be checked is then verified against all the enumerated behaviors. Of course, this automation comes with a price. Namely that desired properties can be checked only on models for which such enumerations are possible. Typically, the model checking tool returns an answer in a matter of seconds or minutes. In such a case, the returned answer is positive if the desired property holds on all enumerated behaviors. A highly appreciated feature of model checking is that negative answers are usually accompanied with counter examples that help locating and fixing the malfunction. The work described in this thesis belongs to the family of model checking techniques.

4.1.3 Safety versus Liveness

The wide application range of computer systems suggests that expressing correctness is done in terms of heterogeneous properties. However, almost all properties fall in one of two categorizes. The first category describes computations where no bad configurations ever occur. An example of a bad configuration is one where the railway traffic program allows two trains heading towards each other. A property in this category is referred to as a *safety* property. Observe that such properties are violated if and only if the system encounters a bad configuration. The second category of properties describes computations where good configurations eventually occur. Configurations where the phone program has completed the downloading are good configurations. Observe that such properties are violated if and only if the system runs without ever encountering a good configuration. A property in this category is referred to as a *liveness* property.

Checking Safety

The techniques used for Model checking safety properties are different from those used for liveness properties. Consider the safety property stipulating that configurations with colliding trains should never be allowed by the railway traffic control program. Define the set of bad configurations to be the set of configurations where collisions are possible. The property is violated if and only if the transition system exhibits a computation where a bad configuration is encountered (i.e *reached*). In other words, the safety property holds if and only if no bad configuration is reachable. Obviously, one way to verify safety properties is to explore all reachable configurations and to verify that no bad configurations are encountered. This verification is always possible, at least from a theoretical point of view, for finite transition systems.

Checking Liveness

Another approach is generally required for Model checking liveness properties. Consider the property stating that the phone program eventually finishes downloading. On a transition system capturing the behavior of the phone program, the set of configurations where the phone finished the downloads corresponds to the set of good configurations. The considered liveness property amounts to checking that each of the computations of the transition system eventually encounters a good configuration. This does not correspond to the reachability of good configurations. Indeed, even if some good configurations are reachable, some computations may never encounter a good configuration. Such transition systems violate the considered liveness property. Instead, we need to verify the property for every computation. In other words, we need to verify the inevitability of good configurations. Therefore, liveness properties do not depend on the configurations seen apart, but rather on the possible sequences of configurations. For finite transition systems, computations correspond to a finite number of configurations, possibly followed by loops. The number of such loops is finite. Good configurations are not inevitable if one can reach a configuration from which starts a loop where no good configuration is encountered. Liveness is hence usually more complex than safety.

4.1.4 Finite versus Infinite Model Checking

Model checking is typically applied to programs whose behaviors are captured by finite transition systems. The model checking approach has showed great success for such programs. Model checking is being (more and more) used by industry as a complement to the traditional testing and simulation approaches. This success suggested extending model checking to infinite transition systems. In the following we introduce both finite and infinite model checking.

Finite Transition Systems

As mentioned earlier, it is always possible to check safety and liveness properties on programs modeled by a finite transition system. For these transition systems, model checking amounts to finding reachable configurations and loops. One difficult problem that arises here is the one of *state explosion*. State explosion refers to the situation where the number of configurations in the transition system is so large that the verification becomes intractable. For example, the number of visited configurations by a system of concurrent components can be polynomial in the number of configurations of each component, and exponential in the number of components. It was believed that state explosion will limit the application of model checking to some academic examples. Techniques like *symbolic methods*, *partial order* and *abstraction* were developed to tackle this issue.

- Symbolic methods manipulate representations of sets of configurations as opposed to explicit individual configurations. Special data structures are ex-

ploited for their efficiency when performing operations on such sets. *Binary Decision Diagrams* [Bry86, Bry92, McM92, BCM+92] are an example of data structures that have proven their efficiency.
- Partial order techniques [GP93, Pel94, Val89, Val92, GW93, God96] take advantage of the irrelevance, for the validity of the desired property, of the order in which certain transitions are taken. This is particularly useful in the case of programs composed of concurrent components. The idea is to restrict the number of computations to be explored to some representatives while still performing an exhaustive search.
- Abstraction techniques [CC77, GL93] map the transition system capturing the behavior of the program to an *abstract transition system*. The latter may, for instance, result from discarding the value of a variable, and replacing each test on its value by a non-deterministic choice. This corresponds to a loss of information that simplifies the enumeration while hopefully still allowing to prove the property. The abstraction has to be refined in case an error is only found in the abstract transition system.

As a result of these efforts, model checking finite programs with finite transition systems is now adopted as a complement for the simulation/testing paradigm. Variations of these techniques were adapted to the case of infinite transition systems.

Infinite Transition Systems

Yet, it is not feasible to faithfully capture the behavior of each program by a corresponding finite transition system. Many programs naturally induce transition systems with an arbitrarily large, sometimes infinite, number of configurations. There are several reasons why systems may have an infinite number of configurations: counters for the number of transmitted packages and number of concurrent components in a system, are examples of such reasons. The success of model checking for finite transition systems suggest extending automatic verification to the case of infinite transition systems. A naive enumeration will never terminate and is therefore out of question. Instead, many existing approaches adopt abstraction techniques or symbolic representations. Abstraction is used in order to work on abstract transition systems that are possible to analyze. For instance, it is sometimes possible to abstract infinite transition systems into finite ones [BLS01, BLS02, KP99]. The analysis is then performed using classical model checking techniques. The partitioning of the infinite transition system of timed automata [ACD90] into a finite number of clock regions can be seen as such an abstraction. Sets with infinite numbers of configurations can sometimes be represented and manipulated symbolically through finite representations. The use of *minimal elements* to represent their *upward closure* with respect to a quasi-order is a symbolic representation that has permitted the verification of several classes of infinite-state systems [AJ96b, ACJT96, FS98].

In this thesis, we focus on model checking the so called *parameterized systems*. These are concurrent systems where the unboundedness of the number of components (or processes) is the main source of infinity.

4.2 Parameterized Systems

A system may be *parameterized* in several ways. For instance, instead of proposing a different design for each number of bits in an adder, one could leave the width of the integers to be added as a parameter of the design. Parameterized systems are given together with a parameter which defines the way particular instances are built. In this thesis we work with systems composed of concurrent components. The parameter is the number of components.

4.2.1 Definition

A parameterized system consists of a number of concurrent components. The components (sometimes referred to as processes) are instantiations of the same *state-machine*. Usually, the state-machine is assumed to be finite and to manipulate a number of bounded variables. Examples of parameterized systems are common in telecommunication protocols, sensor systems, bus protocols, cache coherence protocols, etc.

A system resulting from the instantiation of a fixed number of components induces a transition system where configurations are given by the *state* of each component. Observe that we use the term *configuration* to denote a state of a parameterized system, in order to avoid confusion between states of a system, and the states of the state-machine defining the parameterized system. Initial configurations are configurations where each one of the components is at some initial state of the state-machine. A transition corresponds either to a component changing its state, or to several components simultaneously changing their respective states. We allow all possible interleavings, i.e. we assume *asynchronous* parameterized systems. In the case where the components are finite-state, the induced transition system is clearly finite for each fixed number of components. These transition systems are said to be correct with respect to some safety or liveness property if all possible computations satisfy the property. The challenge is to automatically verify correctness regardless of the number of components, i.e. regardless of the size of the system. This challenge amounts to verifying an infinite family of transition systems; namely one for each size.

4.2.2 Classes of Parameterized Systems

Several criteria may be used to differentiate classes of parameterized systems. We use two aspects to separate such classes, (i) the organization of the com-

ponents (*topology*), and (ii) the types of variables manipulated by the state-machine and the transitions associated with it. We introduce the classes we consider in this work, and give examples of protocols that can be modeled with them. This work focuses on the verification of parameterized systems where:

1. Processes are organized in linear arrays: in these systems, the components are identical except for their position in the array. Unstructured parameterized systems can be seen as a subclass where the position in the array is irrelevant. While performing a transition, a component may change state and read or write his set of variables (*local transitions*). A component may also read values of variables associated with all (or some) of the other components (*global transitions*). We tell apart three subclasses depending on the variables associated to each component:
   a. *Bounded variables*: also referred to as *finite state linear parameterized systems*; this class is the most studied in the literature. Each component owns a finite set of bounded variables, referred to as *local variables*. A component changes state via transitions where it can read and write its local variables, and may read the local variables of other components. These systems permit modeling several non-trivial cache coherence protocols and mutual exclusion algorithms. Among the mutual exclusion algorithms we cite the algorithms of Burns, Dijkstra [LPS93], Szymanski [Szy90, GZ98], and the Java meta-lock [RR04]. High level abstractions of cache coherence protocols can also be modeled by this class of parameterized systems. For instance, the Synapse, Berkley, Mesi, Moesi, Dec Firefly, Xerox P.D, Illinois, Futurebus+ [Han93] and the German [PRZ01] coherence protocols can be modeled here.
   b. *Numerical variables*: the local variables of each component may range over infinite domains. Observe that this class of parameterized systems strictly includes the previous class. It is for instance impossible for the previous class to faithfully model the Lamport bakery algorithm [Lam74] or the ticket algorithm [And99]. In the bakery algorithm, each component has an *identifier* and a *ticket number*. While the identifier can be seen as the position of the component in the array, the ticket number requires another numerical variable. Observe also that these systems have two sources of infinity, namely the number of states for each component (domain of the numerical variables) and the number of instances (size of the system).
   c. *Arrays of bounded and numerical variables*: a process can manipulate bounded and numerical variables, but also arrays of such variables. The arrays have the same size as the system (i.e. the number of processes in the system). This allows modeling complex systems. For instance, we can break down a transition involving reading the variables of all other processes into several steps. A process reads the variables of a single other process and marks it as read in its local array. Once all processes have been marked as read, the component proceeds. This allows more

interleaving in the model. We can also see the arrays as bounded channels between each pair of processes. This allows considering more general versions of the algorithms of Burns, Dijkstra, Szymanski or Bakery. Furthermore, other protocols were originally designed for a distributed setting. The distributed mutex of Lamport [Lam78], and the algorithm of Ricart and Agrawala [RA81] are such protocols.

We explain later how each of the above three classes of parameterized systems may be extended to allow broadcast and rendez-vous communication, creation and deletion of components, and global variables.

1. The processes are organized in *tree-structures*, and manipulate *bounded variables*: in these systems, a component manipulates variables ranging over finite domains, and is located at a node of a tree. A component has therefore a parent (if it is not placed at the root) and a fixed number of children (if it is not placed at a leaf). We consider in this work tree-structures consisting of nodes with a bounded number of children [CDG$^+$98, Tho90]. Protocols manipulating binary trees are examples of systems that are more naturally modeled here than with components organized in linear arrays. Examples of protocols that can be modeled by this formalism include the tree-arbiter protocol [ABH$^+$97], tree versions of the token-passing protocol and of the leader election protocol. Observe that this class includes the class of parameterized systems where the components are arranged in linear arrays and manipulate bounded variables. The relations before and after (or left and right) in the array are replaced by child and parent in the tree.

The challenge is to perform a so called *parameterized verification* on these classes using Model Checking techniques. In other words, we aim at extending Model Checking techniques in order to automatically verify the correctness of the protocols belonging to these classes regardless of the number of components in a particular instance.

4.3 Related Work

The verification of safety and liveness properties of parameterized systems is already undecidable [AK86] for finite state components. We are left with two possibilities [PZ03]; either to identify subclasses for which the verification becomes decidable, or to devise sound and necessarily incomplete methods in the hope of verifying the system at hand. In the following, we introduce the related work from three perspectives. First, we give examples of works dealing with subclasses for which the verification of safety, and sometimes liveness is decidable. Then we look at works proposing sound but incomplete verification procedures. Finally, we introduce works dealing with the verification of liveness (termination) properties for infinite-state systems in general, and parameterized systems in particular.

4.3.1 Decidable Subclasses

The focus here is on identifying subclasses of parameterized systems for which verification of safety or liveness properties is decidable. The components are finite state for each of the following cases.

German and Sistla [GS92] proposed decision procedures for the case of unstructured components. Components can change state either through a transition involving either one component, or two components synchronously changing their states, i.e rendez-vous. The decision procedure is polynomial for identical components; and exponential if a single control component is added to the system.

Emerson and Namjoshi studied in [EN95] the case of parameterized systems where components are organized in a ring and communicate by passing a unique token in a fixed direction. The authors also considered in [EN96] the case of identical components organized in arrays (possibly with a single control component) with synchronous transitions.

In [EFM99], the authors consider the class of broadcast protocols introduced in [EN98]. These are protocols where components communicate by rendez-vous, or broadcast, i.e a component sends a message to all other components. A broadcast action is always enabled, and the components that receive the message change state according to their current state. Although liveness is undecidable for these protocols, the approach of [ACJT96, ACJT00] is used to show the decidability of safety properties.

4.3.2 Incomplete Subclasses

There are no complete verification procedures for many classes of parameterized systems. Instead, sound and incomplete approaches are developed. These approaches come with different degrees of automation, and usually require user intervention; for instance in [SPBA00] the STeP prover is used to discharge some of the verification conditions needed in the verification proof.

One of the most used approaches consists in finding Network invariants [KM89, WL90, HLR92, LHR97]. The concurrent compositions of any number of components is proven to satisfy a given property in the following way. First a component is proven to satisfy a stronger property. Then, the composition of a component from the system with a component satisfying the stronger property is proven to still satisfy the latter. The stronger property is said to be a network invariant. Finding a network invariant is the most difficult step to automatize.

Abstraction is also used for parameterized verification. Some of the abstraction techniques aim at reducing the verification problem to the analysis of a finite state system. The automatic method of [PXZ02] based on counter abstraction for verifying liveness is an example of such techniques. Other approaches use predicate abstraction for parameterized verification; for instance [LB04] proposes heuristics to discover predicates. Different abstraction techniques can

be combined together or with other methods. For example, environment abstraction [CTV06] combines predicate abstraction with counter abstraction to deduce a finite state system, and [Del00a, Del00b] use real arithmetic calculations on the infinite model obtained by counter abstraction of the parameterized system.

Some approaches are tailored to the verification of particular classes of systems like [EK03b, Mai01, EK03a] for snoopy cache protocols. Other approaches like *regular model checking* [KMM$^+$01, ABJN99, JN00, PS00, AJNd03, AJMd02, BT02, DLS02] aim at performing a uniform and fully automatic verification. In regular model checking, configurations of bounded linear (tree) parameterized systems are represented by words (trees) over the set of component states. Sets of configurations by finite (tree) automata, and transitions by finite (tree) automata operating on pairs of states, i.e., *finite-state (tree) transducers*. The central point in regular model checking is to compute the transitive closure of finite-state transducers. To augment the chances of termination, this technique can be combined with abstraction [BHV04, BHRV06].

4.3.3 Fair Termination for Infinite-State Systems

The verification of liveness properties is in general harder than the verification of safety properties. Many classes of liveness properties, including *response*, *recurrence*, *guarantee* and *conditional guarantee* [MP90] can be reduced to termination. General liveness properties do not reduce to termination [MP90, Var91].

For infinite-state systems, the general approach for proving fair termination involves to find auxiliary assertions associated with well-founded ranking functions and helpful directions [MP84]. Automated construction of such ranking functions requires techniques adapted to specific data domains. Recently, significant progress has been achieved for programs that operate on numerical domains [BMS05a, BMS05b, BMS05c, CS01, CS02].

Finite-state abstractions has been difficult to apply when proving liveness properties, since abstractions may introduce spurious loops [PR05] that do not preserve liveness. Podelski and Rybalchenko therefore extended the framework of predicate abstraction to that of *transition predicate abstraction* [PR04, PPR05, CPR07, CGP$^+$07, PR07, CPR06]. This approach is still not applied to parameterized systems. Approaches tailored for parameterized verification of liveness include [BLS01, BLS02, PXZ02, FPPZ04, FPPZ06] and have different degrees of automation and applicability as introduced in 4.3.2.

4.4 Contributions

As introduced earlier, our goal is to perform automatic parameterized verification. The contribution of this work consists in proposing and implementing methods that did permit the verification of safety and liveness properties for challenging parameterized systems.

4.4.1 Methods

We mainly worked with three ideas: (i) Extending Regular Model Checking to parameterized systems with tree-like structures. A method is presented, and implemented, to build the transitive closure of tree transducers by means of an on-the-fly generated simulation relation, (ii) Verifying safety by uniformly over-approximating transition systems induced by parameterized systems into infinite approximated transition systems on which the framework of [ACJT96] is applied, and (iii) Reducing termination under weak fairness to a number of reachability calculations for systems with some simple commutativity properties. The idea is that reachability calculations (associated with safety) are usually less expensive than termination (associated with liveness).

Transitive Closure by Simulation for Tree Structures

Most of the regular model checking techniques target linear topologies when dealing with parameterized systems. We introduce a method to compute the transitive closure of relations accepted by regular tree transducers. This helps extending regular model checking to the tree case. Here, sets of configurations are accepted by tree automata and transitions are specified by tree transducers. The transitive closure of the transition relation makes it possible to verify both safety and liveness properties. We propose a semi-algorithm which starts from the tree transducer representing the transition relation. We derive a transducer, called the *history transducer* whose states are words of states of the original transducer. The history transducer characterizes the transitive closure of the transition relation, and presents in general an infinite set of states. We propose an on-the-fly approach to incrementally build a transducer that accepts the same tree language as the one accepted by the history transducer.

Safety via Abstraction into Infinite Monotonic Systems

In regular model checking, configurations of linear parameterized systems manipulating bounded variables are regarded as words over the finite set of component states. The subword relation, which is a *well quasi-ordering*[1] [Hig52], is a natural ordering on configurations of such systems. Monotonicity with respect to this ordering would have allowed the application of the framework in [ACJT96]. This is however not the case (recall the verification is undecid-

[1]well quasi-ordering (wqo) is a quasi-order $\preceq$ such that for each infinite sequence $a_0, a_1, \ldots$ there are $i < j$ with $a_i \preceq a_j$

able). The idea is to obtain monotonicity by over-approximation of the transition relation. The resulting approximate transition system is infinite and monotonic, and the analysis becomes decidable there. This approach was a success and has been extended to each one of the other mentioned linear classes. When extended to (processes manipulating) numerical variables, we use the formalism of *gap-order constraints* [Rev93] in order to constrain the variables. These are a logical formalism in which we can express simple relations on variables such as lower and upper bounds on the values of individual variables; and equality, and gaps (minimal differences) between values of pairs of variables. Although we loose termination (the new quasi-order is not a *wqo*), the obtained results were surprisingly positive. We were able to automatically verify systems with several sources of infinity, and arbitrary interleavings in the evaluation of global conditions (distribution).

Termination by Reachability Calculations

Many liveness properties may be reduced to termination in the same way safety properties are reduced to reachability. Termination properties state that a system is guaranteed to reach a given set of configurations. We propose methods to simplify the verification of termination (under weak-fairness) by reducing it to some reachability computations. These methods are valid for any (infinite) transition system. Simple commutativity requirements are used to reduce the verification to reachability calculations. We explore the applicability of the introduced methods on several systems including examples of finite state linear parameterized systems.

4.4.2 Implementations

We implemented prototypes to try the above methods. These implementations gave very encouraging results. We briefly report on them:

Safety by Abstraction

The over-approximation is uniformally and automatically applied to verify safety properties for each of the considered classes. The user enters the specification of the set of bad configurations together with the state machine defining the parameterized system. Sets of configurations are represented by constraints denoting upward closed sets with respect to the quasi-orderings defined for the class of parameterized systems at hand. Difference Bound Matrices were adopted as a data structure for manipulating numerical variables. This permitted automatic verification of safety properties for almost all of the linear examples mentioned in 4.2.2. A distributed version of the mutual exclusion algorithm by Szymanski was the only case that generated a *false positive*. We are not aware of any previous fully automatic verification for the protocols and algorithms of German cache coherence, Java Meta-Lock, Lamport Bakery, Lamport distributed mutex or Ricart and Agrawala distributed mutex.

Tree Simulation and Liveness Reduction

We experimented both the tree simulation and the liveness reduction within the framework of regular model checking [Nil02], and its extension by Julien D'Orso to the tree case. Binary Decision Diagrams are used in the tool in order to symbolically represent sets of states or transitions in the system. The tree-simulation implementation requires the user to specify an automaton characterizing the initial and the bad configurations, together with the transducer characterizing the transition relation. The prototype can compute the transitive closure or the set of reachable configurations. The result is used to check reachability of bad configurations. This allowed the fully automatic verification of the previously mentioned tree examples. The tool did not manage to compute the transitive closure of the tree-arbiter protocol. It is not even known whether this relation is regular. It was however possible to compute the set of reachable configurations for this protocol. The liveness reduction was experimented on parameterized systems. The necessary reachability calculations were carried out on the same tool [Nil02]. We took advantage of the acceleration techniques that were already implemented and showed starvation freedom for the mutual exclusion algorithm of Szymanski.

4.5 Organization of the Thesis

In Part I, we start by giving an intuition on how linear parameterized systems may be handled in the framework of regular model checking. We introduce the generalization for tree-structures, and describe examples of tree protocols. We explain our method to compute the transitive closure of a transition relation captured by a tree transducer, and report on the obtained results.

In Part II, we recall the framework of backwards reachability analysis for monotonic transition systems, and introduce the approach of monotonic approximation for safety verification. We explain the application of the approach in the case of linear parameterized systems manipulating both bounded and numerical variables. We continue by generalizing to the case of linear parameterized systems manipulating arrays of such variables. We conclude this Part by a presentation of several challenging examples we have verified.

In Part III, we focus on the reduction of termination verification to a set of reachability computations. We start by explaining two methods that apply, given simple requirements and the possibility to perform reachability analysis, to any (infinite) transition system. Then, we explain how to use the approach to verify termination on several examples.

In Part IV, we conclude the thesis.

4.6 Publications

This thesis is based on the following publications:

1. Parosh Aziz Abdulla, Noomene Ben Henda, Giorgio Delzanno, and Ahmed Rezine. Handling parameterized systems with non-atomic global conditions. In Francesco Logozzo, Doron Peled, and Lenore D. Zuck, editors, *VMCAI*, volume 4905 of *Lecture Notes in Computer Science*, pages 22–36. Springer, 2008.
2. Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine. Parameterized verification of infinite-state processes with global conditions. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *"Lecture Notes in Computer Science"*, pages 145–157. Springer, 2007.
3. Parosh Aziz Abdulla, Giorgio Delzanno, Noomene Ben Henda, and Ahmed Rezine. Regular Model Checking without transducers (on efficient verification of parameterized systems). In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *"Lecture Notes in Computer Science"*, pages 721–736. Springer, 2007.
4. Parosh Aziz Abdulla, Axel Legay, Julien d'Orso, and Ahmed Rezine. Tree regular model checking: a simulation-based approach. *J. Log. Algebr. Program.*, 69(1-2):93–121, 2006.
5. Parosh Aziz Abdulla, Bengt Jonsson, Ahmed Rezine, and Mayank Saksena. Proving liveness by backwards reachability. In Christel Baier and Holger Hermanns, editors, *CONCUR*, volume 4137 of *"Lecture Notes in Computer Science"*, pages 95–109. Springer, 2006.

Other publications

1. Parosh Aziz Abdulla, Ahmed Bouajjani, Jonathan Cederberg, Frédéric Haziza and Ahmed Rezine. Monotonic abstraction for programs with dynamic memory heaps. submitted.
2. Parosh Aziz Abdulla, Noomene Ben Henda, Frédéric Haziza and Ahmed Rezine. Parameterized tree systems. To appear in Proc. FORTE'08, International Conference on Formal Techniques for Networked and Distributed Systems, Tokyo, 2008.
3. Parosh Aziz Abdulla, Axel Legay, Julien d'Orso, and Ahmed Rezine. Simulation-based iteration of tree transducers. In Halbwachs and Zuck [HZ05], pages 30–44.
4. Parosh Abdulla, Julien d'Orso and Ahmed Rezine Quantitative analysis of infinite Markov Chains. 2nd Workshop on Quantitative Aspects of Programming Languages, Barcelona, 2004.

Part I:
Extending Regular Model Checking

5. Introduction

Regular model checking is the name of a family of techniques for analyzing infinite-state systems (like parameterized systems) in which configurations are represented by words, sets of configurations by finite automata, and transitions by finite automata operating on pairs of configurations, i.e. finite-state transducers. The central problem in regular model checking is to compute the transitive closure of a finite-state transducer. Such a representation allows to compute the set of reachable configurations of the system (which is useful to verify safety properties) and to detect loops between configurations (which is useful to verify liveness properties). However, computing the transitive closure is in general undecidable; consequently any method for solving the problem is necessarily incomplete. One of the goals of regular model checking is to provide semi-algorithms that terminate on many practical applications. Such semi-algorithms have already been successfully applied to parameterized systems with linear topologies, and to systems that operate on linear unbounded data structures such as queues, integers, reals, and hybrid systems [BJNT00, DLS02, BLW03, BHV04].

While using a finite-word representation is well-suited for systems with a linear topology, many interesting infinite-state systems fall outside of its scope. This is either because the behavior of the system cannot be captured by a regular relation [FP01], or because the topology of the system is not linear. A solution to the latter problem is to extend the applicability of the regular model checking approach beyond systems with linear topologies.

Outline:

This part contains two chapters. In the first chapter, we recall basic definitions for automata and transducers for both the linear and the tree cases. We use these definitions to describe linear and tree-like parameterized systems. In the second chapter, we provide an efficient and implementable semi-algorithm for computing the transitive closure of a tree transducer. This representation allows to compute the set of reachable configurations of the system and to detect loops between configurations.

6. Regular Representation

We introduce in this chapter representations of parameterized systems in the framework of regular model checking. Here, parameterized systems are of a linear or a tree-like topology. The components in both cases are assumed to be finite state. We start by the linear case, where automata and transducers are used to represent configurations and relations. This is generalized to the tree case were tree automata and tree transducers are used to represent configurations and relations. First, we define basic notions.

A *relation* on a set E is a set $\{(a_1,\ldots,a_n)|\ a_i \in E\}$ of tuples of elements in E. A relation $\mathcal{R}$ is said to be of arity n if all its tuples are of the same size n. Binary relations are relations of arity two. The *domain* of a binary relation $\mathcal{R}$ is the set $\{a|\ (a,b) \in \mathcal{R}\}$; its *image* is the set $\{b|\ (a,b) \in \mathcal{R}\}$. We write Id_E to mean the identity relation on a set E, i.e the binary relation $\{(a,a)|\ a \in E\}$.

Assume two relations on a set E, $\mathcal{R}_1$ with arity n_1 and $\mathcal{R}_2$ of arity n_2. If n_1 is larger than two, the projection with respect to the element i of $\mathcal{R}_1$ is the relation $\mathcal{R}_{|i}$ of arity $n-1$ and defined as $\{(e_1,\ldots,e_{i-1},e_{i+1},\ldots,e_n)|\ (e_1,\ldots,e_n) \in \mathcal{R}\}$. The projection of a binary relation is a relation of arity one, i.e. a set. The domain and the image of a binary relation $\mathcal{R}$ respectively equals $\mathcal{R}_{|2}$ and $\mathcal{R}_{|1}$. The cross product $\mathcal{R}_1 \times \mathcal{R}_2$ is the relation of arity n_1+n_2 defined by $\{(a_1,\ldots,a_n,b_1,\ldots,b_m)|\ (a_1,\ldots,a_n) \in \mathcal{R}_1,(b_1,\ldots,b_m) \in \mathcal{R}_2\}$. For instance, the cross product of two sets is a binary relation. The composition $\mathcal{R}_1 \circ \mathcal{R}_2$ of $\mathcal{R}_1$ and $\mathcal{R}_2$ is the smallest relation of arity n_1+n_2-2 containing all tuples $(a_1,\ldots a_{n_1-1},b_2,\ldots b_{n_2})$ such that there exists some element $c \in E$ satisfying $(a_1,\ldots a_{n_1-1},c) \in \mathcal{R}_1$ and $(c,b_2,\ldots b_{n_2}) \in \mathcal{R}_2$. Notice that the composition of two binary relations is also binary. The transitive closure of a binary relation $\mathcal{R}$ is the smallest relation $\mathcal{R}^+$ including $\mathcal{R}$ and satisfying $\mathcal{R}^+ \circ \mathcal{R} \subseteq \mathcal{R}^+$.

Remark 6.1. Observe that the composition of two binary relations can be expressed as the projection of an intersection of cross products. More concretely, and assuming binary relations $\mathcal{R}_1$ and $\mathcal{R}_2$ defined on E, we have $\mathcal{R}_1 \circ \mathcal{R}_2 = ((\mathcal{R}_1 \times E)\ \cap\ (E \times \mathcal{R}_2))_{|2}$.

We use these definitions to introduce how to handle linear and tree parameterized systems in the framework of regular model checking. First, we start by introducing how configurations and transitions of a linear parameterized system are represented, and how these representations are used to verify safety and liveness properties. Then we look at a generalization of these representations in order to handle tree-like parameterized systems. We use this generalization to describe a number of parameterized systems over tree-like structures.

6.1 Representing Linear Systems

We focus in this section on parameterized systems where components manipulate bounded variables (i.e., are finite state), and are arranged in linear arrays. Such systems are formally defined in chapter 11. For now we retain that a system belonging to this class is a pair of a finite set of *local states* and a finite set of *transition rules*. Intuitively, each component in the system can change state according to these transition rules. A transition rule is said to be *global* if it requires the component to check the state of some (or all) other components in the system, otherwise the transition rule is said to be *local*. In the following we use an example to explain how to handle such systems in the framework of *regular model checking*. More concretely, we explain how to represent and manipulate such infinite systems using automata and transducers.

Consider the instance (with four components) of a parameterized system depicted in figure 6.1. A component in this system can be in one of two states: *idle* or *use*. Intuitively, the components compete for a shared resource. At most one component may access the resource. Initially, all components are in state *idle*. In this system, each component can change state according to one of two transitions rules. A global transition rule, according to which a process may use the resource if all the processes to the left and all the processes to the right are in state *idle* (we write $\forall_{LR}(idle)$ to mean this condition), and a local transition rule where a component using the resource (i.e., in state *use*) may move back to state *idle*.

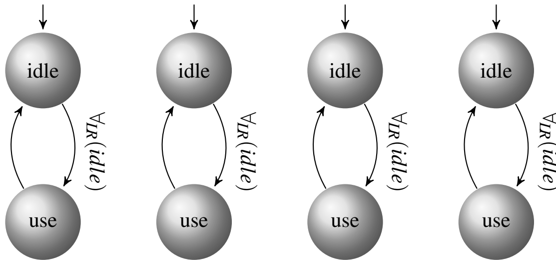


Figure 6.1: Instance of a parameterized system of size four. Each component has two possible states: *idle* and *use*. Initially, all components are in state idle. A component can move to state *use* only if all other components are in state *idle*. A component can always move back to state *idle*.

6.1.1 Sets of Configurations and Automata

Possible configurations of the system are sequences of length four of u (for *use*) and i (for *idle*). For instance, the sequence *iiii* corresponds to the initial configuration, while the sequence *iiui* corresponds to the configuration where all components, except the third one, are at their *idle* state. These sequences can be seen as words of length four over the alphabet $\{i,u\}$. It is then natural to

use *languages* (i.e., sets of words) as encoding of infinite sets of configurations. Since the parameterized system is an infinite family of instances of different sizes, the words in such languages are to be of arbitrary but finite lengths. In regular model checking, *regular languages* are utilized for this purpose. A regular language can be defined as the set of words accepted by a finite automaton [HU79]. More formally, define an *alphabet* to be a finite set Σ of symbols (for example the set $\{i,u\}$). Assume in the following an alphabet Σ.

Definition 6.1 (Finite automaton)**.** *An automaton over Σ is a tuple (Q,Σ,I,Δ,F) where:*
- *Q is a finite set of states,*
- *I is a subset of Q corresponding to the set of initial states.*
- *$\Delta \subseteq Q \times \Sigma \times Q$ is a finite transition relation, and*
- *$F \subseteq Q$ is the set of accepting states.*

The set of words accepted by an automaton A, and denoted by $\mathcal{L}(A)$, defines the language of the automaton (see figure 6.2). The concatenation $\mathcal{L}_1 \cdot \mathcal{L}_2$ of two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ is the set of all words xy where x is in $\mathcal{L}_1$ and y is in $\mathcal{L}_2$. Given a language $\mathcal{L}$, we write $\mathcal{L}^i$ to mean the set of words obtained by the concatenation of i words from $\mathcal{L}$. As a convention, we let $\mathcal{L}^0$ denote the language $\{\varepsilon\}$, where ε is the empty word. The *Kleene star* $\mathcal{L}^*$ of a language $\mathcal{L}$ is defined as the union $(\bigcup_{i\in\mathcal{N}} \mathcal{L}^i)$. For instance, the language Σ^* contains all possible words over Σ, plus the empty word ε. Figure 6.2 shows examples of word automata. Finite word automata are attractive for finitely representing possibly infinite sets of configurations. This representation allows the manipulation of such sets by taking advantage of useful automata properties. For instance, regular languages are known [HU79] to be closed under intersection, complementation and cross product. We can also use automata to represent relations on words.

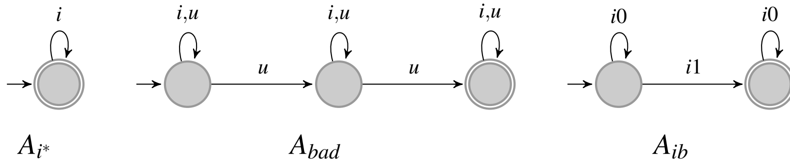


Figure 6.2: Examples of simple word automata: both automata A_{i^*} and A_{bad} are defined over the alphabet $\Sigma = \{i,u\}$ where i represents a process in state *idle* and u a process in state *use*. The accepting states are represented by double circles, while the initial states have an arrow without origin pointing to them. The automaton A_{i^*} accepts all sequences in $\mathcal{L}(A_{i^*}) = \{i\}^*$ and represents the initial configurations; the automaton A_{bad} represents configurations where at least two components are in state *use*. The automaton A_{ib} is defined on the alphabet $\{i0,i1\}$. The language of this automaton is $\mathcal{L}(A_{ib}) = \{i0\}^* \cdot \{i1\} \cdot \{i0\}^*$.

6.1.2 Transitions and Transducers

Configurations in a parameterized system change when a component changes its state by taking a transition. This defines a *relation* that relates two configurations if the second configuration is obtained from the first configuration by having a component taking a transition rule. We use the term *transition relation* to refer to the relation between configurations induced by a parameterized system. A relation that does not relate words of different lengths is said to be *length-preserving*. The relation on configurations associated with the transitions of a parameterized system is clearly length preserving if the number of processes does not change in a given instance of the system (no creation or deletion of processes). It is natural to use sequences of pairs of symbols to represent the state of each process before and after the transition. Such sequences can be accepted by automata over pairs of symbols, also known as *transducers* of arity two. Formally, a *transducer* of arity n over an alphabet Σ is a word automaton over the alphabet $\underbrace{\Sigma \times \ldots \times \Sigma}_{n}$.

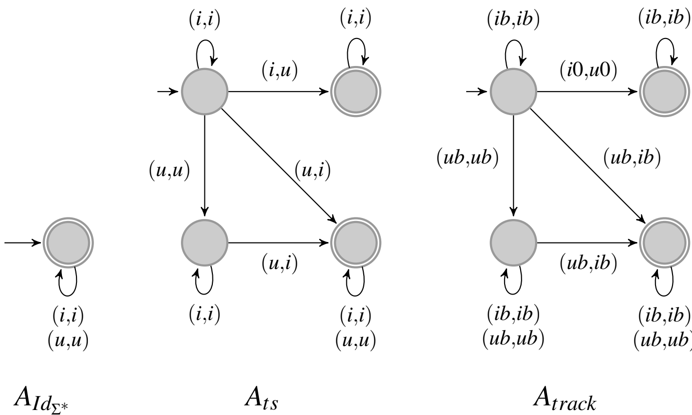


Figure 6.3: Example of word transducers of arity two. The transducer $A_{Id_{\Sigma^*}}$ represents the identity relation on all configurations captured by a finite sequence of elements of $\Sigma = \{i, u\}$; the transducer A_{ts} captures all possible relations between configurations as defined by the transitions of the parameterized system of figure 6.1. The transducer A_{track} is an example of augmenting A_{ts} to track some particular information. A_{track} is defined over $\Sigma = \{ib, ub\}$ with $b \in \{0, 1\}$. We write (xb, yb), for $x, y \in \{i, u\}$, to mean $(x0, y0), (x1, y1)$. Intuitively, a bit is associated to each component and allows enforcing a particular behavior. Here, the associated bit is copied at each transition of the system in figure 6.1. A_{track} enforces that components with the bit set to *true* are not allowed to move to state *use*.

The regular language accepted by a transducer A_D of arity n is a set of sequences of tuples of size n. Such a sequence defines a relation $rel(A_D)$ between n words over Σ. More concretely, consider a word $a_0 a_1 \cdots a_m$ accepted by the transducer A_D of arity n. This word is a sequence of

tuples with $a_i = \left(a_i^1, \ldots, a_i^n\right)$ where each a_i^j is a symbol in Σ. The regular language defines a relation in the sense that it can be seen as the set of tuples $\left((a_0^1 \cdots a_m^1), \ldots, (a_0^n \cdots a_m^n)\right)$. Such a relation is clearly both of arity n and length preserving. Transducers define *regular relations* in the same way automata define regular languages. Figure 6.3 shows examples of transducers defining regular relations.

Transducers are by definition automata. The union, intersection, complementation and Kleene star of regular relations are defined as the languages of the transducers resulting from the same operations on the corresponding transducers. Regular relations are therefore closed under these operations. It is easy to see that regular relations are closed under projection of one of the elements in the tuples forming the alphabet of their representing transducer. Considering remark 6.1, regular relations and transducers are closed under composition. This is important since it allows us to encode the relation resulting from a finite number of applications of the transition relation induced by a parameterized system. We introduce in the following how to use automata and transducers to verify safety and liveness properties for linear parameterized systems.

6.1.3 Verifying Linear Parameterized Systems

A central problem in *regular model checking* is to compute, for a transducer A_D of arity two, a transducer A_{D^+} that accepts the relation corresponding to the *transitive closure* of the relation induced by A_D. This allows two operations; namely to (i) deduce the set of configurations encountered (i.e., *reachable*) when starting from a set of initial configurations; and to (ii) find cycles between reachable configurations.

The set of reachable configurations is useful because it allows checking whether a particular set of configurations (for instance undesired or bad configurations) is encountered by the parameterized system (*safety* properties). For systems with a finite number of configurations (for example an instance of a length preserving finite-state parameterized system), infinite paths need to include at least a cycle. Recall that *liveness* properties can be stated as the reachability of a set of good configurations. Their negation (for infinite paths) amounts to the possibility of avoiding each of the good configurations in an infinite computation. Finding cycles between reachable configurations is useful because it permits checking the possibility for the parameterized system (with a length preserving transition relation) to take an infinite path without ever encountering a particular set of configurations. Observe that the absence of reachable cycles where no good configurations occur is sufficient to conclude that each infinite run of the parameterized system will encounter a good configuration.

We illustrate these ideas by considering the transducers $A_{(ts)^+}$ and $A_{(track)^+}$ of figures 6.4 and 6.5. The relations accepted by these transducers correspond

to the the transitive closures of the relations accepted by the transducers A_{ts} and A_{track} depicted in figure 6.3.

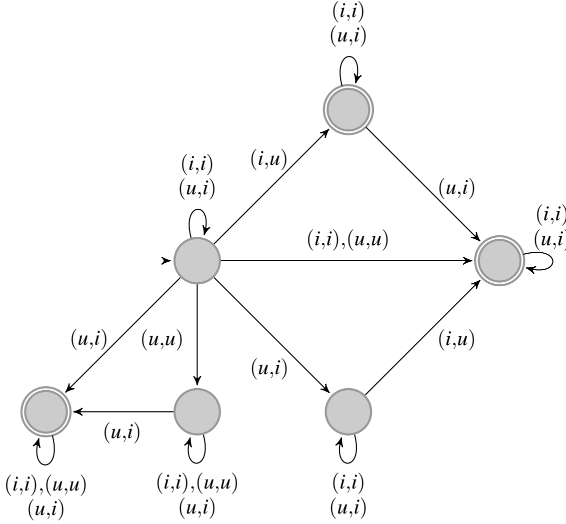


Figure 6.4: The transducer $A_{(ts)^+}$ represents the transitive closure of the relation represented by A_{ts} and depicted in figure 6.3.

The relation defined by the transducer $A_{(ts)^+}$ relates a sequence of two configurations iff the second configuration can be obtained from the first configuration by having a finite number of components of the parameterized system represented in figure 6.1. The set of configurations reachable from the configurations represented by the automaton A_{i^*} of figure 6.2 (corresponding to the initial configurations) are easily computed as $A_{reach} = (A_{i^*} \circ A_{(ts)^+})$. The resulting automaton A_{reach} accepts the language $\{i\}^* \cdot \{i,u\} \cdot i^*$ and represents the set of reachable configurations. Verifying the safety property of mutual exclusion (i.e. at most one process in state *use*) is a matter of checking the emptiness of the intersection of A_{reach} with the automaton A_{bad} depicted in figure 6.2 and representing the set of configurations violating mutual exclusion. In our case this intersection is empty, and the parameterized system is said to be safe.

The transducer $A_{(track)^+}$ is useful to verify whether each process in the parameterized system eventually moves to state *use*. We explain in the following how to perform this verification. The relation defined by the transducer $A_{(track)^+}$ corresponds to the transitive closure of the relation defined by the transducer $A_{(track)}$. Recall (figure 6.3) that we are tracking some extra information in $A_{(track)}$ compared to $A_{(ts)}$; (i) each process is augmented with a bit that is copied at each transition of the parameterized system (whose instance with four components is) represented in figure 6.1; and (ii) components with a bit set to *true* are not allowed to move to state *use*. As a result, an infinite path where a component never moves to state *use* exists iff there is a cycle starting from a reachable configuration where a component has its bit set to *true*. Verifying

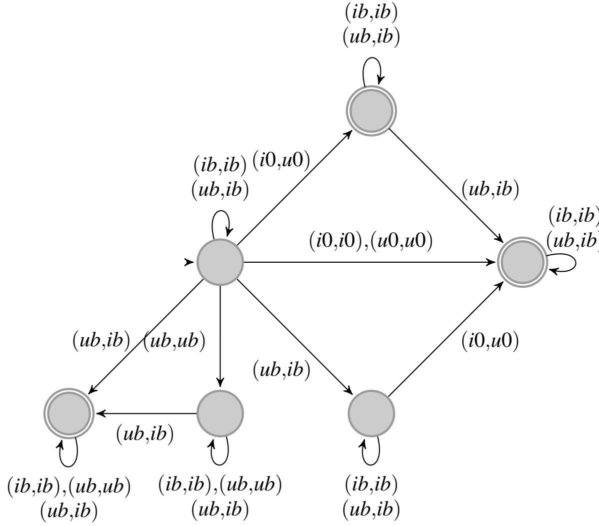


Figure 6.5: The transducer $A_{(track)^+}$ represents the transitive closure of the relation represented by A_{track} depicted in figure 6.3. In a similar manner to the alphabet of A_{track}, we write (xb, yb) for $x, y \in \{i, u\}$ to mean $(x0, y0), (x1, y1)$.

the liveness property of individual starvation freedom (i.e., each component eventually moves to state *use*) is a matter of two operations; (i) computing the set of reachable configurations $A_{rb} = (A_{ib} \circ A_{(track)^+})$ where A_{ib} is depicted in figure 6.2; and (ii) checking the emptiness of the intersection of $A_{(track)^+}$ with the identity relation on reachable configurations A_{rb}. In our case this intersection is not empty. This was predictable, since a process can repeatedly move to state *use* blocking another process each time the latter tries to move from *idle* to *use*. One can introduce some fairness assumptions to tackle this problem. The uniform framework of [AJN$^+$04] allows applying regular model checking under such assumptions.

In regular model checking safety and liveness properties for linear parameterized systems are verified by computing the transitive closure of the transition relation (or some augmented version of it) of the parameterized system at hand. An important effort was invested into finding methods to compute these closures [ABJN99, JN00, PS00]. The example 6.1 illustrates that this computation is not always possible.

Example 6.1. Consider the binary relation $rel(A_{(i \cdot u) \leftrightarrows (u \cdot i)})$ represented by the transducer of figure 6.6. If the transitive closure of $rel(A_{(i \cdot u) \leftrightarrows (u \cdot i)})$ was regular, there would be an automaton $A_{((i \cdot u) \leftrightarrows (u \cdot i))^+}$ such that $rel(A_{((i \cdot u) \leftrightarrows (u \cdot i))^+}) = rel(A_{(i \cdot u) \leftrightarrows (u \cdot i)})^+$. Consider now the automata $A_{(i \cdot u)^*}$ and $A_{i^* \cdot u^+}$ from figure 6.6. The set $\mathcal{L}$ defined as the language of the automaton $\left(A_{(i \cdot u)^*} \circ A_{((i \cdot u) \leftrightarrows (u \cdot i))^+}\right) \cap A_{i^* \cdot u^+}$ is regular by closure of regular languages under composition, projection intersection, and cross product. But $\mathcal{L}$ is exactly $\{b^n a^n \,|\, n \in \mathcal{N}^+\}$. A simple

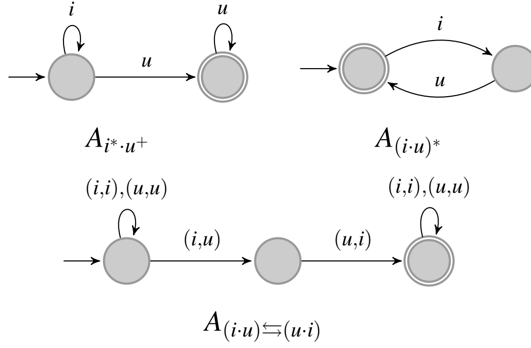


Figure 6.6: Two automata and a transducer of arity two over the alphabet $\{i,u\}$. The automaton $A_{(i^*\cdot u^+)}$ represents configurations where at least one process is in state *use* and no process in state *use* is to the left of a process in state *idle* $\mathcal{L}\left(A_{(i^*\cdot u^+)}\right) = \{i\}^* \cdot \{u\} \cdot \{u\}^*$; the automaton $A_{(i\cdot u)^*}$ represents configurations where a process in state *use* always follows a process in state *idle*, and where there are no consecutive processes in state *use* $\mathcal{L}\left(A_{(i\cdot u)^*}\right) = \{(i \cdot u)^n |\, n \in \mathbb{N}\}$; The transducer $A_{(i\cdot u)\leftrightarrows(u\cdot i)}$ relates a sequence of two configurations if the second configuration can be obtained from the first configuration by performing the two following operations, (i) find in the first configuration a sequence *iu*; (ii) replace the sequence by *ui* and copy the rest of the symbols.

application of the pumping lemma for regular languages [HU79] proves this language not regular.

The fact that transitive closure of a (relation represented by a) transducer can not be computed in general is not strange since the verification problem is undecidable in general. This sets a limitation to the approach of regular model checking. One possibility at this point is to restrict the classes of systems to be verified.

Another approach is to be aware of the limit above, and to devise methods that permit concluding the verification on particular examples. In the following chapter, we extend regular model checking in order to verify parameterized systems where the processes are organized in tree-like structures instead of linear arrays. It is more natural to describe some systems using such tree-structures.

6.2 Representing Tree-Like Systems

In this section we explain how to extend the regular representations defined in the previous section to the case of parameterized systems operating on tree structures. Similarly to linear parameterized systems with finite components, the parameterized systems we consider are composed of components with a finite number of states. Figure 6.7 describes three configurations of a particular instance of a tree parameterized system. There are five components in this

instance. In parameterized systems over tree-like structures, trees play the role played by words in the case of linear systems.

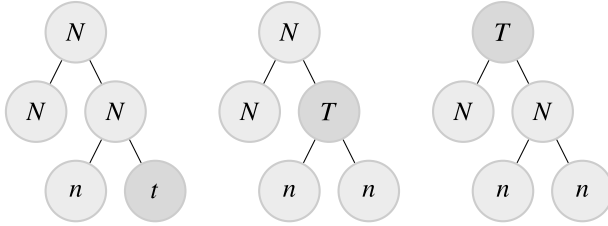


Figure 6.7: Configurations of an instance of a parameterized system over tree-like structures. The components are placed in the nodes of the tree. Components with labels t or T have a token. Symbols n or N are used to label components that do not have a token.

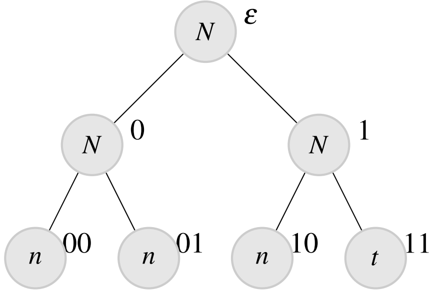
We need to define ranked alphabets in order to formally describe trees. A *ranked alphabet* is a pair (Σ, ρ), where Σ is a finite set of symbols and ρ a mapping from Σ to the set of natural numbers $\mathcal{N}$. For a symbol $f \in \Sigma$, $\rho(f)$ is the *arity* of f. We use Σ_p to denote the set of symbols in Σ with arity p. Intuitively, each node in a tree is labeled with a symbol in Σ with the same arity as the out-degree (i.e. number of successors) of the node. We sometimes abuse notation and use Σ to denote the ranked alphabet (Σ, ρ). The nodes in a tree are represented by words over $\mathcal{N}$. More precisely, the empty word ε represents the root of the tree, while a node $b_1 b_2 ... b_k$ is a child of the node $b_1 b_2 ... b_{k-1}$. Each node is also labeled by a symbol from Σ.

Definition 6.2 (trees)**.** *A tree T over (Σ, ρ) is a pair (S, λ) where:*

- *S, called the tree structure, is a finite set of sequences over $\mathcal{N}$ (i.e, a finite subset of $\mathcal{N}^*$). Each sequence n in S is called a node of T. If S contains a node $n = b_1 b_2 ... b_k$, then S will also contain the node $n' = b_1 b_2 ... b_{k-1}$, and the nodes $n_r = b_1 b_2 ... b_{k-1} r$, for $r : 0 \leq r < b_k$. We say that n' is the parent of n, and that n is a child of n'. A leaf of T is a node n which does not have any child, i.e., there is no $b \in \mathcal{N}$ with $nb \in S$.*
- *λ is a mapping from S to Σ. The number of children of n is equal to $\rho(\lambda(n))$. Observe that if n is a leaf then $\lambda(n) \in \Sigma_0$.*

Configurations of tree-like parameterized systems are naturally described as trees labeled by the states of the components. Figure 6.8 depicts a configuration of an instance of a parameterized system with seven components placed in a binary tree.

We use $T(\Sigma)$ to denote the set of all trees over Σ. Trees represent configurations of parameterized systems over tree like structures in the same way words represent configurations of linear parameterized systems. Sets of configurations are represented by tree automata.



The tree to the left is a tuple (S,λ) over (Σ,ρ) where:

- $\Sigma_0 = \{n,t\}$, $\Sigma_2 = \{N,T\}$;
- $S = \{\varepsilon,0,1,00,01,10,11\}$;
- $\lambda(00) = \lambda(01) = \lambda(10) = n$;
- $\lambda(11) = t$;
- $\lambda(\varepsilon) = \lambda(0) = \lambda(1) = N$.

Figure 6.8: A tree with the component placed in the left most leaf is the only process holding the token. All other components are *labeled* by n or N to reflect that they do not have the token.

6.2.1 Sets of Configurations and Tree Automata

In a manner that is similar to the way word languages are used to represent possibly infinite sets of words, *Tree languages* [Tho90, CDG+99] are used to represent possibly infinite sets of trees. Tree automata are used to accept and manipulate these languages. There exists various kinds of tree automata. We use here bottom-up tree automata. In the sequel, we will omit the term bottom-up.

Definition 6.3 (tree automata)**.** *A tree automaton A over a ranked alphabet Σ is a tuple (Q,F,Δ), where Q is a set of states, $F \subseteq Q$ is a set of final states, and Δ is the transition relation. Δ is given as a set of rules each of the form:*

$$(q_1,\ldots,q_p) \xrightarrow{f} q$$

where $f \in \Sigma_p$ and $q_1,\ldots,q_p,q \in Q$.

Unless stated otherwise, Q and Δ are assumed to be finite. Intuitively, the automaton A takes a tree $T \in T(\Sigma)$ as input. It proceeds from the leaves to the root, annotating states to the nodes of T. A transition rule of the form $(q_1,\ldots,q_p) \xrightarrow{f} q$ tells us that if the children of a node n are already annotated from left to right with $q_1,\ldots,q_p$ respectively, and if $\lambda(n) = f$, then the node n can be annotated by q. As a special case, a transition rule of the form $\xrightarrow{f} q$ implies that a leaf labeled with $f \in \Sigma_0$ can be annotated by q. The tree is accepted by the automaton if its root is labeled by a final state.

More formally, the set of trees accepted by an automaton A (denoted by $\mathcal{L}(A)$) are characterized in terms of *runs*. A *run* r of $A = (Q,\Sigma,I,\Delta,F)$ on a tree $T = (S,\lambda) \in T(\Sigma)$ is a mapping from S to Q such that for each node $n \in T$ with children $n_1,\ldots,n_k$ we have $\left((r(n_1),\ldots,r(n_k)) \xrightarrow{\lambda(n)} r(n)\right) \in \Delta$. For a state q, we let $T \overset{r}{\Longrightarrow}_A q$ denote that r is a run of A on T such that $r(\varepsilon) = q$. We use $T \Longrightarrow_A q$ to denote that $T \overset{r}{\Longrightarrow}_A q$ for some r. For a set $S \subseteq Q$ of

states, we let $T \stackrel{r}{\Longrightarrow}_A S$, resp. $T \Longrightarrow_A S$, denote that $T \stackrel{r}{\Longrightarrow}_A q$, resp. $T \Longrightarrow_A q$, for some $q \in S$. We say that A *accepts* T if $T \Longrightarrow_A F$. We define $\mathcal{L}(A) = \{T \mid T \text{ is accepted by } A\}$. A tree language $\mathcal{L}$ is said to be *regular* if there is a tree automaton A such that $\mathcal{L} = \mathcal{L}(A)$.

Example 6.2. The following tree automaton A accepts the tree of Figure 6.8. Let $\Sigma = \{n,t,N,T\}$, with $\rho(n) = \rho(t) = 0$ and $\rho(N) = \rho(T) = 2$. Define A to be the tuple (Q,F,Δ) where $Q = \{q_0,q_1\}$ and $F = \{q_1\}$. The transition rules in Δ are as follows:

$$\stackrel{n}{\longrightarrow} q_0 \qquad \stackrel{t}{\longrightarrow} q_1 \qquad (q_0,q_0) \stackrel{T}{\longrightarrow} q_1$$

$$(q_0,q_0) \stackrel{N}{\longrightarrow} q_0 \qquad (q_0,q_1) \stackrel{N}{\longrightarrow} q_1 \qquad (q_1,q_0) \stackrel{N}{\longrightarrow} q_1$$

The automaton A accepts all trees over Σ containing exactly one occurrence of t or T. The trees of figure 6.7 are accepted by this automaton.

Finite tree automata can be used as a symbolic representation of possibly infinite tree languages. The choice of bottom-up automata ensures [Tho90, CDG+99] closedness under operations like intersection, union, minimization, determinization, inclusion test and complementation. We can also use tree automata to represent relations on trees. For this we use tree transducers.

6.2.2 Transitions and Tree Transducers

In this section we describe *tree relations* and use special tree automata known as *tree transducers* to define *regular tree relations*. Similarly to the linear case, this allows the representation of transitions of a parameterized system.

Given a ranked alphabet (Σ,ρ) and a positive natural m, we let $\Sigma^\bullet(m)$ be the ranked alphabet $(\Sigma^\bullet,\rho^\bullet)$, where $\Sigma^\bullet = \{(f_1,\ldots,f_m) \in \Sigma^m | \rho(f_1) = \cdots = \rho(f_m)\}$ and $\rho^\bullet((f_1,\ldots,f_m)) = \rho(f_1)$. In other words, the alphabet $\Sigma^\bullet(m)$ contains the m-tuples, where all the elements in the same tuple have equal arities. Furthermore, the arity of a tuple in $\Sigma^\bullet(m)$ is equal to the arity of any of its elements. For trees $T_1 = (S_1,\lambda_1)$ and $T_2 = (S_2,\lambda_2)$, we say that T_1 and T_2 are *structurally equivalent* (denoted $T_1 \cong T_2$) if $S_1 = S_2$. Consider structurally equivalent trees $T_1,\ldots,T_m$ over an alphabet Σ, where $T_i = (S,\lambda_i)$ for $i : 1 \leq i \leq m$. We let $T_1 \times \cdots \times T_m$ be the tree $T = (S,\lambda)$ over $\Sigma^\bullet(m)$ such that $\lambda(n) = (\lambda_1(n),\ldots,\lambda_m(n))$ for each $n \in S$. An m-ary *tree relation* on the alphabet Σ is a set of tuples of the form $(T_1,\ldots,T_m)$, where $T_1,\ldots,T_m \in T(\Sigma)$ and $T_1 \cong \cdots \cong T_m$. A tree language $\mathcal{L}$ over $\Sigma^\bullet(m)$ characterizes an m-ary tree relation $[\mathcal{L}]$ on Σ as follows: $(T_1,\ldots,T_m) \in [\mathcal{L}]$ iff $T_1 \times \cdots \times T_m \in \mathcal{L}$.

Tree automata can also be used to characterize tree relations. A *tree transducer* is a tree automaton A over $\Sigma^\bullet(m)$. A tree transducer characterizes an m-ary tree relation on Σ, namely the relation $[\mathcal{L}(A)]$. A tree relation is said to be *regular* if it is equal to $[\mathcal{L}(A)]$ for some tree transducer A. In such a case, this relation is denoted by $rel(A)$. Like regular word relations, regular tree languages are closed under cross product, projection, and [CDG+99] all boolean operations. As a consequence, regularity of tree relations is preserved by the

operators for union, intersection, complementation and composition. Regular relations are not closed under transitive closure because that would imply the closure of regular word languages under transitive closure.

Remark 6.2. There are other definitions of tree transducers. The one we adopt here is a restricted version of the definition considered in [BT02] in the sense that we only consider transducers that do not modify the structure of the tree. In [BT02], such transducers are called relabeling transducers.

We give in the following examples of parameterized systems that can be modeled by tree automata and tree transducers.

6.2.3 Examples of Tree-Like Systems

We describe, using tree automata and tree transducers, encoding of parameterized systems on tree structures.

Simple Token Protocol

The processes in this parameterized system are connected in a binary tree-like fashion. Each process may or may not possess a token. In this system, a token is passed from a leaf to the root. Figure 6.7 depicts three consecutive configurations of an instance with five components.

To represent the parameterized system, we use a tree transducer over an alphabet consisting of $t,n \in \Sigma_0$ representing processes at the leaves, and $N,T \in \Sigma_2$ representing processes at the inner nodes of the tree. Processes labeled by $\{n,N\}$ are those which do not have a token, while those labeled by $\{t,T\}$ are those which do have a token.

The set of initial configurations can be encoded by a tree automaton which recognizes all possible configurations in which one leaf has the token, i.e. $A_i = (Q_i, F_i, \delta_i)$, where $\Sigma = \{n,t,N,T\}$, $Q_i = \{q_{i_0}, q_{i_1}\}$, $F_i = \{q_{i_1}\}$, and δ_i is defined by the following set of rules:

$$
\begin{array}{ccc}
\xrightarrow{n} q_{i_0} & \xrightarrow{t} q_{i_1} & \\
(q_{i_0}, q_{i_0}) \xrightarrow{N} q_{i_0} & (q_{i_0}, q_{i_1}) \xrightarrow{N} q_{i_1} & (q_{i_1}, q_{i_0}) \xrightarrow{N} q_{i_1}
\end{array}
$$

The transducer D that models the behavior of the protocol is given by :

$$
\begin{array}{ccc}
 & \xrightarrow{(n,n)} q_0 & \xrightarrow{(t,n)} q_1 \\
(q_0, q_0) \xrightarrow{(T,N)} q_1 & (q_0, q_1) \xrightarrow{(N,T)} q_2 & (q_1, q_0) \xrightarrow{(N,T)} q_2 \\
(q_0, q_0) \xrightarrow{(N,N)} q_0 & (q_0, q_2) \xrightarrow{(N,N)} q_2 & (q_2, q_0) \xrightarrow{(N,N)} q_2
\end{array}
$$

Where the states intuitively correspond to the following :

* q_0 : the node is idle, i.e., the token is not in the node, nor in the subtree below the node;

* q_1 : the node is releasing the token to the node above it in the tree;
* q_2 : the token is either in the node or in a subtree below the node (accepting state).

The set of bad configurations can be taken to be the set of tree configurations in which at least two nodes have the token. This set can be encoded by the tree automaton $A_b = (Q_b, F_b, \delta_b)$, where $\Sigma = \{n, t, N, T\}$, $Q_b = \{q_{b_0}, q_{b_1}, q_{b_2}\}$, $F_b = \{q_{b_2}\}$, and δ_b is defined by the following set of rules:

$$\begin{array}{rrr}
\xrightarrow{n} q_{b_0} & \xrightarrow{t} q_{b_1} & (q_{b_0}, q_{b_0}) \xrightarrow{N} q_{b_0} \\
(q_{b_0}, q_{b_0}) \xrightarrow{T} q_{b_1} & (q_{b_0}, q_{b_1}) \xrightarrow{N} q_{b_1} & (q_{b_1}, q_{b_0}) \xrightarrow{N} q_{b_1} \\
(q_{b_0}, q_{b_1}) \xrightarrow{T} q_{b_2} & (q_{b_1}, q_{b_0}) \xrightarrow{T} q_{b_2} & (q_{b_1}, q_{b_1}) \xrightarrow{\{N,T\}} q_{b_2} \\
(q_{b_{01}}, q_{b_2}) \xrightarrow{\{N,T\}} q_{b_2} & (q_{b_2}, q_{b_{01}}) \xrightarrow{\{N,T\}} q_{b_2} & (q_{b_2}, q_{b_2}) \xrightarrow{\{N,T\}} q_{b_2}
\end{array}$$

where $q_{b_{01}} = \{q_{b_0}, q_{b_1}\}$.

Two Way Token Protocol

This example is an extension of the *Simple Token Protocol* above. The difference is that the token can move both upwards and downwards, in contrast to the Simple Token Protocol in which the token only moves upwards. The protocol is encoded using the same alphabet as the *Simple Token Protocol* above. The states we use are $\{q_0, q_1, q_2, q_3\}$. Their intuitive meaning is as follows:

* q_0: the node is idle, i.e., the token is not in the node, nor in the subtree below the node;
* q_1: the node is releasing the token to the node above it;
* q_2: the token is either in the node or in a subtree below the node (this state is accepting);
* q_3: the node is receiving the token from the node above it.

The transition relation is given by:

$$\begin{array}{rrr}
\xrightarrow{(n,n)} q_0 & \xrightarrow{(t,n)} q_1 & \xrightarrow{(n,t)} q_3 \\
(q_0, q_0) \xrightarrow{(T,N)} q_1 & (q_0, q_1) \xrightarrow{(N,T)} q_2 & (q_1, q_0) \xrightarrow{(N,T)} q_2 \\
(q_0, q_0) \xrightarrow{(N,N)} q_0 & (q_0, q_2) \xrightarrow{(N,N)} q_2 & (q_2, q_0) \xrightarrow{(N,N)} q_2 \\
(q_3, q_0) \xrightarrow{(T,N)} q_2 & (q_0, q_3) \xrightarrow{(T,N)} q_2 & (q_0, q_0) \xrightarrow{(N,T)} q_3
\end{array}$$

The initial and bad configurations are the same as for the Simple Token Protocol above.

The Percolate Protocol

This protocol simulates the way results propagate in a set of `OR` gates organized in a tree. At the leaves, we have a sequence of binary inputs. The nodes of the tree act as the `OR` gates. At first, every gate has its output set to *unknown*. At each step of the protocol, a gate for which each input is determined can set its

output to a definite value. The process iterates until the root gate has a definite value. Each process has a local variable with values $\{0,1\}$ for the leaf nodes and $\{U,0,1\}$ for the internal nodes[1], (U is interpreted as "undefined yet"). The system percolates the disjunction of values in the leaves up to the root.

The states we use are $Q = \{q_0, q_1, q_u, q_d\}$. Intuitively, these states correspond to the following

* q_0: All nodes below (and including) the current node are labeled with 0; Do not make any change to them;
* q_1: The current node and at least one node below is labeled with 1. No node below is labeled with U. Do not change the nodes below;
* q_u: All nodes above (and including) the current node have not yet been changed (they are still undefined);
* q_d: A single change has occurred in the current node or below (accepting state).

We use q_m to denote any member of $\{q_u, q_1, q_0\}$ in the transition relation δ given below :

$$\begin{array}{rrr}
\xrightarrow{(0,0)} q_0 & \xrightarrow{(1,1)} q_1 & (q_0,q_0) \xrightarrow{(0,0)} q_0 \\
(q_0,q_1) \xrightarrow{(1,1)} q_1 & (q_1,q_0) \xrightarrow{(1,1)} q_1 & (q_1,q_1) \xrightarrow{(1,1)} q_1 \\
(q_m,q_m) \xrightarrow{(U,U)} q_u & (q_m,q_d) \xrightarrow{(U,U)} q_d & (q_d,q_m) \xrightarrow{(U,U)} q_d \\
(q_0,q_0) \xrightarrow{(U,0)} q_d & (q_0,q_1) \xrightarrow{(U,1)} q_d & (q_1,q_0) \xrightarrow{(U,1)} q_d \\
(q_1,q_1) \xrightarrow{(U,1)} q_d & &
\end{array}$$

The set of initial configurations consists of all binary trees whose leaves are labeled with 0 or 1, while the rest of the nodes are labelled with U. This set is characterized with the following tree automaton

$$\begin{array}{rrr}
\xrightarrow{0} q_i & \xrightarrow{1} q_i & (q_i,q_i) \xrightarrow{U} q_{iu} \\
(q_{iu},q_{iu}) \xrightarrow{U} q_{iu} & (q_i,q_{iu}) \xrightarrow{U} q_{iu} & (q_{iu},q_i) \xrightarrow{U} q_{iu}
\end{array}$$

where the accepting state is q_{iu}.

The set of bad configurations can be taken to be the set of tree configurations where some node is labeled by 0 while one of its descendants is labeled by 1. These configurations are accepted by the tree automaton $A_b = (Q_b, F_b, \delta_b)$, where the alphabet is identical to the one of the Percolate protocol, $Q_b = \{q_{b_0}, q_{b_1}, q_{b_2}\}$, $F_b = \{q_{b_2}\}$, and δ_b is defined by the following set of rules:

$$\begin{array}{rrr}
\xrightarrow{0} q_{b_0} & \xrightarrow{1} q_{b_1} & (q_{b_0},q_{b_0}) \xrightarrow{\{U,0\}} q_{b_0} \\
(q_{b_{01}},q_{b_1}) \xrightarrow{\{U,0,1\}} q_{b_1} & (q_{b_1},q_{b_{01}}) \xrightarrow{\{U,0,1\}} q_{b_1} & (q_{b_0},q_{b_0}) \xrightarrow{1} q_{b_2} \\
(q_{b_{01}},q_{b_2}) \xrightarrow{\{U,0,1\}} q_{b_2} & (q_{b_2},q_{b_{01}}) \xrightarrow{\{U,0,1\}} q_{b_2} & (q_{b_2},q_{b_2}) \xrightarrow{\{U,0,1\}} q_{b_2}
\end{array}$$

[1]To simplify the notation, we do not distinguish between the nullary and binary versions of the symbols 0 and 1.

where $q_{b_{01}} = \{q_{b_0}, q_{b_1}\}$.

The Tree-Arbiter Protocol

(See e.g. [ABH$^+$97]) The tree arbiter is an asynchronous circuit that solves the mutual exclusion problem by building a tree of arbiter cells. The circuit works by performing elimination rounds: an arbiter cell arbitrates between its two children. The leaves of the tree are processors which may want to access asynchronously a shared resource. The n processors at the lowest level are arbitrated by $\frac{n}{2}$ cells. The winners of that level are arbitrated by the next level, and so forth. If both children of a cell are requesting the resource, then the cell chooses either of them non-deterministically. The requests propagate upwards until the root is reached. The root cell grants the resource to at most one child, and the grant propagates downward to one of the processors. When the processor is finished with the resource, it sends a release request that propagates upwards, until it encounters a cell that is aware of a request. That cell can either grant the resource to one of the requesting nodes, or continue to propagate the release signal. In our model of the protocol, any process can be labeled as follows:

* *idle*: the process does not do anything;
* *requesting*: the process wants to access the shared resource;
* *token*: the process has been granted the shared resource.

Furthermore, an interior process can be labeled as follows:

* *idle*: the process together with all the processes below are idle;
* *below*: the token is somewhere in one subtree below this node (but not in the node itself).

The alphabet we use is $\{i, r, t, b\}$ for respectively *idle*, *requesting*, *token*, and *below*.

When a leaf is in state *requesting*, the request is propagated upwards until it encounters a node which is aware of the presence of the token (i.e. a node that either owns the token or has a descendant which owns the token). If a node has the token it can always pass it upwards, or pass it downwards to a child which is requesting. Each time the token moves a step, the propagation moves a step or there is no move; the request and the token cannot propagate at the same time.

We now describe the transition relation. The states are $\{q_i, q_r, q_t, q_{req}, q_{rel}, q_{grant}, q_m, q_{rt}\}$. Intuitively, these states have the following meaning:

* q_i: Every node up to the current one is idle;
* q_r: Every node up to the current node are either idle or requesting, with at least one requesting. there was no move of the propagation below;
* q_t: The token is either in this node or below; token has not moved;
* q_{req}: The current node is requesting the token for itself or on behalf of a child : The request is being propagated;
* q_{rel}: The token is moving upwards from the current node;

* q_{grant}: The token is moving downwards to the current node;
* q_m: The token is in this node or below; the token has moved (this is an accepting state);
* q_{rt}: The token is either in this node or below it, i.e, nothing happens above the current node (this is an accepting state).

Using these states, the transition relation is given by:

$$
\begin{array}{rcl rcl rcl}
 & \xrightarrow{(i,i)} & q_i & & \xrightarrow{(i,r)} & q_{req} & & \xrightarrow{(r,r)} & q_r \\
 & \xrightarrow{(r,t)} & q_{grant} & & \xrightarrow{(t,t)} & q_t & & \xrightarrow{(t,i)} & q_{rel} \\
(q_i,q_i) & \xrightarrow{(i,i)} & q_i & (q_r,q_i) & \xrightarrow{(i,i)} & q_i & (q_i,q_r) & \xrightarrow{(i,i)} & q_i \\
(q_r,q_r) & \xrightarrow{(i,i)} & q_r & (q_{req},q_i) & \xrightarrow{(i,i)} & q_{req} & (q_i,q_{req}) & \xrightarrow{(i,i)} & q_{req} \\
(q_{req},q_r) & \xrightarrow{(i,i)} & q_{req} & (q_r,q_{req}) & \xrightarrow{(i,i)} & q_{req} & (q_r,q_r) & \xrightarrow{(i,r)} & q_{req}
\end{array}
$$

$$
\begin{array}{rcl rcl rcl}
(q_r,q_i) & \xrightarrow{(i,r)} & q_{req} & (q_i,q_r) & \xrightarrow{(i,r)} & q_{req} & (q_r,q_r) & \xrightarrow{(r,r)} & q_r \\
(q_r,q_i) & \xrightarrow{(r,r)} & q_r & (q_i,q_r) & \xrightarrow{(r,r)} & q_r & (q_{req},q_r) & \xrightarrow{(r,r)} & q_{req} \\
(q_r,q_{req}) & \xrightarrow{(r,r)} & q_{req} & (q_r,q_i) & \xrightarrow{(r,t)} & q_{grant} & (q_i,q_r) & \xrightarrow{(r,t)} & q_{grant} \\
(q_r,q_r) & \xrightarrow{(r,t)} & q_{grant} & (q_i,q_i) & \xrightarrow{(t,t)} & q_t & (q_i,q_r) & \xrightarrow{(t,t)} & q_t \\
(q_r,q_i) & \xrightarrow{(t,t)} & q_t & (q_r,q_r) & \xrightarrow{(t,t)} & q_t & (q_i,q_{req}) & \xrightarrow{(t,t)} & q_{rt} \\
(q_{req},q_i) & \xrightarrow{(t,t)} & q_{rt} & (q_r,q_{req}) & \xrightarrow{(t,t)} & q_{rt} & (q_{req},q_r) & \xrightarrow{(t,t)} & q_{rt} \\
(q_i,q_{grant}) & \xrightarrow{(t,b)} & q_m & (q_{grant},q_i) & \xrightarrow{(t,b)} & q_m & (q_{grant},q_r) & \xrightarrow{(t,b)} & q_m \\
(q_r,q_{grant}) & \xrightarrow{(t,b)} & q_m & (q_t,q_i) & \xrightarrow{(b,b)} & q_t & (q_t,q_r) & \xrightarrow{(b,b)} & q_t \\
(q_r,q_t) & \xrightarrow{(b,b)} & q_t & (q_i,q_t) & \xrightarrow{(b,b)} & q_t & (q_m,q_r) & \xrightarrow{(b,b)} & q_m \\
(q_m,q_i) & \xrightarrow{(b,b)} & q_m & (q_r,q_m) & \xrightarrow{(b,b)} & q_m & (q_i,q_m) & \xrightarrow{(b,b)} & q_m \\
(q_t,q_{req}) & \xrightarrow{(b,b)} & q_{rt} & (q_{req},q_t) & \xrightarrow{(b,b)} & q_{rt} & (q_r,q_{rt}) & \xrightarrow{(b,b)} & q_{rt} \\
(q_{rt},q_r) & \xrightarrow{(b,b)} & q_{rt} & (q_i,q_{rt}) & \xrightarrow{(b,b)} & q_{rt} & (q_{rt},q_i) & \xrightarrow{(b,b)} & q_{rt} \\
(q_i,q_{rel}) & \xrightarrow{(b,t)} & q_m & (q_{rel},q_i) & \xrightarrow{(b,t)} & q_m & (q_r,q_{rel}) & \xrightarrow{(b,t)} & q_m \\
(q_{rel},q_r) & \xrightarrow{(b,t)} & q_m & (q_i,q_i) & \xrightarrow{(t,i)} & q_{rel} & (q_i,q_r) & \xrightarrow{(t,r)} & q_{rel} \\
(q_r,q_i) & \xrightarrow{(t,r)} & q_{rel} & (q_r,q_r) & \xrightarrow{(t,r)} & q_{rel} & & &
\end{array}
$$

The set of initial configurations consists of binary trees where all the nodes are idle except the root which holds the token. This set is characterized with the

following tree automaton

$$
\begin{array}{rclrclrcl}
 & \xrightarrow{i} & q_{i_i} & & \xrightarrow{r} & q_{i_r} & (q_{i_i}, q_{i_i}) & \xrightarrow{i} & q_{i_i} \\
(q_{i_r}, q_{i_i}) & \xrightarrow{i} & q_{i_i} & (q_{i_i}, q_{i_r}) & \xrightarrow{i} & q_{i_i} & (q_{i_r}, q_{i_r}) & \xrightarrow{i} & q_{i_i} \\
(q_{i_i}, q_{i_i}) & \xrightarrow{t} & q_{i_t} & (q_{i_i}, q_{i_r}) & \xrightarrow{t} & q_{i_t} & (q_{i_r}, q_{i_i}) & \xrightarrow{t} & q_{i_t} \\
(q_{i_r}, q_{i_r}) & \xrightarrow{t} & q_{i_t} & (q_{i_i}, q_{i_t}) & \xrightarrow{b} & q_{i_b} & (q_{i_r}, q_{i_t}) & \xrightarrow{b} & q_{i_b} \\
(q_{i_t}, q_{i_i}) & \xrightarrow{b} & q_{i_b} & (q_{i_t}, q_{i_r}) & \xrightarrow{b} & q_{i_b} & (q_{i_i}, q_{i_b}) & \xrightarrow{b} & q_{i_b} \\
(q_{i_r}, q_{i_b}) & \xrightarrow{b} & q_{i_b} & (q_{i_b}, q_{i_i}) & \xrightarrow{b} & q_{i_b} & (q_{i_b}, q_{i_r}) & \xrightarrow{b} & q_{i_b}
\end{array}
$$

where q_{i_t} and q_{i_b} are the accepting states.

The set of bad configurations can be taken to be the set of tree configurations in which at least two nodes have the token. This set can be encoded by the tree automaton $A_b = (Q_b, F_b, \delta_b)$, with the same alphabet as the Tree Arbiter protocol, $Q_b = \{q_{b_0}, q_{b_1}, q_{b_2}\}$, $F_b = \{q_{b_2}\}$, and δ_b is defined by the following set of rules:

$$
\begin{array}{rclrclrcl}
 & \xrightarrow{\{i,r,b\}} & q_{b_0} & & \xrightarrow{t} & q_{b_1} & (q_{b_0}, q_{b_0}) & \xrightarrow{\{i,r,b\}} & q_{b_0} \\
(q_{b_0}, q_{b_0}) & \xrightarrow{T} & q_{b_1} & (q_{b_0}, q_{b_1}) & \xrightarrow{\{i,r,b\}} & q_{b_1} & (q_{b_1}, q_{b_0}) & \xrightarrow{\{i,r,b\}} & q_{b_1} \\
(q_{b_0}, q_{b_1}) & \xrightarrow{T} & q_{b_2} & (q_{b_1}, q_{b_0}) & \xrightarrow{T} & q_{b_2} & (q_{b_1}, q_{b_1}) & \xrightarrow{\{i,r,b\}} & q_{b_2} \\
(q_{b_{01}}, q_{b_2}) & \xrightarrow{\Sigma_2} & q_{b_2} & (q_{b_2}, q_{b_{01}}) & \xrightarrow{\Sigma_2} & q_{b_2} & (q_{b_2}, q_{b_2}) & \xrightarrow{\Sigma_2} & q_{b_2}
\end{array}
$$

where $q_{b_{01}} = \{q_{b_0}, q_{b_1}\}$ and $\Sigma_2 = \{i, r, t, b\}$.

The Leader Election Protocol

A set of processes placed in the leaves of a binary tree want to elect a leader. Each of them decides first whether to be a candidate or not. The election process proceeds in two phases. The first phase consists of the internal nodes polling their children nodes to see if at least one of them is candidate. In such a case, the internal node becomes a candidate as well. The second phase is the actual election procedure. The root chooses (elects) one candidate non-deterministically among its children. An internal node that has been elected, can elect one of its children that declared itself candidate.

In the protocol, the node is said to be elected if it has changed from candidate to elected. We use also undefined when the node has not been defined (i.e. the internal node does not know whether there are candidates below).

There are six states $\{q_c, q_n, q_{el}, q_u, q_{jel}, q_{ch}\}$ with the following intuitive meanings:

* q_c: There is at least a candidate in the tree below;
* q_n: No candidates below;
* q_{el}: The candidate to be elected is below (this is an accepting state);
* q_u: Undefined yet;
* q_{jel}: Just elected(this is an accepting state);

* q_{ch}: Something changed below(this is an accepting state).
The transition relation is given below:

$$
\begin{array}{lllllllll}
 & \xrightarrow{(c,c)} & q_c & & \xrightarrow{(c,el)} & q_{jel} & & \xrightarrow{(n,n)} & q_n \\
 & \xrightarrow{(el,el)} & q_{el} & (q_c,q_c) & \xrightarrow{(c,c)} & q_c & (q_c,q_n) & \xrightarrow{(c,c)} & q_c \\
(q_n,q_c) & \xrightarrow{(c,c)} & q_c & (q_c,q_n) & \xrightarrow{(c,el)} & q_{jel} & (q_c,q_c) & \xrightarrow{(c,el)} & q_{jel} \\
(q_n,q_c) & \xrightarrow{(c,el)} & q_{jel} & (q_n,q_n) & \xrightarrow{(n,n)} & q_n & (q_u,q_u) & \xrightarrow{(u,u)} & q_u \\
(q_n,q_u) & \xrightarrow{(u,u)} & q_u & (q_u,q_n) & \xrightarrow{(u,u)} & q_u & (q_u,q_c) & \xrightarrow{(u,u)} & q_u \\
(q_c,q_u) & \xrightarrow{(u,u)} & q_u & (q_c,q_c) & \xrightarrow{(u,u)} & q_u & (q_n,q_n) & \xrightarrow{(u,u)} & q_u
\end{array}
$$

$$
\begin{array}{lllllllll}
(q_n,q_c) & \xrightarrow{(u,u)} & q_u & (q_c,q_n) & \xrightarrow{(u,u)} & q_u & (q_{ch},q_u) & \xrightarrow{(u,u)} & q_{ch} \\
(q_{ch},q_n) & \xrightarrow{(u,u)} & q_{ch} & (q_{ch},q_c) & \xrightarrow{(u,u)} & q_{ch} & (q_u,q_{ch}) & \xrightarrow{(u,u)} & q_{ch} \\
(q_n,q_{ch}) & \xrightarrow{(u,u)} & q_{ch} & (q_c,q_{ch}) & \xrightarrow{(u,u)} & q_{ch} & (q_c,q_c) & \xrightarrow{(u,c)} & q_{ch} \\
(q_n,q_c) & \xrightarrow{(u,c)} & q_{ch} & (q_c,q_n) & \xrightarrow{(u,c)} & q_{ch} & (q_n,q_n) & \xrightarrow{(u,n)} & q_{ch} \\
(q_n,q_{el}) & \xrightarrow{(el,el)} & q_{el} & (q_c,q_{el}) & \xrightarrow{(el,el)} & q_{el} & (q_{el},q_n) & \xrightarrow{(el,el)} & q_{el} \\
(q_{el},q_c) & \xrightarrow{(el,el)} & q_{el} & (q_n,q_{jel}) & \xrightarrow{(el,el)} & q_{el} & (q_c,q_{jel}) & \xrightarrow{(el,el)} & q_{el} \\
(q_{jel},q_n) & \xrightarrow{(el,el)} & q_{el} & (q_{jel},q_c) & \xrightarrow{(el,el)} & q_{el} & & &
\end{array}
$$

The set of initial configurations is given by the following tree automaton, where q_0 is the only accepting state:

$$
\begin{array}{lllllllll}
 & \xrightarrow{c} & q_0 & & \xrightarrow{n} & q_1 & (q_0,q_0) & \xrightarrow{u} & q_0 \\
(q_0,q_1) & \xrightarrow{u} & q_0 & (q_1,q_0) & \xrightarrow{u} & q_0 & (q_1,q_1) & \xrightarrow{u} & q_1
\end{array}
$$

The set of bad configurations can be taken to be the set of tree configurations where at least two leaves have been elected. This set can be encoded by the tree automaton $A_b = (Q_b, F_b, \delta_b)$, where the alphabet is identical to the one of the Leader Election protocol, $Q_b = \{q_{b_0}, q_{b_1}, q_{b_2}\}$, $F_b = \{q_{b_2}\}$, and δ_b is defined by the following set of rules:

$$
\begin{array}{lllllllll}
 & \xrightarrow{\{n,c\}} & q_{b_0} & & \xrightarrow{el} & q_{b_1} & (q_{b_0},q_{b_0}) & \xrightarrow{\Sigma_2} & q_{b_0} \\
(q_{b_0},q_{b_1}) & \xrightarrow{\Sigma_2} & q_{b_1} & (q_{b_1},q_{b_0}) & \xrightarrow{\Sigma_2} & q_{b_1} & (q_{b_1},q_{b_1}) & \xrightarrow{\Sigma_2} & q_{b_2} \\
(q_{b_{01}},q_{b_2}) & \xrightarrow{\Sigma_2} & q_{b_2} & (q_{b_2},q_{b_{01}}) & \xrightarrow{\Sigma_2} & q_{b_2} & (q_{b_2},q_{b_2}) & \xrightarrow{\Sigma_2} & q_{b_2}
\end{array}
$$

where $q_{b_{01}} = \{q_{b_0}, q_{b_1}\}$, and $\Sigma_2 = \{n, c, u, el\}$.

6.3 Conclusions

In this chapter, we introduced in two parts the framework of regular model checking for the verification of parameterized systems. First, we looked at the

case of linear parameterized systems. We introduced representations of configurations and transitions based on automata and transducers. Given a finite transducer encoding the transitive closure of the transition relation induced by a parameterized system, one can verify safety and liveness properties. We explained that the transitive closure cannot always be encoded as a finite transducer. Then, we generalized the representations to the case of parameterized systems where the components are placed in tree structures. For this purpose, we introduced tree automata and tree transducers, and used them to model several parameterized systems. In the next chapter, we propose a method to compute a tree transducer that characterizes the transitive closure of the regular relation defined by a tree transducer. We use this method to verify the tree parameterized systems introduced in this chapter.

7. Transitive Closure of Tree Transducers

We provide an efficient and implementable semi-algorithm for computing the transitive closure of a tree transducer of arity two. In the following, and unless otherwise specified, the arities of the transducers are equal to two. Starting from a tree transducer D describing the set of transitions of the system, we derive a transducer, called the *history transducer* whose states are *columns* (words) of states of D. The history transducer characterizes the transitive closure of the rewriting relation corresponding to D. The set of states of the history transducer is infinite, which makes it inappropriate for computational purposes. Therefore, we present a method for computing a finite-state transducer which is an abstraction of the history transducer. The abstract transducer is generated on-the-fly by a procedure which starts from the original transducer D, and then incrementally adds new transitions and merges equivalent states. To compute the abstract transducer, we define an equivalence relation on columns (states of the history transducer). We identify *good* equivalence relations, i.e., equivalence relations which can be used by our on-the-fly algorithm. An equivalence relation is considered to be *good* if it satisfies the following two conditions:

- *Soundness and completeness:* merging two equivalent columns must not add traces which are not present in the history transducer. Consequently, the abstract transducer accepts the same language as the history transducer (and therefore characterizes exactly the transitive closure of D).
- *Computability of the equivalence relation:* This allows on-the-fly merging of equivalent states during the generation of the abstract transducer.

We present a methodology for deriving good equivalence relations. More precisely, an equivalence relation is induced by two simulation relations; namely a *downward* and an *upward* simulation relation, both of which are defined on tree automata. We provide sufficient conditions on the simulation relations which guarantee that the induced equivalence is good. Furthermore, we give examples of concrete simulations which satisfy the sufficient conditions.

We also show that our technique can be directly adapted in order to compute the set of reachable states of a system without computing the transitive closure. When checking for safety properties, such an approach is often (but not always) more efficient.

Outline:
In the next section, we introduces *history transducers* which characterize the transitive closure of a given transducer. In section 7.2, we introduce *downward* and *upward* simulations on tree automata, and give sufficient conditions which guarantee that the induced equivalence relation is exact and computable. Section 7.3 gives an example of simulations which satisfy the sufficient conditions. In section 7.4, we describe how to compute the reachable states. In section 7.5 we report on the results of running a prototype on the tree examples introduced in subsection 6.2.3. Finally, we conclude in section 7.6.

7.1 Computing the Transitive Closure

In this section we introduce the notion of *history transducer*. With a transducer D we associate a *history transducer* H which corresponds to the transitive closure of D. Each state of H is a word of the form $q_1 \cdots q_k$, where $q_1, \ldots, q_k$ are states of D. For a word w, we let $w(i)$ denote the i-th symbol of w. Intuitively, for each $(T, T') \in rel(D^+)$, the history transducer H encodes the successive runs of D needed to derive T' from T. The term "history transducer" reflects the fact that the transducer encodes the histories of all such derivations.

Definition 7.1 (History Transducer)**.** *Consider a tree transducer $D = (Q, \Sigma, I, \Delta, F)$ over a ranked alphabet Σ. The history (tree) transducer H for D is an (infinite) transducer (Q_H, F_H, δ_H), where $Q_H = Q^+$, $F_H = F^+$, and δ_H contains all rules of the form*

$$(w_1, \ldots, w_p) \overset{(f,f')}{\longrightarrow} w$$

such that there is $k \geq 1$ where the following conditions are satisfied

- $|w_1| = \cdots = |w_p| = |w| = k$;
- *there are $f_1, f_2, \ldots, f_{k+1}$, with $f = f_1$, $f' = f_{k+1}$, and* $(w_1(i) \ldots, w_p(i)) \overset{(f_i, f_{i+1})}{\longrightarrow} w(i)$ *belongs to δ, for each $i : 1 \leq i \leq k$.*

Observe that all the symbols $f_1, \ldots, f_{k+1}$ are of the same arity p. Also, notice that if $(T \times T') \overset{r}{\Longrightarrow}_H w$, then there is a $k \geq 1$ such that $|r(n)| = k$ for each $n \in (T \times T')$. In other words, any run of the history transducer assigns states (words) of the same length to the nodes. From the definition of H we derive the following lemma which states that H characterizes the transitive closure of the relation of D.

Lemma 7.1. *For a transducer $D = (Q, \Sigma, I, \Delta, F)$ and its history transducer $H = (Q_H, F_H, \delta_H)$, we have that $rel(H) = rel\,(D^+)$.*

Proof. We show inclusion in both direction. $rel(H) \subseteq rel(D^+)$:

Consider $(T, T') \in rel(H)$, and let r be an accepting run, i.e. $(T \times T') \overset{r}{\Longrightarrow}_H F_H$. Let $k = |r(\varepsilon)|$ be the size of the states encountered in the run r. For $i : 1 \leq$

$i \le k$, and for each node n in the structure of tree T, let $r_i(n) = (r(n))(i)$. We show that $r_1, \ldots, r_k$ are successive runs of D.

By definition of a run, for each node n with children $n_1, \ldots, n_p$, there is a rule: $(r(n_1), \ldots, r(n_p)) \xrightarrow{(f,f')} r(n) \in \delta_H$.

By definition of H, there exist symbols $f_1, \ldots, f_{k+1}$ with $f = f_1$ and $f' = f_{k+1}$, such that $(r(n_1)(i), \ldots, r(n_p)(i)) \xrightarrow{(f_i,f_{i+1})} r(n)(i) \in \delta$, for each $i : 1 \le i \le k$. We let $T_1, \ldots, T_{k+1}$ be (structurally equivalent) trees such that for node n, their labeling function is given by $\lambda_1(n) = f_1, \ldots, \lambda_{k+1}(n) = f_{k+1}$.

Observe that the rule $(r(n_1)(i), \ldots, r(n_p)(i)) \xrightarrow{(f_i,f_{i+1})} r(n)(i)$ can be rewritten as: $(r_i(n_1), \ldots, r_i(n_p)) \xrightarrow{(f_i,f_{i+1})} r_i(n)$. We conclude that each r_i is indeed a run of D. Notice that $T = T_1$ and $T' = T_{k+1}$. Furthermore, for each $i : 1 \le i \le k$, $T_i \times T_{i+1} \overset{r_i}{\Longrightarrow}_D F$, i.e. each pair (T_i, T_{i+1}) is accepted by D with run r_i.

Hence, we conclude that $(T, T') \in rel(D^k) \subseteq rel(D^+)$.

$rel(H) \supseteq rel(D^+)$:

Conversely, suppose that $(T, T') \in rel(D^+)$. Let k be the smallest integer such that $(T, T') \in rel(D^k) \subseteq rel(D^+)$. By definition of composition, there exist structurally equivalent trees $T_1, \ldots, T_{k+1}$ with labeling functions $\lambda_1, \ldots, \lambda_{k+1}$ such that $T = T_1$, $T' = T_{k+1}$, and for each $i : 1 \le i \le k$, $(T_i, T_{i+1}) \in rel(D)$. For each $T_i \times T_{i+1}$, let r_i be an accepting run of D. Finally, let r be the mapping $r(n) = r_1(n) \cdots r_k(n)$ for each node n.

Observe that for each node n with children $n_1, \ldots, n_p$, the following holds:

- $|r(n)| = |r(n_1)| = \ldots = |r(n_p)| = k$;
- for each $i : 1 \le i \le k$, $(r(n_1)(i), \ldots, r(n_p)(i)) \xrightarrow{(\lambda_i(n),\lambda_{i+1}(n))} r(n)(i) \in \delta$.

Hence, for each node n, there is a rule: $(r(n_1), \ldots, r(n_p)) \xrightarrow{(\lambda_1(n),\lambda_{k+1}(n))} r(n) \in \delta_H$, Thus, r is a run of H that accepts $T \times T'$. We conclude that $(T, T') \in rel(H)$. $\square$

The problem with H is that it has infinitely many states. Therefore, we define an *equivalence* $\simeq$ on the states of H, and construct a new transducer where equivalent states are merged. This new transducer will hopefully only have a finite number of states.

Given an equivalence relation $\simeq$, the symbolic transducer $D_\simeq$ obtained by merging states of H according to $\simeq$ is defined as $(Q/\simeq, F/\simeq, \delta_\simeq)$, where:

- $Q/\simeq$ is the set of equivalence classes of Q_H w.r.t. $\simeq$;
- $F/\simeq$ is the set of equivalence classes of F_H w.r.t. $\simeq$ (this will always be well-defined, see condition 5 of Sufficient Conditions 7.1, in section 7.2.3);
- $\delta_\simeq$ contains rules of the form $(x_1, \ldots, x_n) \xrightarrow{f}_\simeq x$ if and only if there are states $w_1 \in x_1, \ldots, w_n \in x_n, w \in x$ such that there is a rule equal to $(w_1, \ldots, w_n) \xrightarrow{f} w \in \delta_H$ in H.

Since H is infinite we cannot derive $D_\simeq$ by first computing H. Instead, we compute $D_\simeq$ on-the-fly, collapsing states which are equivalent according to $\simeq$.

In other words, we perform the following *procedure* (which need not terminate in general).

- The procedure computes successive reflexive powers of D: $D^{\leq 1}, D^{\leq 2}, D^{\leq 3}, \ldots$ (where $D^{\leq i} = \bigcup_{n=1}^{n=i} D^n$ and $D^n = D \circ D^{n-1}$), and collapses states[1] according to $\simeq$. We thus obtain $D_{\simeq}^{\leq 1}, D_{\simeq}^{\leq 2}, \ldots$
- The procedure terminates when the relation $rel(D^+)$ is accepted by $D_{\simeq}^{\leq i}$. This can be tested by checking if the language of $D_{\simeq}^{\leq i} \circ D$ is included in the language of $D_{\simeq}^{\leq i}$.

In the next section, we explain how we can make the above procedure sound, complete, and implementable.

7.2 Soundness, Completeness, and Computability

In this section, we describe how to derive equivalence relations on the states of the history transducer which can be used in the procedure given in section 7.1. A *good* equivalence relation $\simeq$ satisfies the following two conditions:

- It is sound and complete, i.e., $rel(D_{\simeq}) = rel(H)$. This means that $D_{\simeq}$ characterizes the same relation as D^+.
- It is computable. This turns the procedure of section 7.1 into an *implementable algorithm*, since it allows on-the-fly merging of equivalent states.

We provide a methodology for deriving such a good equivalence relations as follows:

1. In 7.2.1 we define two simulation relations; namely a downward simulation relation $\preccurlyeq_{down}$ and an *upward simulation relation* $\preccurlyeq_{up}$.
2. In 7.2.2, an upward and a downward simulation are put together to induce an equivalence relation $\simeq$.
3. Next, in 7.2.3 we give sufficient conditions on the simulation relations which guarantee that the induced equivalence $\simeq$ is sound and complete.
4. Finally, we focus in 7.2.4 on the computability of $\simeq$.

7.2.1 Downward and Upward Simulation

We start by giving the definitions.

Definition 7.2 (Downward Simulation). *Let $A = (Q, F, \delta)$ be a tree automaton over Σ. A binary relation $\preccurlyeq_{down}$ on the states of A is a downward simulation iff for any $n \geq 1$ and any symbol $f \in \Sigma_n$, for all states $q, q_1, \ldots, q_n, r \in Q$, the following holds:*
Whenever $q \preccurlyeq_{down} r$ and $(q_1, \ldots, q_n) \xrightarrow{f} q \in \delta$, there exist states $r_1, \ldots, r_n \in Q$ such that $q_1 \preccurlyeq_{down} r_1, \ldots, q_n \preccurlyeq_{down} r_n$ and $(r_1, \ldots, r_n) \xrightarrow{f} r \in \delta$.

[1]The states of $D^{\leq i}$ are by construction states of the history transducer.

Definition 7.3 (Upward Simulation)**.** *Let $A = (Q, F, \delta)$ be a tree automaton over Σ. Given a downward simulation $\preccurlyeq_{down}$, a binary relation $\preccurlyeq_{up}$ on the states of A is an upward simulation w.r.t. $\preccurlyeq_{down}$ iff for any $n \geq 1$ and any symbol $f \in \Sigma_n$, for all states $q, q_1, \ldots, q_i, \ldots, q_n, r_i \in Q$, the following holds:*
Whenever $q_i \preccurlyeq_{up} r_i$ and $(q_1, \ldots, q_n) \xrightarrow{f} q \in \delta$, there exist states $r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n, r \in Q$ such that $q \preccurlyeq_{up} r$ and $\forall j \neq i : q_j \preccurlyeq_{down} r_j$ and $(r_1, \ldots, r_n) \xrightarrow{f} r \in \delta$.

While the notion of a downward simulation is a straightforward extension of the word case [AJNd03], the notion of an upward simulation is not as obvious. This comes from the asymmetric nature of trees. If we follow the execution of a tree automaton downwards, it is easy to see that all respective children of two nodes related by simulation should continue to be related pairwise. If we now consider how a tree automaton executes when going upwards, we are confronted to the problem that the parent of the current node may have several children. The question is then how to characterize the behavior of such children. The answer lies in constraining their prefixes, i.e. using a downward simulation.

We state some elementary properties of the simulation relations

Lemma 7.2. *Let $A = (Q, F, \delta)$ be a tree automaton. Let $\preccurlyeq_{down}$ be a relation on the states of A which is a downward simulation. The reflexive closure and the transitive closure of $\preccurlyeq_{down}$ are both downward simulations. Furthermore, there is a unique maximal downward simulation.*

Proof. We consider the three claims.

- *Reflexivity*: Let $\preccurlyeq^1_{down} = \preccurlyeq_{down} \cup Id$. We show that $\preccurlyeq^1_{down}$ is also a downward simulation. Assume $(q_1, \ldots, q_n) \xrightarrow{f} q \in \delta$ and $q \preccurlyeq^1_{down} r$. We find states $r_1, \ldots, r_n \in Q$ such that $q_1 \preccurlyeq^1_{down} r_1, \ldots, q_n \preccurlyeq^1_{down} r_n$ and $(r_1, \ldots, r_n) \xrightarrow{f} r \in \delta$ as follows:
  - If $q = r$, then we choose $r_1 = q_1, \ldots, r_n = q_n$. Observe that $\preccurlyeq^1_{down} \supseteq Id$, implies $q_1 \preccurlyeq^1_{down} r_1, \ldots, q_n \preccurlyeq^1_{down} r_n$. Thus, the claim holds.
  - If $q \preccurlyeq_{down} r$, then we apply the hypothesis that $\preccurlyeq_{down}$ is a downward simulation, and conclude that there exist $r_1, \ldots, r_n \in Q$ such that $(r_1, \ldots, r_n) \xrightarrow{f} r \in \delta$ and $q_1 \preccurlyeq_{down} r_1, \ldots, q_n \preccurlyeq_{down} r_n$. From $\preccurlyeq^1_{down} \supseteq \preccurlyeq_{down}$, we conclude that the claim holds.
- *Transitivity*: Let $\preccurlyeq^1_{down}$ be the transitive closure of $\preccurlyeq_{down}$. We show that $\preccurlyeq^1_{down}$ is also a downward simulation. Assume $(q_1, \ldots, q_n) \xrightarrow{f} q \in \delta$ and $q \preccurlyeq^1_{down} r$. We find states $r_1, \ldots, r_n \in Q$ such that $q_1 \preccurlyeq^1_{down} r_1, \ldots, q_n \preccurlyeq^1_{down} r_n$ and $(r_1, \ldots, r_n) \xrightarrow{f} r \in \delta$ as follows:
  - If $q \preccurlyeq_{down} r$, then the claim trivially holds.
  - If there is $s \in Q$ such that $q \preccurlyeq_{down} s \preccurlyeq_{down} r$, then apply the hypothesis that $\preccurlyeq_{down}$ is a downward simulation with $q \preccurlyeq_{down} s$,

and find states $s_1,\ldots,s_n \in Q$ with $(s_1,\ldots,s_n) \xrightarrow{f} s \in \delta$ and $q_1 \preccurlyeq_{down} s_1,\ldots,q_n \preccurlyeq_{down} s_n$. Now, we apply this a second step using $s \preccurlyeq_{down} r$, and find states $r_1,\ldots,r_n \in Q$ such that $(r_1,\ldots,r_n) \xrightarrow{f} r \in \delta$ and $s_1 \preccurlyeq_{down} r_1,\ldots,s_n \preccurlyeq_{down} r_n$. By transitivity, we get $q_1 \preccurlyeq^1_{down} r_1,\ldots,q_n \preccurlyeq^1_{down} r_n$. Thus, the claim holds.

Observe that in the second alternative above, we only treat the case of one step transitivity. Arbitrary transitivity follows by induction on the number of steps.

- *Uniqueness:* Assume two maximal downward simulations $\preccurlyeq^1_{down}$ and $\preccurlyeq^2_{down}$. Let $\preccurlyeq_{down}=\preccurlyeq^1_{down} \cup \preccurlyeq^2_{down}$. We show that $\preccurlyeq_{down}$ is also a simulation. Assume $(q_1,\ldots,q_n) \xrightarrow{f} q \in \delta$ and $q \preccurlyeq_{down} r$. We find states $r_1,\ldots,r_n \in Q$ such that $q_1 \preccurlyeq_{down} r_1,\ldots,q_n \preccurlyeq_{down} r_n$ and $(r_1,\ldots,r_n) \xrightarrow{f} r \in \delta$ as follows:
  - If $q \preccurlyeq^1_{down} r$, then since $\preccurlyeq^1_{down}$ is a simulation, and $\preccurlyeq_{down}\supseteq\preccurlyeq^1_{down}$, the claim holds.
  - If $q \preccurlyeq^2_{down} r$, then since $\preccurlyeq^2_{down}$ is a simulation, and $\preccurlyeq_{down}\supseteq\preccurlyeq^2_{down}$, the claim holds.

  We have $\preccurlyeq_{down}\supseteq\preccurlyeq^1_{down}$ and $\preccurlyeq_{down}\supseteq\preccurlyeq^2_{down}$. Now, if we assume $\preccurlyeq^1_{down}\neq\preccurlyeq^2_{down}$, we get either $\preccurlyeq_{down}\supset\preccurlyeq^1_{down}$ or $\preccurlyeq_{down}\supset\preccurlyeq^2_{down}$. This violates the maximality of either $\preccurlyeq^1_{down}$ or $\preccurlyeq^2_{down}$.

□

Lemma 7.3. *Let $A = (Q,F,\delta)$ be a tree automaton. Let $\preccurlyeq_{down}$ be a reflexive (transitive) downward simulation on the states of A. The reflexive (transitive) closure of an upward simulation w.r.t to $\preccurlyeq_{down}$ is also an upward simulation w.r.t $\preccurlyeq_{down}$. Furthermore there exists a unique maximal upward simulation w.r.t. any downward simulation.*

Proof. We consider the three claims.

- *Reflexivity*: Let $\preccurlyeq^1_{up}=\preccurlyeq_{up} \cup Id$. We show that $\preccurlyeq^1_{up}$ is also an upward simulation. Assume $(q_1,\ldots,q_i,\ldots,q_n) \xrightarrow{f} q \in \delta$ and $q_i \preccurlyeq^1_{up} r_i$. We find states $r_1,\ldots,r_{i-1},r_{i+1},\ldots,r_n,r \in Q$ such that $q \preccurlyeq^1_{up} r$ and $\forall j \neq i : q_j \preccurlyeq_{down} r_j$ and $(r_1,\ldots,r_n) \xrightarrow{f} r \in \delta$ as follows:
  - If $q_i \preccurlyeq_{up} r_i$, then since $\preccurlyeq_{up}$ is an upward simulation, and $\preccurlyeq^1_{up}\supseteq\preccurlyeq_{up}$, the claim trivially holds.
  - If $q_i = r_i$, then we choose $r = q$ and $r_j = q_j$ for each $j \neq i$. By reflexivity of $\preccurlyeq_{down}$, we have $q_j \preccurlyeq_{down} r_j$ for each $j \neq i$. Since $\preccurlyeq^1_{up}\supset Id$, we also have $q \preccurlyeq^1_{up} r$. Hence, the claim holds.
- *Transitivity*: Let $\preccurlyeq^1_{up}$ be the transitive closure of $\preccurlyeq_{up}$. We show that $\preccurlyeq^1_{up}$ is also an upward simulation. Assume $(q_1,\ldots,q_i,\ldots,q_n) \xrightarrow{f} q \in \delta$ and $q_i \preccurlyeq^1_{up} r_i$. We find states $r_1,\ldots,r_{i-1},r_{i+1},\ldots,r_n,r \in Q$ such that $q \preccurlyeq^1_{up} r$ and $\forall j \neq i : q_j \preccurlyeq_{down} r_j$ and $(r_1,\ldots,r_n) \xrightarrow{f} r \in \delta$ as follows:

- If $q_i \preccurlyeq_{up} r_i$, then since $\preccurlyeq_{up}$ is an upward simulation, and $\preccurlyeq_{up}^1 \supseteq \preccurlyeq_{up}$, the claim trivially holds.
- If there is s_i with $q_i \preccurlyeq_{up} s_i \preccurlyeq_{up} r_i$, then since $q_i \preccurlyeq_{up} s_i$, we apply the hypothesis that $\preccurlyeq_{up}$ is an upward simulation. We get states $s, s_1, \ldots, s_n \in Q$ with $(s_1, \ldots, s_i, \ldots, s_n) \xrightarrow{f} s \in \delta$ and $q \preccurlyeq_{up} s$ and for each $j \neq i$, $q_j \preccurlyeq_{down} s_j$. With $s_i \preccurlyeq_{up} r_i$, we use simulation a second time, and get states $r, r_1, \ldots, r_n \in Q$ with $(r_1, \ldots, r_i, \ldots, r_n) \xrightarrow{f} r \in \delta$ and $s \preccurlyeq_{up} r$ and for each $j \neq i$, $s_j \preccurlyeq_{down} r_j$. By transitivity of $\preccurlyeq_{down}$, we get for each $j \neq i$, $q_j \preccurlyeq_{down} r_j$. By transitivity of $\preccurlyeq_{up}^1$, we also get $q \preccurlyeq_{up}^1 r$. Hence, the claim holds.

Observe that in the second alternative above, we only treat the case of one step transitivity. Arbitrary transitivity follows by induction on the number of steps.

- *Uniqueness*: Assume two maximal upward simulations $\preccurlyeq_{up}^1$ and $\preccurlyeq_{up}^2$. Let $\preccurlyeq_{up} = \preccurlyeq_{up}^1 \cup \preccurlyeq_{up}^2$. We show that $\preccurlyeq_{up}$ is also a simulation. Assume $(q_1, \ldots, q_i, \ldots, q_n) \xrightarrow{f} q \in \delta$ and $q_i \preccurlyeq_{up} r_i$. We find states $r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n, r \in Q$ such that $q \preccurlyeq_{up} r$ and $\forall j \neq i : q_j \preccurlyeq_{down} r_j$ and $(r_1, \ldots, r_n) \xrightarrow{f} r \in \delta$ as follows:
  - If $q_i \preccurlyeq_{up}^1 r_i$, then since $\preccurlyeq_{up}^1$ is a simulation, and $\preccurlyeq_{up} \supseteq \preccurlyeq_{up}^1$, the claim holds.
  - If $q_i \preccurlyeq_{up}^2 r_i$, then since $\preccurlyeq_{up}^2$ is a simulation, and $\preccurlyeq_{up} \supseteq \preccurlyeq_{up}^2$, the claim holds.

  We have $\preccurlyeq_{up} \supseteq \preccurlyeq_{up}^1$ and $\preccurlyeq_{up} \supseteq \preccurlyeq_{up}^2$. Now, if we assume $\preccurlyeq_{up}^1 \neq \preccurlyeq_{up}^2$, we get either $\preccurlyeq_{up} \supset \preccurlyeq_{up}^1$ or $\preccurlyeq_{up} \supset \preccurlyeq_{up}^2$. This violates the maximality of either $\preccurlyeq_{up}^1$ or $\preccurlyeq_{up}^2$.

$\square$

Observe that both for downward simulations, and upward simulations, maximality implies transitivity and reflexivity.

7.2.2 Induced Equivalence Relation

We now define an equivalence relation derived from two binary relations.

Definition 7.4 (Independence)**.** *Two binary relations $\preceq_1$ and $\preceq_2$ are said to be* independent *iff whenever $q \preceq_1 r$ and $q \preceq_2 r'$, there exists s such that $r \preceq_2 s$ and $r' \preceq_1 s$.*

Definition 7.5 (Induced Relation)**.** *The relation $\simeq$* induced *by two binary relations $\preceq_1$ and $\preceq_2$ is defined as:*

$$\preceq_1 \circ \preceq_2^{-1} \cap \preceq_2 \circ \preceq_1^{-1} .$$

The following Lemma gives sufficient conditions for two relations to induce an equivalence relation.

Lemma 7.4. *Let $\preceq_1$ and $\preceq_2$ be two binary relations. If $\preceq_1$ and $\preceq_2$ are reflexive, transitive, and independent, then their induced relation $\simeq$ is an* equivalence relation.

Proof. 1. If $\preceq_1$ and $\preceq_2$ are reflexive, then for any q we have $q \preceq_1 q \preceq_2^{-1} q$ and $q \preceq_2 q \preceq_1^{-1} q$. Thus, we have $q \simeq q$.
2. If $q \simeq r$, then $q \preceq_1 \circ \preceq_2^{-1} r$ and $q \preceq_2 \circ \preceq_1^{-1} r$. We can rewrite this $r \preceq_1 \circ \preceq_2^{-1} q$ and $r \preceq_2 \circ \preceq_1^{-1} q$. Hence, $r \simeq q$.
3. Assume $q \simeq r \simeq s$. Then by definition of $\simeq$, we can find t', t'' such that $q \preceq_1 t' \preceq_2^{-1} r$ and $r \preceq_1 t'' \preceq_2^{-1} s$. Since $\preceq_1$ and $\preceq_2$ are independent, there is t such that $t' \preceq_1 t$ and $t'' \preceq_2 t$. By transitivity, we get $q \preceq_1 t \preceq_2^{-1} s$. Hence, $q \preceq_1 \circ \preceq_2^{-1} s$. Similarly, we can get $q \preceq_2 \circ \preceq_1^{-1} s$, and finally conclude that $q \simeq s$

□

To conclude, we state a property of $\simeq$ which follows from independence.

Lemma 7.5. *Let $\preceq_1$ and $\preceq_2$ be both reflexive and transitive, and $\simeq$ be their induced relation. Furthermore, let $\preceq_1$ and $\preceq_2$ be independent. Whenever $x \simeq y$ and $x \preceq_1 z$, there exists t such that $y \preceq_1 t$ and $z \preceq_2 t$.*

Proof. Assume $x \simeq y$ and $x \preceq_1 z$. By definition of $\simeq$, we know that there is u with $x \preceq_2 u$ and $y \preceq_1 u$. We apply the definition of independence to x, u, z, and conclude that there is a state t such that $z \preceq_2 t$ and $u \preceq_1 t$. By transitivity of $\preceq_1$, we have $y \preceq_1 t$. □

7.2.3 Sufficient Conditions for Soundness and Completeness

We give sufficient conditions for two simulation relations to induce a sound and complete equivalence relation on states of a tree automaton. We assume a tree automaton $A = (Q, \Sigma, I, \Delta, F)$. We now define a relation $\simeq$ on the states of A, induced by the two relations $\preceq$ and $\preccurlyeq_{down}$ both on the states of A, satisfying the following sufficient conditions:

Sufficient Conditions 7.1. *The sufficient conditions are:*
1. *$\preccurlyeq_{down}$ is a reflexive downward simulation;*
2. *$\preceq$ is a reflexive and transitive relation included in $\preccurlyeq_{up}$ which is an upward simulation w.r.t. $\preccurlyeq_{down}$;*
3. *$\preccurlyeq_{down}$ and $\preceq$ are independent;*
4. *whenever $x \in F$ and $x \preccurlyeq_{up} y$, then $y \in F$;*
5. *F is a union of equivalence classes w.r.t. $\simeq$;*
6. *whenever $\xrightarrow{f} x$ and $x \preccurlyeq_{down} y$, then $\xrightarrow{f} y$.*

□

We first obtain the following Lemma which shows that if the simulations satisfy the above Sufficient Conditions, then the induced relation is indeed an equivalence.

Lemma 7.6. *Let $A = (Q, \Sigma, I, \Delta, F)$ be a tree automaton. Consider two binary relations $\preccurlyeq_{down}$ and $\preceq$ on the states of A, which satisfy Sufficient Conditions 7.1, as well as their induced relation $\simeq$. Then $\simeq$ is an equivalence relation on states of A.*

Proof. The claim holds since Conditions 1 through 3 of Sufficient Conditions 7.1 above imply directly that $\preccurlyeq_{down}$ and $\preceq$ satisfy the hypothesis needed by Lemma 7.4. □

We then prove that an equivalence relation satisfying Sufficient Conditions 7.1 is sound. This result is stated in Theorem 7.1. Lemma 7.7 is an intermediate result.

We first show that the tree automaton $A_\simeq$ has the same traces as A.

Lemma 7.7. *Let $A = (Q, \Sigma, I, \Delta, F)$ be a tree automaton. Consider two binary relations $\preccurlyeq_{down}$ and $\preceq$ on the states of A, satisfying Sufficient Conditions 7.1, and let $\simeq$ be their induced relation. Let $A_\simeq = (Q/\simeq, F/\simeq, \delta_\simeq)$ be the automaton obtained by merging the states of A according to $\simeq$. For any states $Z_1, \ldots, Z_k, Z$ of $A_\simeq$ and context C,*
if $C[Z_1, \ldots, Z_k] \Longrightarrow_\simeq Z$, then there exist states $z_1, \ldots, z_k, z$ and states $t_1, \ldots, t_k, t$ of A such that $C[t_1, \ldots, t_k] \Longrightarrow t$ and $z_1 \in Z_1, \ldots, z_k \in Z_k, z \in Z$ and $z_1 \preccurlyeq_{down} t_1, \ldots, z_k \preccurlyeq_{down} t_k, z \preccurlyeq_{up} t$.

Proof. The claim is shown by induction on the structure of C.
Base case: C contains only a hole. We choose a $z \in Z$. By reflexivity of $\preccurlyeq_{down}$ and $\preccurlyeq_{up}$, the claim obviously holds.
Induction case: C is not just a hole. Consider a run r of $A_\simeq$ on $C = (S_C, \lambda_C)$ satisfying
$C[Z_1, \ldots, Z_k] \Longrightarrow_\simeq Z$. Let $n_1, \ldots, n_j$ be the left-most leaves of C with a common parent. Let n be the parent of $n_1, \ldots, n_j$. Note that $Z_1 = r(n_1), \ldots, Z_j = r(n_j)$. Let $Y = r(n)$.

We let C' be the context C, with the leaves $n_1, \ldots, n_j$ deleted. In other words $C' = (S'_C, \lambda'_C)$ where $S'_C = S_C - \{n_1, \ldots, n_j\}$, $\lambda'_C(n') = \lambda_C(n')$ if $n' \in S_C - \{n, n_1, \ldots, n_j\}$, and $\lambda'_C(n) = \square$. Since C' is smaller than C, we can apply the induction hypothesis. Let $u, z_{j+1}, \ldots, z_k, y$ and $v, t'_{j+1}, \ldots, t'_k, t'$ be states of A such that $C'\left[v, t'_{j+1}, \ldots, t'_k\right] \Longrightarrow t'$ and $u \in Y, z_{j+1} \in Z_{j+1}, \ldots, z_k \in Z_k, y \in Z$ and $u \preccurlyeq_{down} v, z_{j+1} \preccurlyeq_{down} t'_{j+1}, \ldots, z_k \preccurlyeq_{down} t'_k, y \preccurlyeq_{up} t'$.

By definition of $A_\simeq$, there are states $z \in Y, z_1 \in Z_1, \ldots, z_j \in Z_j$ such that $(z_1, \ldots, z_j) \xrightarrow{f} z$ for some f.

We now use Lemma 7.5 with premise $u \simeq z$ and $u \preccurlyeq_{down} v$. We thus find state w such that $z \preccurlyeq_{down} w$ and $v \preceq w$. Note that this implies $v \preccurlyeq_{up} w$.

By definition of a downward simulation, and premises $z \preccurlyeq_{down} w$ and $(z_1, \ldots, z_j) \xrightarrow{f} z$, we find states $t_1, \ldots, t_j$ with $z_1 \preccurlyeq_{down} t_1, \ldots, z_j \preccurlyeq_{down} t_j$ and $(t_1, \ldots, t_j) \xrightarrow{f} w$.

By definition of an upward simulation and premises $v \preccurlyeq_{up} w$ and $C'\left[v, t'_{j+1}, \ldots, t'_k\right] \Longrightarrow t'$, we find states $t, t_{j+1}, \ldots, t_k$ with $t' \preccurlyeq_{up} t$, $t'_{j+1} \preccurlyeq_{down} t_{j+1}, \ldots, t'_k \preccurlyeq_{down} t_k$ and $C'[w, t_{j+1}, \ldots, t_k] \Longrightarrow t$.

The claim thus holds. □

We are now ready to prove the soundness of merging with $\simeq$.

Theorem 7.1. *Let $A = (Q, \Sigma, I, \Delta, F)$ be a tree automaton. Consider two binary relations $\preccurlyeq_{down}$ and $\preceq$ on the states of A, satisfying Sufficient Conditions 7.1, and let $\simeq$ be their induced relation. Let $A_{\simeq} = (Q/\simeq, F/\simeq, \delta_{\simeq})$ be the automaton obtained by merging the states of A according to $\simeq$. Then, $\mathcal{L}(A_{\simeq}) = \mathcal{L}(A)$.*

Proof. Since $A_{\simeq}$ is a collapsed version of A, we trivially have $\mathcal{L}(A_{\simeq}) \supseteq \mathcal{L}(A)$.

Conversely, let T be a tree accepted by $A_{\simeq}$. We construct a context C by replacing all leaves in T by holes. We apply the construction of Lemma 7.7 to context C. We now have a run of A on C. Conditions 4 and 5 of Sufficient Conditions 7.1 ensure that this run is accepting. Condition 6 of Sufficient Conditions 7.1 ensures that we can extend the run on C to a run on T. □

Theorem 7.1 can be used to relate the languages of H and $D_{\simeq}$ (recall that $D_{\simeq}$ was defined in section 7.1). We are now ready to prove the soundness and the completeness of the procedure of section 7.1 (assuming a computable equivalence relation $\simeq$).

Theorem 7.2. *Consider a transducer $D = (Q, \Sigma, I, \Delta, F)$ and its associated history transducer $H = (Q_H, F_H, \delta_H)$. Consider two binary relations $\preccurlyeq_{down}$ and $\preceq$ on the states of H which satisfy the hypothesis of Theorem 7.1. Let $\simeq$ be their induced equivalence relation. If the procedure of section 7.1 terminates at step i, then the transducer $D_{\simeq}^{\leq i}$ accepts the same relation as $D_{\simeq}$.*

Proof. We can easily see that by construction, $D_{\simeq}^{\leq i}$ is a sub-automaton of $D_{\simeq}$.

Conversely, let (T_1, T_2) be a pair accepted by $D_{\simeq}$. We use Theorem 7.1, and let r be the corresponding run in H. Let $w_0, w_1, \ldots, w_n$ be the states in r. Let k be the length $k = |w_0| = |w_1| = \ldots = |w_n|$. Note that (T_1, T_2) is accepted by $D^{\leq k}$.

- If $k \leq i$, then by construction, states $[w_0], [w_1], \ldots, [w_n]$ are in $D_{\simeq}^{\leq i}$, and there is an accepting run in $D_{\simeq}^{\leq i}$ for the pair (T_1, T_2).
- If $k > i$, then we let T be such that (T_1, T) is recognized by $D^{\leq i}$ and (T, T_2) is recognized by $D^{\leq k-i}$. By the reasoning above, we know that (T_1, T) is recognized by $D_{\simeq}^{\leq i}$.
  Hence, we can write $(T_1, T_2) \in rel(D_{\simeq}^{\leq i} \circ D^{\leq k-i})$. Using ($k - i$ times) the termination condition $rel(D_{\simeq}^{\leq i} \circ D) \subseteq rel(D_{\simeq}^{\leq i})$, we get that (T_1, T_2) is recognized by $D_{\simeq}^{\leq i}$.

□

7.2.4 Sufficient Condition for Computability

The next step is to give conditions on the simulations which ensure that the induced equivalence relation is computable.

Definition 7.6 (Effective relation)**.** *A relation $\preceq$ is said to be effective if the image of a regular set w.r.t. $\preceq$ and w.r.t. $\preceq^{-1}$ is regular and computable.*

Effective relations induce an equivalence relation which is also computable.

Theorem 7.3. *Let $D = (Q,\Sigma,I,\Delta,F)$ be a transducer and $H = (Q_H,F_H,\delta_H)$ its associated history transducer. Let $\preceq_1$ and $\preceq_2$ be relations on the states of H that are both reflexive, transitive, effective and independent. Let $\simeq$ be their induced equivalence. Then for any state $w \in Q_H$, we can compute its equivalence class $[w]$ w.r.t. $\simeq$.*

Proof. The claim follows by definition of $\simeq$, and effectiveness [2] of $\preceq_1$ and $\preceq_2$. □

Using relations on the states of H that satisfy the premises of Theorem 7.3 naturally turns the procedure of section 7.1 into an *algorithm*. If the relations used also satisfy the premises of Theorem 7.1, then the on-the-fly algorithm of section 7.1 computes (when it terminates) the transitive closure of a tree transducer. The next step is to provide a concrete example of such relations. Because we are not able to compute the infinite representation of H, the relations will be directly computed from the powers of D provided by the on-the-fly algorithm.

7.3 Good Equivalence Relation

In this section, we provide concrete relations satisfying Theorem 7.1 and Theorem 7.3. We first introduce prefix- and suffix-copying states.

Definition 7.7 (Prefix-Copying State)**.** *Given a transducer $D = (Q,\Sigma,I,\Delta,F)$, and a state $q \in Q$, we say that q is a prefix-copying state if for any tree $T = (S,\lambda) \in \mathrm{pref}(q)$, then for any node $n \in S$, $\lambda(n) = (f,f)$ for some symbol $f \in \Sigma$.*

Definition 7.8 (Suffix-Copying State)**.** *Given a transducer $D = (Q,\Sigma,I,\Delta,F)$, and a state $q \in Q$, we say that q is a suffix-copying state if for any context $C = (S_C,\lambda_C) \in \mathrm{suff}(q)$, then for any node $n \in S_C$ with $\lambda_C(n) \neq \square$, we have $\lambda_C(n) = (f,f)$ for some symbol $f \in \Sigma$.*

We let Q_{pref} (resp. Q_{suff}) denote the set of prefix-copying states (resp. the set of suffix-copying states) of D and we assume that $Q_{pref} \cap Q_{suff} = \emptyset$. We let $Q_N = Q - [Q_{pref} \cup Q_{suff}]$.

We now define relations by the means of rewriting rules on the states of the history transducer.

[2] A state w of the history transducer is a word. The set $\{w\}$ is regular.

Definition 7.9 (Generated Relation). *Let $D = (Q, \Sigma, I, \Delta, F)$ be a tree transducer, and $H = (Q_H, F_H, \delta_H)$ its associated history transducer. Given a set $S \subseteq Q_H \times Q_H = Q^* \times Q^*$, we define the relation $\mapsto$* generated *by S to be the smallest reflexive and transitive relation such that $\mapsto$ contains S, and $\mapsto$ is a congruence w.r.t. concatenation (i.e. if $x \mapsto y$, then for any w_1, w_2, we have $w_1 \cdot x \cdot w_2 \mapsto w_1 \cdot y \cdot w_2$).*

Next, we find relations $\preceq$ and $\preccurlyeq_{down}$ on the states of H that satisfy the sufficient conditions for computability (Theorem 7.3) and conditions for exactness of abstraction (Theorem 7.1).

Definition 7.10 (Simulation Relations). *Let $D = (Q, \Sigma, I, \Delta, F)$ be a tree transducer, and $H = (Q_H, F_H, \delta_H)$ its associated history transducer. Let Q_{pref} (resp. Q_{suff}) denote the set of prefix-copying states (resp. the set of suffix-copying states) of D, assuming $Q_{pref} \cap Q_{suff} = \emptyset$. We let $Q_N = Q - [Q_{pref} \cup Q_{suff}]$.*

- *We define $\preccurlyeq_{down}$ to be the downward simulation generated by all pairs of the form $(q_{pref} \cdot q_{pref}, q_{pref})$ and $(q_{pref}, q_{pref} \cdot q_{pref})$, where $q_{pref} \in Q_{pref}$.*
- *Let $\preccurlyeq^1_{up}$ be the maximal upward simulation computed on $D \cup D^2$. We define $\preceq$ to be the relation generated by the maximal set $S \subseteq \preccurlyeq^1_{up}$ such that*
  - *$(q_{suff} \cdot q_{suff}, q_{suff}) \in S$ iff $(q_{suff}, q_{suff} \cdot q_{suff}) \in S$,*
  - *$(q \cdot q_{suff}, q) \in S$ iff $(q, q \cdot q_{suff}) \in S$,*
  - *$(q_{suff} \cdot q, q) \in S$ iff $(q, q_{suff} \cdot q) \in S$,*

  where $q_{suff} \in Q_{suff}$, and $q \in Q_N$.

Algorithms for computing the simulations needed for Definition 7.10 can be found in [ALdR05]. These algorithms are adapted from those provided by Henzinger *et al.* [HHK95] for the case of finite words.

Let us state that the simulations of Definition 7.10 satisfy the Sufficient Conditions 7.1, and hence satisfy the premises of Theorem 7.3 and Theorem 7.1.

Lemma 7.8. *Let $\preccurlyeq_{down}$ as defined in Definition 7.10. The following properties of $\preccurlyeq_{down}$ hold:*

1. *$\preccurlyeq_{down}$ is a downward simulation;*
2. *$\preccurlyeq_{down}$ is effective.*

Proof. 1. Let $x \cdot q_{pref} \cdot y$ be a state of H. Any transition rule leading to that state will be of the form:

$$\left(x^1 \cdot q^1_{pref} \cdot y^1, \ldots, x^n \cdot q^n_{pref} \cdot y^n\right) \xrightarrow{f}_H x \cdot q_{pref} \cdot y$$

Suppose $x \cdot q_{pref} \cdot y \preccurlyeq_{down} z$. Then by definition, we know that z is of the form $x \cdot q_{pref} \cdot q_{pref} \cdot y$. Observe that for each $i : 1 \leq i \leq n$, we have $x^i \cdot q^i_{pref} \cdot y^i \preccurlyeq_{down} x^i \cdot q^i_{pref} \cdot q^i_{pref} \cdot y^i$. We also have a rule:

$$\left(x^1 \cdot q^1_{pref} \cdot q^1_{pref} \cdot y^1, \ldots, x^n \cdot q^n_{pref} \cdot q^n_{pref} \cdot y^n\right) \xrightarrow{f}_H x \cdot q_{pref} \cdot q_{pref} \cdot y$$

Conversely, we consider the state $x \cdot q_{pref} \cdot q_{pref} \cdot y$ of H. We notice that since D is deterministic, it follows that a state of form $q^1_{pref} \cdot q^2_{pref}$ is not reachable in H unless $q^1_{pref} = q^2_{pref}$. We can thus ignore states in which $q^1_{pref} \neq q^2_{pref}$. Then, any transition rule leading to state $x \cdot q_{pref} \cdot q_{pref} \cdot y$ will be of the form:

$$\left(x^1 \cdot q^1_{pref} \cdot q^1_{pref} \cdot y^1, \ldots, x^n \cdot q^n_{pref} \cdot q^n_{pref} \cdot y^n\right) \xrightarrow{f}_H x \cdot q_{pref} \cdot q_{pref} \cdot y$$

Suppose $x \cdot q_{pref} \cdot q_{pref} \cdot y \preccurlyeq_{down} z$. Then we have z of the form $x \cdot q_{pref} \cdot y$. Observe that we also have for each i: $x^i \cdot q^i_{pref} \cdot q^i_{pref} \cdot y^i \preccurlyeq_{down} x^i \cdot q^i_{pref} \cdot y^i$. We also have a rule:

$$\left(x^1 \cdot q^1_{pref} \cdot y^1, \ldots, x^n \cdot q^n_{pref} \cdot y^n\right) \xrightarrow{f}_H x \cdot q_{pref} \cdot y$$

2. If we consider a regular set of states of H given by a word automaton, then its image w.r.t. $\preccurlyeq_{down}$ or $\preccurlyeq^{-1}_{down}$ can be expressed by adding edges to this automaton: for each transition $x \xrightarrow{q_{pref}} y$, add an edge $y \xrightarrow{q_{pref}} y$; similarly, for each consecutive edges $x \xrightarrow{q_{pref}} y$ and $y \xrightarrow{q_{pref}} z$, add an edge $x \xrightarrow{q_{pref}} z$.

□

Lemma 7.9. *Let $\preceq$ as defined in Definition 7.10. The following properties of $\preceq$ hold:*
1. *$\preceq$ is included in an upward simulation;*
2. *$\preceq$ is effective.*

Proof. 1. We know that $\preccurlyeq^1_{up}$ is an upward simulation. If we let S be the relation generated by $\preccurlyeq^1_{up}$, then S is also an upward simulation. Furthermore, we have $\preceq \subseteq S$.

2. If we consider a regular set of states of H given by a word automaton, then its image w.r.t. $\preceq$ or $\preceq^{-1}$ can be expressed by adding edges to this automaton:
   - for each transition $x \xrightarrow{q_{suff}} y$, add an edge $y \xrightarrow{q_{suff}} y$; similarly, for each consecutive edges $x \xrightarrow{q_{suff}} y$ and $y \xrightarrow{q_{suff}} z$, add an edge $x \xrightarrow{q_{suff}} z$.
   - if there is a pair $\left(q \cdot q_{suff}, q\right) \in S$ in Definition 7.10, then for each transition $x \xrightarrow{q} y$, we add an edge $y \xrightarrow{q_{suff}} y$; similarly, for two consecutive edges $x \xrightarrow{q} y$ and $y \xrightarrow{q_{suff}} z$, add an edge $x \xrightarrow{q} z$.
   - if there is a pair $\left(q_{suff} \cdot q, q\right) \in S$ in Definition 7.10, then for each transition $x \xrightarrow{q} y$, we add an edge $x \xrightarrow{q_{suff}} x$; similarly, for two consecutive edges $x \xrightarrow{q_{suff}} y$ and $y \xrightarrow{q} z$, add an edge $x \xrightarrow{q} z$.

□

We now state that $\preceq$ and $\preccurlyeq_{down}$ are independent.

Lemma 7.10. *Let $\preccurlyeq_{down}$ and $\preceq$ as defined in Definition 7.10. $\preceq$ and $\preccurlyeq_{down}$ are independent.*

Proof. Assume $x \preceq y$ and $x \preccurlyeq_{down} z$. Then $x = x_1 \cdot q_{pref} \cdot x_2$ and $z = x_1 \cdot z' \cdot x_2$, with $z' \in \{\varepsilon, q_{pref} \cdot q_{pref}\}$. Since the left-hand side of each pair generating $\preceq$ does not contain any prefix-copying state, we conclude that $y = y_1 \cdot q_{pref} \cdot y_2$, where either $x_1 \preceq y_1$ and $x_2 = y_2$, or $x_1 = y_1$ and $x_2 \preceq y_2$. In either case, we have $z = x_1 \cdot z' \cdot x_2 \preceq y_1 \cdot z' \cdot y_2$. Furthermore, we also have $y = y_1 \cdot q_{pref} \cdot y_2 \preccurlyeq_{down} y_1 \cdot z' \cdot y_2$.

We have shown independence of *single steps* of $\preceq$ and $\preccurlyeq_{down}$. This is sufficient for proving that independence also holds for the transitive closure w.r.t. concatenation. Hence the claim holds. □

The properties proved above are enough to ensure partial soundness (as stated in Lemma 7.7). We now state the remaining properties needed to comply with Theorem 7.1.

Lemma 7.11. *Let $\preccurlyeq_{down}$ and $\preceq$ as defined in Definition 7.10. The following holds:*

- *whenever $x \in F_H$ and $x \preccurlyeq_{up} y$, then $y \in F_H$;*
- *F_H is a union of equivalence classes w.r.t. $\simeq$;*
- *whenever $\xrightarrow{f} x$ and $x \preccurlyeq_{down} y$, then $\xrightarrow{f} y$;*

Proof. We observe that all states in F are either in Q_{suff} or in Q_N. Therefore, the first and second claim hold.

The third claim holds since $x \preccurlyeq_{down} y$ only involves prefix-copying states. For a prefix-copying state q_{pref}, an arity 0 rule will be of the form $\xrightarrow{(f,f)} q_{pref}$, which means that the claim holds. □

We conclude that $\preceq$ and $\preccurlyeq_{down}$ satisfy the hypothesis of Theorem 7.1 and Theorem 7.3 and can thus be used by the on-the-fly procedure presented in section 7.1.

Consider the relations $\preceq$ and $\preccurlyeq_{down}$ of Definition 7.10. They induce an equivalence relation $\simeq$ used to merge states of the history transducer H. To get an intuition of the effect of merging states according to $\simeq$, we look at states considered equivalent:

- For any prefix-copying state $q_{pref} \in Q$, and any words x, y, all states in the set $x \cdot q_{pref}^{+} \cdot y$ will be equivalent.
- For any suffix-copying state $q_{suff} \in Q$, and any words x, y, all states in the set $x \cdot q_{suff}^{+} \cdot y$ will be equivalent.
- For any states q_{suff}, q such that $(q \cdot q_{suff}, q) \in S$ according to Definition 7.10, and for any words x, y, all states in the set $x \cdot q \cdot q_{suff}^{*} \cdot y$ will be equivalent.
- For any states q_{suff}, q such that $(q_{suff} \cdot q, q) \in S$ according to Definition 7.10, and for any words x, y, all states in the set $x \cdot q_{suff}^{*} \cdot q \cdot y$ will be equivalent.

7.4 Computing Reachable Configurations

In this section we describe the modifications needed to compute $rel\,(D^{+})\,(\phi_I)$ without computing D^{+}. For checking safety properties, such a computation is sufficient (see [VW86]). Computing $rel\,(D^{+})\,(\phi_I)$ rather than D^{+} can be done by slightly modifying the definition of the history transducer associated with D^{+}.

Let $D = (Q, \Sigma, I, \Delta, F)$. Assume that we have constructed a tree automaton $A_{\phi_I} = \left(Q_{\phi_I}, F_{\phi_I}, \delta_{\phi_I}\right)$ for ϕ_I. Then, we define our new history transducer to be $H(\phi_I) = (Q_H, F_H, \delta_H)$, where

- $Q_H = Q_{\phi_I} \times Q^{+}$,
- $F_H = F_{\phi_I} \times F^{+}$,
- and δ_H contains all rules of the form

$$(w_1, \ldots, w_p) \xrightarrow{(f, f')} w$$

such that there is $k \geq 1$ where the following two conditions are satisfied

1. $|w_1| = \cdots = |w_p| = |w| = k+1$;
2. there are $f_1, f_2, \ldots, f_{k+1}$, with $f = f_1$, $f' = f_{k+1}$, and $(w_1(1) \ldots, w_p(1)) \xrightarrow{f_1} w(1) \in \delta_{\phi_I}$, and $(w_1(i+1) \ldots, w_p(i+1)) \xrightarrow{(f_i, f_{i+1})} w(i+1) \in \delta$, for each $i : 1 \leq i \leq k$.

Intuitively, we just add a composition with the automaton representing the set of initial configurations, so that we effectively compute $\left(\bigcup_{i=1}^{\infty} (T(\Sigma) \times A_{\phi_I}) \circ D^{i}\right)|_1$.

Computing $rel\,(D^{+})\,(\phi_I)$ is often less expensive than computing D^{+} because it only considers reachable sets of states. Moreover, as for the word case, there exist situations for which $rel\,(D^{+})\,(\phi_I)$ is regular while D^{+} is not. We have an example for which our technique can compute $rel\,(D^{+})\,(\phi_I)$ but cannot compute D^{+}.

7.5 Experimental Results

The techniques presented in this chapter have been applied on the case studies of subsection 6.2.3 using a prototype implementation that relies in part on the regular model checking tool (see [Nil02]). For each example, we compute the set of reachable states, as well as the transitive closure of the transition relation. In Table 7.1, we report the results. For each automaton, we give its size in terms of states (column labeled *st*), and of transitions (column labeled *tr*). The columns labeled t and m indicate respectively the time (in seconds) and the memory (in Mb) required for computing the result of the precedent column.The largest automaton encountered during this computation is indicated by the column labeled *max*. The computations were run on a 1.6Ghz laptop with 1G of memory.

Note that since the protocols are all parameterized with respect to the number of participants, verification of such protocols has to take into account all possible instances (number of participants, how these participants connect to each other, etc.). Details of the encoding of the protocols in the tree regular model checking framework can be found in subsection 6.2.3.

Protocol	R		ϕ_I		$R^+ \| R^+(\phi_I)$				
	st	tr	st	tr	st	tr	t	m	max
Simple Token	3	8	2	6	3|2	10|6	1.4|0.2	22|22	15|5
Two-Way Token	4	12	2	6	5|2	22|6	7.4|4.7	22|22	58|48
Percolate	4	13	3	6	6|5	58|43	38|36	22|22	52|63
Tree-arbiter	8	59	4	18	-|4	-|22	-|5100	-|600	-|1739
Leader Election	6	38	2	6	9|10	87|92	164|258	22|23	98|146

Table 7.1: *For each tree-protocol introduced in 6.2.3, we compute the reachable states and the transitive closure. We give the size of each automaton: number of states (*st*) and transitions (*tr*); the computations time (*t*) in seconds and memory (*m*) in MB. We also indicate the size of the largest encountered automaton (*max*).*

Observe that we are not able to compute the transitive closure of the transition relation of the tree-arbiter protocol (it is not known whether the closure is even regular). However, we are still able to compute transitive closure of individual actions for this protocol as well as the reachable set of states with the technique of section 7.4.

7.6 Conclusions

In this chapter, we have presented a technique for computing the transitive closure of a tree transducer. The technique is based on the definition and the computation of a good equivalence relation which is used to collapse the states of the transitive closure of the tree transducer. Our technique has been implemented and successfully tested on a number of protocols. The restriction to structure-preserving tree transducers might be seen as a weakness of this approach. However, structure-preserving tree transducers can model the relation of many interesting parametrized network protocols. One can investigate the case of non structure-preserving tree transducers. One possible solution would be to use *padding* to simulate a structure-preserving behavior. The technique of padding works by adding extra symbols to the alphabet to denote positions in the tree that are empty, and can be ignored. Using such symbols, transducer rules which change the structure of a tree can be rewritten as structure preserving rules. Finally, it would be interesting to extend our framework to check liveness properties on tree-like architecture systems as it is done for linear topologies in [AJN$^+$04, BLW04].

8. Conclusions

In this part, we have extended the framework of regular model checking to the case of parameterized systems with tree-like structures. A central problem in regular model checking is the computation of the transitive closure of the transition relation induced by the system at hand. First, we have introduced how to represent configurations and transitions, and how to verify safety and liveness properties for the case of linear parameterized systems (with finite components). Then, we have generalized the representations to the case of parameterized systems where the components are placed in tree structures. For this purpose, we have used tree automata and tree transducers, and did employ them to model several parameterized systems. We have presented a technique for computing the transitive closure of a tree transducer. The technique is based on the definition and the computation of a good equivalence relation which is used to collapse the states of the transitive closure of the tree transducer. Our technique has been implemented and successfully tested on a number of protocols.

The approach of regular model checking has the advantage of being uniform and fully automatic. Both safety and liveness properties can be verified in this approach. It would for instance be interesting to use the transitive closures computed here to check liveness properties on tree-like systems as it is done for linear topologies in [AJN$^+$04, BLW04]. The main problem with transducer-based techniques is that they are very heavy and usually rely on several layers of computationally expensive automata-theoretic constructions. In the next part, we propose a much more light-weight and efficient approach to regular model checking, and describe its application in the context of parameterized systems.

Part II:
Monotonic Over-Approximation for Safety

9. Introduction

The framework introduced in Part I, namely the framework of *regular model checking*, allows automatic and uniform verification of safety and liveness properties for infinite state systems in general, and parameterized systems in particular. The main problem with this transducer-based technique is that it is heavy and usually rely on several layers of computationally expensive automata-theoretic constructions; in many cases severely limiting its applicability. Other approaches, like abstraction techniques can be more light weight. In abstraction approaches, the transition system induced by the parameterized system is approximated into another transition system, denoted by *approximate transition system* (*approximate system* for short) in order to facilitate the verification. For instance, some abstraction techniques aim at producing a finite approximate transition system for which finite model checking techniques can be used for verification. However, some transition systems are inherently infinite, and finite approximations may omit information that is essential for the conclusion of the verification.

In this chapter, we present a method for model checking safety properties on *linear parameterized systems*. Typically, such a system consists of an arbitrary number of components organized in a linear array. Parameterized systems with unstructured architectures can be viewed as a special case of linear parameterized systems where the ordering of the components is not relevant. In this part, we use indices to reflect the relative positions of components in the linear array. More concretely, we say that the index i of a component is strictly smaller than the index j of a second component (write $i < j$) to mean that the former is to the left of the latter (we also say that the latter is to the right of the former) in the linear array. In a similar manner, we write $i \neq j$ and $i = j$ to respectively mean different and identical components. We will often refer to a component by its position in the array, and say process i instead of process at position i in the array. The task is to check correctness regardless of the number of components inside the system. Here, correctness is defined with respect to safety properties (introduced later in chapter 10).

Components (or processes) in the linear parameterized systems we consider are modeled as instantiations of the same automaton. The automaton operates on a finite number of states Q and "owns" a set of *local variables*. Depending on the types of variables manipulated by the components, we define three classes of linear parameterized systems. The first class is defined as the class where the local variables are *bounded*; these are the most studied in the lit-

erature. The second class is the one for which local variables may assume both *bounded* and *numerical* (i.e. natural) domains. This class strictly generalizes the first one. The third class we consider is the class of parameterized systems where individual components manipulate local arrays where they associate variables to each other component.

In order to simplify the presentation, we introduce the method in two steps. In the first step the method is defined for the case of systems where the automaton (instantiating the components) manipulates a finite set of local variables X ranging over some finite or numerical domains. The automaton changes state and alters the values of its local variables according to a set of transition rules T. This corresponds to the second class of linear parameterized systems. The second step generalizes the method to the case where the automaton owns and manipulates, in addition to the variables in X, arrays indexed by the other components. We give below a short description for both classes. For the sake of simplifying the presentation, and without loss of generality, we assume that all variable with finite domains are Boolean variables.

Systems with bounded and numerical variables

In this class, the local variables in X range over finite or numerical domains. The transitions T of the automaton are conditioned by the local state of the component taking the transition, values of its local variables, and by *global conditions*. A global condition is either *universally* or *existentially* quantified. An example of a universal condition is that "all components to the left of the component i should satisfy a property θ". Process i is allowed to perform the transition only in the case where each component with index $j : j < i$ satisfies θ. In an existential condition we require that *some* (rather than *all*) components satisfy θ. Most existing approaches to automatic verification of parameterized systems make the restriction that the domains of the variables in X are finite. However, there are many applications where the system behavior relies on unbounded data structures such as counters, priorities, local clocks, time-stamps, and process identifiers. The conditions appearing in the transitions are stated as propositional constraints on the finite variables, and as *gap-order constraints* on the numerical variables. Gap-order constraints [Rev93] are a logical formalism in which we can express simple relations on variables such as lower and upper bounds on the values of individual variables; together with equalities and gaps (minimal differences) between values of pairs of variables. We use this formalism to constrain both the *current-values* and the *next-values* of variables appearing in local and global conditions. An example of a universal condition is "variable x belonging to process i has a value which is strictly greater than the value of variable y in all other processes j inside the system, in the meantime variable y belonging to the same process i takes the maximum on the values of variables x belonging to each other processes j inside the system". In addition, components may communicate through broadcast, rendez-vous, and

shared variables. We also allow components to be dynamically created and deleted during the execution of the system.

Systems with array variables

This generalizes the class above. In addition to the variables in X, each component associates a set of variables R with each other component in the system. In other words, each process operates on a number of arrays (as many as there are variables in R) indexed by the other processes in the system. The variables in R can have finite or numerical domains. The transitions in T are also conditioned by the local state of the component, values of the local variables X; and by *existential* and *universal* conditions which check the local states and variables X of the other components together with the variables in R locally associated with each one of them. An example of an existential condition is "the variable x belonging to process i has a value equal to the value of variable y belonging to a process j to the left of i", and "the Boolean entry associated with the same process j in some local array belonging to i is set to $true$". An example of a universal condition is "the variable x belonging to process i has a value that is, for each other processes j in the system, larger than the value of the entry associated with j in some local array belonging to i, and this value is smaller than the value of the variable y belonging to j". This class can easily be used to model linear parameterized systems where the components communicate in a *non-distributed* fashion. *Non-distributed* Communication means that *global conditions* are evaluated by checking states and values of variables belonging to *other* components, *one other component* at a time.

We can uniformly apply our method on parameterized systems belonging to any of the above classes.

Method of monotonic abstraction

The proposed method applies in two steps. In the first step, an approximate transition system is uniformly and implicitly generated for the parameterized system at hand. The resulting approximate system has the same configurations as the original and is therefore infinite. The transition relation of the approximate transition system over-approximates the one of the original system. Verifying that a safety property holds in the approximate transition system entails the safety of the original system. The generated approximate transition system exhibits properties that permit the verification as a second step of the method. In the second step we apply, based on properties of the approximate system, the approach of [ACJT96] to define and mechanically check safety properties (introduced later). This results in a backwards analysis that starts from a set of bad configurations representing the violation of the safety property at hand.

The analysis is carried out until a fixpoint is reached. This analysis on the approximate transition system is not guaranteed to terminate in general. Termination is however guaranteed on approximate transition systems resulting from parameterized systems where each component manipulates bounded variables.

For such systems, termination of the approximate analysis relies on results from the theory of *well quasi-ordering* [Hig52]. The analysis is however undecidable for each of the considered classes of parameterized systems. The method presented in this part allowed the automatic verification of an important number of parameterized systems (some of them for the first time in a fully mechanized fashion).

We conclude this introduction by introducing some preliminary notations and the outline of this Part.

Preliminaries

We use $\mathcal{B}$ to denote the set $\{true, false\}$ of Boolean values; and use $\mathcal{N}$ to denote the set of natural numbers. For a natural number $n \in \mathcal{N}$, we write $\overline{n}$ to denote the set $\{1, \ldots, n\}$. We work with sets of variables. Unless otherwise specified, such a set A is often partitioned into two subsets: *Boolean* variables $A^{\mathcal{B}}$ which range over $\mathcal{B}$, and *numerical* variables $A^{\mathcal{N}}$ which range over $\mathcal{N}$. We let $\mathbb{B}(A^{\mathcal{B}})$ denote the set of formulas which have members of $A^{\mathcal{B}}$ as atomic formulas, and which are closed under the Boolean connectives $\neg, \wedge, \vee$. We will also use a subset of *gap formulas* [Rev93], to constrain the numerical variables. More precisely, we let $\mathbb{G}(A^{\mathcal{N}})$ be the set of formulas which are either of the form $x = y$ or of the form $x \sim_k y$ where $\sim \in \{<, \leq\}$, $x, y \in A^{\mathcal{N}}$, and $k \in \mathcal{N}$. Here $x \sim_k y$ stands for $x + k \sim y$. We use $\mathbb{F}(A)$ to denote the set of formulas which has members of $\mathbb{B}(A^{\mathcal{B}})$ and of $\mathbb{G}(A^{\mathcal{N}})$ as atomic formulas, and which is closed under the Boolean connectives $\wedge, \vee$. For instance, if $A^{\mathcal{B}} = \{a, b\}$ and $A^{\mathcal{N}} = \{x, y\}$ then $\theta = (a \Rightarrow b) \wedge (x <_3 y)$ is in $\mathbb{F}(A)$.

Sometimes, we write a formula as $\theta(y_1, \ldots, y_k)$ where $y_1, \ldots, y_k$ are the variables which may occur in θ (we can for instance write $\theta(x, y, a, b)$ to refer to the above formula). For sets of variables $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_n\}$, we sometimes abuse notations and write $\theta(X, Y)$ to mean $\theta(x_1, \ldots, x_n, y_1, \ldots, y_n)$. We perform substitutions on formulas in $\mathbb{F}(A)$. A *substitution* is a set $\{x_1 \leftarrow e_1, \ldots, x_n \leftarrow e_n\}$ of pairs where x_i are variables, and e_i are all constants or all variables. For each $i : 1 \leq i \leq n$, e_i is of the same type as x_i (i.e. if e_i is a constant then it should be in the domain of variable x_i, if it is a variable then it should have the same domain). Here, we assume that all the variables are distinct, i.e., $x_i \neq x_j$ if $i \neq j$. For a formula θ and a substitution S, we use $\theta[S]$ to denote the formula we get from θ by simultaneously replacing all occurrences of the variables $x_1, \ldots, x_n$ by $e_1, \ldots, e_n$ respectively. If $e_1, \ldots, e_n$ are constants, all variables appearing in θ are to be replaced. Observe that in such a case, the formula $\theta[S]$ evaluates either to $true$ or to $false$. Sometimes, we may write $\theta[S_1][S_2] \cdots [S_m]$ instead of $\theta[S_1 \cup S_2 \cup \cdots \cup S_m]$. As an example, if $\theta = (x_1 < x_2) \wedge (x_3 <_2 x_4)$ then $\theta[x_1 \leftarrow y_2, x_4 \leftarrow x_3][x_2 \leftarrow x_3] = (y_2 < x_3) \wedge (x_3 <_2 x_3)$.

Outline

In chapter 10, we introduce a general scheme that we rely on to define safety properties and automatically check them. We instantiate this scheme, in chapters 11 and 12 respectively, on the second and the third class of linear parameterized systems (introduced above). In chapter 13, we report on the encouraging results obtained using our prototype implementation. Some of the results represent (as far as we know) the first *fully mechanized* check of the considered safety properties for such systems. We conclude in chapter 14.

10. Reachability for Monotonic Systems

In this chapter, we recall the approach [ACJT96] for defining safety properties, and checking their correctness on general *monotonic* transition systems. We state some requirements on the considered transition systems in order to turn the approach into a semi-algorithm, and possibly conclude termination.

Assume in the remaining of the current chapter a parameterized system $\mathcal{P}$, where each component may take a number of transitions $\{t | \, t \in T\}$. The parameterized system induces an infinite transition system $(C, \longrightarrow)$ where C is the infinite set of configurations and the transition relation $\longrightarrow \subseteq C \times C$ is the union $\cup_{t \in T} \xrightarrow{t}$ where $c \xrightarrow{t} c'$ if some component(s) in configuration c can take transition t to c'. For a configuration c and a set S of configurations, write $c \longrightarrow S$ to denote the existence of a transition t and a configuration $c' \in S$ such that $c \xrightarrow{t} S$. This naturally extends to $S \longrightarrow S'$, where S,S' are two sets of configurations. Write $\xrightarrow{*}$ to mean the transitive reflexive closure of $\longrightarrow$. Denote by I the set of initial configurations of the parameterized system. These typically correspond to configurations where each component is at some initial state. Let $\preceq$ be a decidable *quasi-order* on C[1].

In the following we start by defining, using $\preceq$, the safety properties we are interested in checking. We explain why verifying safety properties on an approximate transition system $(C, \rightsquigarrow)$ where $\rightsquigarrow$ includes $\longrightarrow$ is sufficient to conclude that the original system also does satisfy the properties. Then, we introduce *monotonicity* w.r.t $\preceq$, and explain its role in the backwards analysis. Finally, we explicit requirements on $(C, \rightsquigarrow)$ in order to turn the analysis into an effective semi-algorithm, and conclude by giving sufficient conditions for termination.

Safety Properties and Coverability

Safety properties can be formalized [VW86] as regular sets of bad sequences of transitions the system should not engage in. The verification of these properties are translated in the following manner (see for instance [EFM99]) into instances of the *coverability problem* w.r.t $\preceq$. Let Σ_{bad} be a finite set of symbols[2]. Let $bad = (Q_{bad}, \Sigma_{bad}, I_{bad}, \Delta_{bad}, F_{bad})$ be an automaton accepting the bad sequences defined as a regular language $\mathcal{L}(bad)$ over Σ_{bad}. We write $\mathcal{L}(\mathcal{P}, I)$ to

[1] i.e., a decidable reflexive and transitive binary relation on elements of C.

[2] for example the set $\{t | \, t \in T\}$ where T contains labels of the transitions a component may take.

denote all sequences of transitions the parameterized system can execute starting from any of the initial configurations in I. Define the cross product of $\mathcal{P}$ and *bad* to be the subset of $(C \times Q_B) \times \Sigma_B \times (C \times Q_B)$ where $(c,q) \xrightarrow{t} (c',q')$ iff $c \xrightarrow{t} c'$ and (q,t,q') is in Δ_{bad}. The system $\mathcal{P}$ is said to be **safe** if and only if the intersection $\mathcal{L}(\mathcal{P},I) \cap \mathcal{L}(bad)$ is empty.

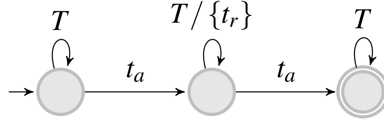


Figure 10.1: If $t_a, t_r \in T$ are the only transitions allowing components to access or release the critical section, then the automaton above exactly captures all the sequences violating the property of mutual exclusion.

Example 10.1. Assume a parameterized system where a component accesses its critical section via transition t_a and releases it with t_r. Assume other transitions in T do not access or release the critical section. The regular set defining the violation of the mutual exclusion property is captured by the automaton depicted in figure 10.1.

The *upward closure* (denoted $(c)\uparrow$) of a configuration $c \in C$ is the set $\{c' \mid c' \in C \text{ with } c \preceq c'\}$. A set D is *upward closed* if $c \in D$ and $c \preceq c'$ implies $c' \in D$. We define as follows the *coverability problem* with respect to $\preceq$:

PAR-COV
Instance • A transition system $(C, \longrightarrow)$ and a set of initial configurations I. • An upward closed set F of configurations. **Question** $I \xrightarrow{*} F$?

Safety properties can be formulated as an instance of the coverability problem in the following manner. Given an automaton *bad* characterizing the regular set of bad sequences, we extend the quasi-order $\preceq$ to $Q_{bad} \times C$. We write $(q,c) \preceq (q',c')$ iff $q = q'$ and $c \preceq c'$. The considered parameterized system is safe if and only if the set $\{(q,c) \mid q \in F_{bad} \text{ and } c \in C\}$ (which is upward closed) is not reachable from any of the elements belonging to the set $\{(q,c) \mid q \in I_{bad} \text{ and } c \in I\}$.

Remark 10.1. Observe that an upward closed set of configurations is coverable[3] in $(C, \longrightarrow)$ only if it is coverable in each over-approximation $(C, \rightsquigarrow)$[4]. In other words, it is sufficient to show that an over-approximation $(C, \rightsquigarrow)$ is safe in order to conclude the safety of the original transition system $(C, \longrightarrow)$.

[3] i.e., the answer to the corresponding coverability problem is positive

[4] In this part, an over-approximation of $(C, \longrightarrow)$ is any transition system $(C, \rightsquigarrow)$ with a set of configurations C and a transition relation $\rightsquigarrow$ that are respectively identical and larger than the set of configurations and the transition relation in $(C, \longrightarrow)$.

Backwards analysis and monotonicity

We solve the coverability problem by performing backwards reachability analysis starting from a set F that is upward closed w.r.t $\preceq$. The set F corresponds to the set characterizing the violation of the safety property above. In this analysis we iteratively compute the set of configurations from which we can reach (by application of $\longrightarrow$) a configuration in F. The parameterized system is declared to be safe if (i) we reach a fixpoint, and (ii) the set of configurations computed during the analysis does not intersect the initial set of configurations.

During the analysis, we work with constraints defined over C. A *constraint* ϕ characterizes a set of configurations that we denote by $[\![\phi]\!]$. Constraints are defined to represent upward closed sets of configurations. Observe that the upward closure of a set can denote, even for the case of singletons, an infinite set of configurations. Constraints are therefore symbolic representations for potentially infinite sets of configurations. For a finite set Φ of constraints, we let $[\![\Phi]\!] = \bigcup_{\phi \in \Phi} [\![\phi]\!]$. We define an *entailment relation* $\sqsubseteq$ on constraints, where $\phi_1 \sqsubseteq \phi_2$ iff $[\![\phi_2]\!] \subseteq [\![\phi_1]\!]$. For sets Φ_1, Φ_2 of constraints, abusing notation, we let $\Phi_1 \sqsubseteq \Phi_2$ denote that for each $\phi_2 \in \Phi_2$ there is a $\phi_1 \in \Phi_1$ with $\phi_1 \sqsubseteq \phi_2$. Notice that $\Phi_1 \sqsubseteq \Phi_2$ implies that $[\![\Phi_2]\!] \subseteq [\![\Phi_1]\!]$ (the converse is not true in general).

In the analysis we repeatedly perform the operation consisting in computing (for a constraint ϕ) the set $\{c \mid \exists c' \in [\![\phi]\!] \,.\, c \longrightarrow c'\}$. This set corresponds to the *predecessor* configurations, i.e. the configurations from which we can reach a configuration in $[\![\phi]\!]$ by the application of a single transition rule in $\longrightarrow$. This set needs to be upward closed in order to be characterized by a number of constraints on which we continue the analysis. A sufficient condition for a general transition system $(D, \Longrightarrow)$ in order to preserve upward closedness (w.r.t a quasi-ordering $\leq$ on D) under predecessor computations is the one of *monotonicity*. Intuitively, the transition system $(D, \Longrightarrow)$ is monotonic if from larger configurations we can obtain (by a single transition) configurations that are larger than those obtained by smaller configurations. More formally, $(D, \Longrightarrow)$ is monotonic if for all configurations $d_1, d_2, d_3 \in D$ the following holds. If $d_1 \leq d_2$ and $d_1 \Longrightarrow d_3$, then there exists a configuration $d_4 \in D$ such that $d_3 \leq d_4$ and $d_2 \Longrightarrow d_4$.

Lemma 10.1. *Let $(D, \Longrightarrow)$ be a transition system that is monotonic w.r.t to a quasi-order $\leq$ on D. The set $\{c \mid \exists c' \in U \,.\, c \Longrightarrow c'\}$ is upward closed if the set of configurations U is also upward closed.*

Proof. follows directly from the definition of monotonicity. □

In general, transition systems (induced by the classes of parameterized systems we look at in this work) are not monotonic. For each of the considered classes, we propose an over-approximation $(C, \rightsquigarrow)$ of $(C, \longrightarrow)$ that is monotonic with respect to the ordering $\preceq$ used in the definition of the safety properties. We perform the backwards analysis on the monotonic

over-approximations $(C, \rightsquigarrow)$. If the approximated transition system $(C, \rightsquigarrow)$ is safe, we conclude (remark 10.1) that the original transition system is also safe.

Assume a monotonic transition system $(C, \rightsquigarrow)$. Suppose we can characterize the set of predecessor configurations as a finite set $Pre(\phi)$ of constraints (i.e. $[\![Pre(\phi)]\!] = \{c \mid \exists c' \in [\![\phi]\!] \,.\, c \rightsquigarrow c'\}$). For a set Φ of constraints, we let $Pre(\Phi) = \bigcup_{\phi \in \Phi} Pre(\phi)$. In order to solve the coverability problem, the backwards reachability analysis starts from a finite set Φ_F of constraints[5], and checks whether $I \overset{*}{\rightsquigarrow} [\![\Phi_F]\!]$. In the analysis, we generate a sequence $\Phi_0 \sqsupseteq \Phi_1 \sqsupseteq \Phi_2 \sqsupseteq \cdots$ of finite sets of constraints such that $\Phi_0 = \Phi_F$, and $\Phi_{j+1} = \Phi_j \cup Pre(\Phi_j)$. Since $[\![\Phi_0]\!] \subseteq [\![\Phi_1]\!] \subseteq [\![\Phi_2]\!] \subseteq \cdots$, the procedure terminates when we reach a point j where $\Phi_j \sqsubseteq \Phi_{j+1}$. Notice that the termination condition implies that $[\![\Phi_j]\!] = (\bigcup_{0 \le i \le j} [\![\Phi_i]\!])$. Consequently, Φ_j characterizes the set of all predecessors of $[\![\Phi_F]\!]$. This means that $I \overset{*}{\rightsquigarrow} [\![\Phi_F]\!]$ iff $(I \bigcap [\![\Phi_j]\!]) \neq \emptyset$.

Required properties, termination

Observe that, in order to implement the scheme (i.e., transform it into an algorithm), we need to be able to:

- compute $Pre(\phi)$ for a given constraint ϕ as a finite set of constraints; and
- check for entailment $\phi \sqsubseteq \phi'$ between constraints ϕ and ϕ'; and
- check for the emptiness of $I \bigcap [\![\phi]\!]$ for the set of initial configurations I and a given constraint ϕ.

A constraint system satisfying these three conditions is said to be *effective*. Moreover, in [ACJT96], it is shown that termination is guaranteed in case the constraint system is *well quasi-ordered (WQO)* with respect to $\sqsubseteq$, i.e., if for each infinite sequence $\phi_0, \phi_1, \phi_2, \ldots$ of constraints, there are $i < j$ with $\phi_i \sqsubseteq \phi_j$.

In the following chapters, we define (for both the second and the third classes of parameterized systems introduced above) a quasi-order on the configurations of the induced transition system. We explain how to derive an approximate transition system that is monotonic with respect to the quasi-order. We also show how to perform each of the required operations above, and discuss termination of the approximate analysis.

[5] in all the applications we consider, the set Φ_F is easily deduced from the safety property to be checked.

11. Systems with Boolean and Numerical Variables

We introduce a basic model for *linear parameterized systems* with *Boolean* and *numerical* variables (i.e., second class in the introduction 9). The basic model will be enriched by additional features in section 11.5. As introduced earlier, we focus on parameterized systems where each process is modeled as an extended finite-state automaton operating on local variables which range over the Booleans and the natural numbers. The transitions of the automaton are conditioned by the values of the local variables and by *global* conditions in which the process checks, for instance, the local states and variables of all the other processes inside the system. A transition may change the value of any local variable inside the process (possibly deriving the new values from those of the other processes). A parameterized system induces an infinite family of (potentially infinite-state) systems, namely one for each size n. The aim is to verify correctness of the systems for the whole family (regardless of the number n of processes inside the system).

Formally, a *parameterized system* $\mathcal{P}$ is a triple (Q, X, T), where Q is a finite set of *local states*, X is a finite set of *local variables* partitioned into $X^{\mathcal{B}}$ (ranging over $\mathcal{B}$) and $X^{\mathcal{N}}$ (ranging over $\mathcal{N}$), and T is a finite set of *transition rules*. A transition rule t is of the form depicted in figure 11.1 with $\underline{q}, \underline{q}' \in Q$ and θ

$$t : \left[\ \underline{q} \rightarrow \underline{q}' \ \rhd \ \theta \ \right]$$

Figure 11.1: A transition rule in a parameterized system: a component takes transition t and moves from the *original* state (origin of the arrow) to the *target* state (target of the arrow). In the meantime, the (global) condition θ constrains the *current-values* and *next-values* associated with the involved variables.

is either a *local* or a *global condition*. Intuitively, the process which takes the transition changes its local state from $\underline{q}$ to $\underline{q}'$. In the meantime, the values of the local variables of the process are updated according to θ. Below, we describe how we define local and global conditions.

To simplify the definitions, we sometimes regard members of the set Q as Boolean variables. We define the set of variables $Y = X \cup Q$. Intuitively, the value of the Boolean variable $q \in Q$ is *true* for a particular process if and only if the process is in local state q. We also let $Y_1, Y_2 \ldots$ and $Y_1', Y_2', \ldots$ be distinct copies of the set Y, with $x_i \in Y_i$ and $x_i' \in Y_i'$ if $x \in Y$. Furthermore, we introduce

the set X^{next} which contains the *next-value* versions of the variables in X. A variable $x^{next} \in X^{next}$ represents the next value of $x \in X$.

In the transition t of figure 11.1 the current process changes state (while the other ones remain passive) from $\underline{q}$ to $\underline{q}'$. To capture this change of state, we define the formula $\eta_{\underline{q},\underline{q}'}(q, q^{next})$ as:

$$(q = \underline{q}) \wedge \left(\bigwedge_{p \in Q/\{\underline{q}\}} (p \neq \underline{q}) \right) \wedge (q^{next} = \underline{q}') \wedge \left(\bigwedge_{p \in Q/\{\underline{q}\}} (p^{next} \neq \underline{q}') \right)$$

Intuitively, the first two conjuncts encode the fact that the state of the process before performing the transition is equal to $\underline{q}$, while the last two conjuncts encode the information that the state after performing the transition is equal to $\underline{q}'$.

A *local condition* is a formula in $\mathbb{F}(X \cup X^{next})$. The formula specifies how local variables of the current process are updated with respect to their current values.

Global conditions check the values of local variables of the current process, together with the local states and the values of local variables of other processes. Consequently, we need to distinguish between a local variable, say x, of the process which is about to perform a transition, and the same local variable x of the other processes inside the system. We do that by introducing, for each $x \in Y$, a new variable $o{\cdot}x$. We define the sets $o{\cdot}Y = \{o{\cdot}x | x \in Y\}$ and $o{\cdot}Y^{next} = \{o{\cdot}x^{next} | x \in Y\}$. A *global condition* is of the form $\Box\theta_1$ where $\Box$ is a *universal* or an *existential* quantifier, and θ_1 is a formula in $\mathbb{F}(X \cup o{\cdot}Y \cup X^{next})$. A universal quantifier is in $\{\forall_{LR}, \forall_L, \forall_R\}$. An existential quantifier is one of $\{\exists_L, \exists_R, \exists_{LR}\}$. The subscripts L, R, and LR stand for *Left*, *Right*, and *Left-Right* respectively. A global condition checks the local variables of the process which is about to make the transition (through X), and the local states and variables of the other processes (depending on the quantifier) through $o{\cdot}Y$. A global condition also specifies how the local variables of the process in transition are updated (through X^{next}). Notice that the new values are defined in terms of the current values of variables and local states of all the processes inside the system. A global condition is said to be *universal* or *existential* depending on the type of the quantifier appearing in it. As an example, the following universal formula (assume $X = \{a, x\}$)

$$\forall_L \left(a \wedge (o{\cdot}x < x^{next}) \wedge o{\cdot}q_1 \wedge a^{next} = a \right)$$

states that the transition may be performed only if variable a of the current process has the value *true*, and all the other processes to the left of the process in transition are in local state q_1. When the transition is performed, variable x of the current process is assigned a value which is strictly larger than the value of variables x belonging to all the other processes to the left of the process in transition.

Outline

The next section defines a basic model for parameterized systems belonging to the second class introduced at the beginning of this Part II. In the remaining of the current chapter, this model will be referred to as *the basic model*. Section 11 defines the *original* transition system induced by a parameterized system. Section 11.2 fixes an ordering on configurations. This ordering characterizes safety properties as introduced in chapter 10. Section 11.3 proposes an over-approximated transition system that is monotonic with respect to the defined ordering. We show in section 11.4 that the approximate transition system satisfies the requirements described in chapter 10, and identify a class for which termination of the approximate analysis is guaranteed. Finally, we consider in section 11.5 a set of extensions to the basic model. These include *binary communication*, *broadcast communication* and *shared variables*. We conclude the chapter in section 11.6.

11.1 Transition System

Each parameterized system induces two transition systems: an *original transition system* and an *approximate transition system*. We describe later (section 11.3) the approximate transition system. For now we introduce the original transition system.

The original transition system $\mathcal{T}(\mathcal{P}) = (C, \longrightarrow)$ induced by a parameterized system $\mathcal{P} = (Q, X, T)$ is described as follows. A configuration is defined by the local states and the values of the local variables of each process in the system. Formally, a *local variable state* v is a mapping from X to $\mathcal{B} \cup \mathcal{N}$ which respects the types of the variables. A *process state* u is a pair (q, v) where $q \in Q$ and v is a local variable state. As mentioned earlier, we may regard members of Q as Boolean variables. Thus, we can view a process state (q, v) as a mapping $u : Y \mapsto \mathcal{B} \cup \mathcal{N}$, where $u(x) = v(x)$ for each $x \in X$, $u(q) = \mathit{true}$, and $u(q') = \mathit{false}$ for each $q' \in Q / \{q\}$. The process state thus agrees with v on the values of local variables, and maps all elements of Q, except q, to *false*. A *configuration* is a sequence $u_1 \cdots u_n$ of process states. Intuitively, such a configuration corresponds to an instance of the system with n processes. Each pair $u_i = (q_i, v_i)$ gives the local state and the values of local variables belonging to process i.

We define the transition relation $\longrightarrow$ on the set of configurations as follows. We start by describing the semantics of local conditions. Recall that a local condition corresponds to one process changing state independently of the states and local variables belonging to the other processes. Therefore, the semantics is defined in terms of two local variable states v, v' respectively corresponding to the current and next values of the local variables of the process; and a formula $\theta \in \mathbb{F}(X \cup X^{next})$ (representing the local condition). We write $(v, v') \models \theta$ to denote the validity of the formula $\theta[\rho]$ where the substitution ρ

is defined by $\{x \leftarrow v(x) | \, x \in X\} \cup \{x^{next} \leftarrow v'(x) | \, x \in X\}$. In other words, we check the formula we get by respectively replacing the current and next-value variables in θ by their values as defined by v and v'. The formula is evaluated using the standard interpretations of the Boolean connectives and the arithmetical relations $<, \leq, =$. For process states $u = (q, v)$ and $u' = (q', v')$, we use $(u, u') \models \theta$ to denote that $(v, v') \models \theta$.

Next, we describe the semantics of global conditions. The definition is given in terms of two local variable states v and v', a process state u_j, and a formula θ in the set $\mathbb{F}(X \cup o{\cdot}Y \cup X^{next})$ (representing a global condition). The roles of v and v' are the same as for local conditions. We recall that a global condition also checks states and local variables of all (or some) of the other processes. Here, u_j represents the local state and variables of one such process. We write $(v, v', u_j) \models \theta$ to denote the validity of the formula $\theta[\rho]$ where the substitution ρ is defined as the union of $\{x \leftarrow v(x) | \, x \in X\}$, $\{x^{next} \leftarrow v'(x) | \, x \in X\}$, and $\{o{\cdot}x \leftarrow u_j(x) | \, x \in Y\}$. The substitutions on v and v' are analogous to the case of local conditions. In addition, we replace the local states and variables of the other process by their values as defined by v_j. The relation $(u, u', u_j) \models \theta$ is interpreted in a similar manner to the case of local conditions.

Now, we are ready to define $\longrightarrow$. Let t be a transition rule similar to the one depicted in figure 11.1. Consider two configurations $c = u_1 \cdots u_n$ and $c' = u'_1 \cdots u'_n$ with $u_k = (q_k, v_k)$ and $u'_k = (q'_k, v'_k)$ for each $k : 1 \leq k \leq n$. We denote by $c \xrightarrow{t} c'$ the fact that for some $i : 1 \leq i \leq n$ it is the case that $u_k = u'_k$ for each $k : 1 \leq k \neq i \leq n$, that $q_i = q$ and $q'_i = q'$, and that one of the following three conditions is satisfied:

1. θ is a local condition and $(u_i, u'_i) \models \theta$.
2. $\theta = \Box\theta_1$ is a global condition where $\Box$ is a **universal** quantifier and **both** of the following hold:
   - $(u_i, u'_i, u_j) \models \theta_1$ for **each** $j : 1 \leq j < i$ if $\Box \in \{\forall_L, \forall_{LR}\}$, **and**
   - $(u_i, u'_i, u_j) \models \theta_1$ for **each** $j : i < j \leq n$ if $\Box \in \{\forall_{LR}, \forall_R\}$;
3. $\theta = \Box\theta_1$ is a global condition where $\Box$ is an **existential** quantifier and **at least one** of the followings hold:
   - $(u, u', u_j) \models \theta_1$ for **some** (witness) $j : 1 \leq j < i$ if $\Box \in \{\exists_L, \exists_{LR}\}$, **or**
   - $(u, u', u_j) \models \theta_1$ for **some** (witness) $j : i < j \leq n$ if $\Box \in \{\exists_{LR}, \exists_R\}$.

We use $c \longrightarrow c'$ to denote that $c \xrightarrow{t} c'$ for some $t \in T$.

Given a parameterized system $\mathcal{P} = (Q, X, T)$, we assume that prior to starting the execution of the system, each process is in an (identical) *initial* process state $u_{init} = (q_{init}, v_{init})$. The set I of initial configurations corresponds to configurations of the form $u_{init} \cdots u_{init}$ (all processes are in their initial states). This set is infinite.

11.2 Ordering on Configurations

We introduce an ordering on configurations. This ordering is used for the instantiation of the scheme in chapter 10. We define the ordering on configurations as follows. Consider two configurations $c = u_1 \cdots u_m$ and $c' = u'_1 \cdots u'_n$, where $u_i = (q_i, v_i)$ for each $i : 1 \leq i \leq m$, and $u'_i = (q'_i, v'_i)$ for each $i : 1 \leq i \leq n$. We write $c \preceq_h c'$ to denote that there exists a strictly monotonic injection[1] $h : \overline{m} \to \overline{n}$ such that the following four conditions are satisfied for each $i, j : 1 \leq i, j \leq m$:

1. $q_i = q'_{h(i)}$.
2. $v_i(x)$ iff $v'_{h(i)}(x)$ for each $x \in X^{\mathcal{B}}$.
3. $v_i(x) = v_j(y)$ iff $v'_{h(i)}(x) = v'_{h(j)}(y)$, for each $x, y \in X^{\mathcal{N}}$.
4. for each $\sim \in \{<, \leq\}$, $k \in \mathcal{N}$, and $x, y \in X^{\mathcal{N}}$, it is the case that $v_i(x) \sim_k v_j(y)$ implies that there exists a natural l larger than k with $v'_{h(i)}(x) \sim_l v'_{h(j)}(y)$.

In other words, there is a corresponding (from left to right) process in c' for each process in c. The local states and the values of the Boolean variables coincide in the corresponding processes (Conditions 1 and 2). Regarding the numerical variables, the ordering preserves equality (Condition 3), while gaps between variables in c' are at least as large as the gaps between the corresponding variables in c (Condition 4). We use $c \preceq c'$ to denote that $c \preceq_h c'$ for some monotonic injection $h : \overline{m} \to \overline{n}$.

We state in the following some consequences resulting from the definition of the above ordering.

Lemma 11.1. *Consider two configurations $c = u_1 \cdots u_m$ and $c' = u'_1 \cdots u'_n$ such that $c \preceq_h c'$. Define the substitution ρ to be the set $\{x_i \leftarrow u_i(x) |\, x \in Y \text{ and } i : 1 \leq i \leq m\}$, and the substitution ρ_h to be the set $\left\{x_i \leftarrow u'_{h(i)}(x) |\, x \in Y \text{ and } i : 1 \leq i \leq m\right\}$. The following three conditions hold:*

1. *For each $\Gamma^{\mathcal{B}} \in \mathbb{B}\left(Y_1^{\mathcal{B}} \ldots Y_m^{\mathcal{B}}\right)$, it is the case that $\Gamma^{\mathcal{B}}[\rho] \iff \Gamma^{\mathcal{B}}[\rho_h]$.*
2. *For each $\Gamma^{\mathcal{N}} \in \mathbb{N}\left(Y_1^{\mathcal{N}} \ldots Y_m^{\mathcal{N}}\right)$, it is the case that $\Gamma^{\mathcal{N}}[\rho] \Rightarrow \Gamma^{\mathcal{N}}[\rho_h]$.*
3. *For each $\Gamma \in \mathbb{F}(Y_1 \ldots Y_m)$, it is the case that $\Gamma[\rho] \Rightarrow \Gamma[\rho_h]$.*

Proof. Consequence 1 follows from the fact that $c \preceq_h c'$ implies, for each $i : 1 \leq i \leq m$, that u_i and $u'_{h(i)}$ are required to map the Boolean variables appearing in $\Gamma^{\mathcal{B}}$ to the same values (Conditions 1 and 2 in the definition of $\preceq_h$). Consequence 2 follows from the fact that $c \preceq_h c'$ implies gaps and equalities between any numerical variables appearing in $\Gamma^{\mathcal{N}}$ are maintained or augmented (Conditions 3 and 4 in the definition of $\preceq_h$). Consequence 3 follows from consequences 1 and 2 and the fact that formulas in $\mathbb{F}(Y_1 \ldots Y_m)$ are by definition obtained as the closure, under $\wedge, \vee$, of formulas in $\mathbb{B}\left(Y_1^{\mathcal{B}} \ldots Y_m^{\mathcal{B}}\right)$ and $\mathbb{N}\left(Y_1^{\mathcal{B}} \ldots Y_m^{\mathcal{B}}\right)$. □

[1] $h : \overline{m} \to \overline{n}$ is a strictly monotonic injection if $i < j \Rightarrow h(i) < h(j)$ for each $i, j : 1 \leq i, j \leq m$

Intuitively, the above lemma states that it is always possible to find in larger configurations, sub-sequences that satisfy all formulas satisfied by smaller configurations. In the following lemma, we make use of this lemma 11.1, and of a result proved in [Rev93] ensuring that the set of gap-formulas is closed under the elimination (i.e. projection) of variables.

Lemma 11.2. *Consider configurations* $\underline{c} = \underline{u}_1 \cdots \underline{u}_m$*,* $\underline{c}' = \underline{u}'_1 \cdots \underline{u}'_{m'}$ *and* $c = u_1 \cdots u_n$*. Define* $\underline{\rho}$ *and* $\underline{\rho}'$*, respectively as* $\{x_i \leftarrow \underline{u}_i(x) \mid x \in Y \text{ and } i : 1 \leq i \leq m\}$*, and* $\{x'_i \leftarrow \underline{u}'_i(x) \mid x \in Y \text{ and } i : 1 \leq i \leq m'\}$*. Let* $\underline{\Gamma}$ *be a formula in* $\mathbb{F}\left(Y_1, \dots Y_m, Y'_1, \dots, Y'_{m'}\right)$*. Define both following conditions: (i)* $\underline{c} \preceq_h c$*, and (ii)* $\underline{\Gamma}\left[\underline{\rho}\right]\left[\underline{\rho}'\right]$ *evaluates to* true*. Let substitution* ρ_h *be* $\left\{x_i \leftarrow u_{h(i)}(x) \mid x \in Y \text{ and } i : 1 \leq i \leq m\right\}$*. If conditions (i) and (ii) are satisfied by* $\underline{c}$*,* $\underline{c}'$ *and* c*, then there exists a fourth configuration* $c' = u'_1 \cdots u'_{m'}$ *(of the same size as* $\underline{c}'$*), verifying*[2] $\underline{c}' \preceq_{id_{m'}} c'$*, and such that* $\underline{\Gamma}[\rho_h][\rho']$ *evaluates to* true *for substitution* ρ' *equal to* $\{x_i \leftarrow u'_i(x) \mid x \in Y \text{ and } i : 1 \leq i \leq m'\}$*. The situation is depicted in figure 11.2.*

$$\begin{array}{ccc} c = u_1 \cdots u_n & \leftarrow \underline{\Gamma}[\rho_h][\rho'] \rightarrow & c' = u'_1 \cdots u'_{m'} \\ \curlyvee\!|_h & & \curlyvee\!|_{id_{m'}} \\ \underline{c} = \underline{u}_1 \cdots \underline{u}_m & \leftarrow \underline{\Gamma}[\underline{\rho}][\underline{\rho}'] \rightarrow & \underline{c}' = \underline{u}'_1 \cdots \underline{u}'_{m'} \end{array}$$

Figure 11.2: The existence of a configuration c' such that $\underline{c}' \preceq c'$ and c, c' satisfy $\underline{\Gamma}[\rho_h][\rho']$ follows (lemma 11.2) from the definition of the ordering in $\underline{c} \preceq c$ and the fact that $\underline{c}, \underline{c}'$ also satisfy $\underline{\Gamma}[\underline{\rho}][\underline{\rho}']$.

Proof. Define formulas $\varphi^{\mathcal{B}} \in \mathbb{B}\left(Y'_1, \dots Y'_{m'}\right)$ and $\varphi^{\mathcal{B}} \in \mathbb{N}\left(Y'_1, \dots Y'_{m'}\right)$ as:

$$\begin{array}{rcl} \varphi^{\mathcal{B}} & = & \bigwedge_{i:1 \leq i \leq m'} \left(\bigwedge_{x \in Y^{\mathcal{B}}} x'_i = \underline{u}_i(x)\right) \\ \varphi^{\mathcal{N}} & = & \bigwedge_{i,j:1 \leq i,j \leq m'} \left(\bigwedge_{y,x \in Y^{\mathcal{N}}, (\underline{u}_i(x) \leq \underline{u}_j(y))} \left((\underline{u}_i(x) - \underline{u}_j(y)) \leq (x'_i - y'_j)\right)\right) \end{array}$$

Intuitively, $\varphi^{\mathcal{B}} \wedge \varphi^{\mathcal{N}}$ defines conditions on the values of the variables in a (candidate) configuration $c' = u'_1 \cdots u'_{m'}$ of size m', in order to ensure that $\underline{c}' \preceq_{id_{m'}} c'$[3]. Define $\Gamma \in \mathbb{F}\left(Y_1, \dots, Y_m, Y'_1, \dots, Y'_{m'}\right)$ to be the formula $\Gamma = \underline{\Gamma} \wedge \varphi^{\mathcal{B}} \wedge \varphi^{\mathcal{N}}$. By construction, $\Gamma\left[\underline{\rho}\right]\left[\underline{\rho}'\right]$ is valid for $\underline{\rho}$ and $\underline{\rho}'$ as defined in the lemma 11.2. As a consequence, the formula $\left(\exists\left(Y'_1 \dots \cup Y'_{m'}\right).\Gamma\right)\left[\underline{\rho}\right]$[4] evaluates also to *true*.

---

[2] $id_{m'} : \overline{m'} \rightarrow \overline{m'}$ with $id(i) = i$ for each $i : 1 \leq i \leq m'$

[3] More generally, if $\left(\varphi^{\mathcal{B}} \wedge \varphi^{\mathcal{N}}\right)[\rho_{h'}]$ evaluates to *true* for a substitution $\rho_{h'} = \left\{x'_i \leftarrow u'_{h'(i)}(x) \mid x \in Y \text{ and } i : 1 \leq i \leq m'\right\}$, a configuration $c' = u'_1 \cdots u'_{n'}$ and a strictly monotonic injection $h' : \overline{m'} \rightarrow \overline{n'}$; then $\underline{c}' \preceq_{h'} c'$.

[4] The formula $\exists\left(Y'_1 \dots \cup Y'_{m'}\right).\Gamma$ obtained by elimination in $\Gamma\left(Y_1, \dots, Y_m, Y'_1, \dots, Y'_{m'}\right)$ of the variables in $Y'_1 \dots \cup Y'_{m'}$ is (by closure of Boolean and gap-formulas under projection [Rev93]) in $\mathbb{F}(Y_1, \dots, Y_m)$

We apply lemma 11.1, using the fact that $\underline{c} \preceq_h c$, to deduce the validity of $\left(\exists\left(Y_1' \ldots \cup Y_{m'}'\right).\Gamma\right)[\rho_h]$ for ρ_h defined in lemma 11.2. This means that there is a configuration $c' = u_1', \ldots, u_{m'}'$ such that $\Gamma[\rho_h][\rho']$ evaluates to *true* (ρ' defined in lemma 11.2). We get that $\underline{\Gamma}[\rho_h][\rho']$ and $\underline{c}' \preceq_{id_{m'}} c'$. □

11.3 Approximation

We introduce an over-approximation of the transition relation induced by a parameterized system $\mathcal{P} = (Q, X, T)$. The aim of the over-approximation is to derive an *approximate* transition system $\mathcal{A}(\mathcal{P})$ which is *monotonic* with respect to the ordering $\preceq$ on configurations defined in the previous section.

A transition relation $\xrightarrow{t}$ is said to be *monotonic* if the following holds. For any three configurations $\underline{c}, \underline{c}'$ and c such that $\underline{c} \preceq c$ and $\underline{c} \xrightarrow{t} \underline{c}'$, there exists a fourth configuration c' such that $c \xrightarrow{t} c'$ and $\underline{c}' \preceq c'$. Observe that a parameterized system where all transition relations $\left\{\xrightarrow{t} \mid t \in T\right\}$ are monotonic induces a *monotonic* transition system[5]. The following lemma states that transition relations resulting defined in terms of local and global existential conditions are monotonic.

Lemma 11.3. *Transition relations involving (only) local or global existential conditions are monotonic.*

Proof. We look at the case of a transition of the form depicted in figure 11.1 and involving a global existential condition $\exists_L \theta_1$ where θ_1 is in $\mathbb{F}(X \cup o{\cdot}Y \cup X^{next})$. The other cases of local and global existential conditions can be easily deduced. Consider three configurations $\underline{c} = \underline{u}_1 \cdots \underline{u}_m$, $\underline{c}' = \underline{u}_1' \cdots \underline{u}_m'$, and $c = u_1 \cdots u_n$ such that $\underline{c} \preceq_h c$ and $\underline{c} \xrightarrow{t} \underline{c}'$. Let i, j be the indices in $\underline{c}$ of respectively the process taking the transition t, and the witness process (observe that $j < i$). We show the existence of a fourth configuration $c' = u_1' \cdots u_n'$ such that $c \xrightarrow{t} c'$ and $\underline{c}' \preceq_h c'$. Take $\underline{\Gamma}$ in lemma 11.2 to be:

$$\underline{\Gamma} := \left(\theta_1 \wedge \eta_{\underline{q},\underline{q}'}(q, q')\right)[\rho_1] \wedge \bigwedge_{k:1 \le k \ne i \le m} \left(\bigwedge_{x \in Y} (x_k = x_k')\right)$$

where the substitution ρ_1 is defined as the union $\{x \leftarrow x_i | \, x \in Y\} \cup \left\{o{\cdot}x \leftarrow x_j | \, x \in Y\right\} \cup \{x^{next} \leftarrow x_i' | \, x \in Y\}$. The fact that $\underline{c} \xrightarrow{t} \underline{c}'$ implies that $\underline{\Gamma}\left[\underline{\rho}\right]\left[\underline{\rho}'\right]$ evaluates to *true* with $\underline{\rho} := \{x_i \leftarrow \underline{u}_i(x) | \, x \in Y \text{ and } i : 1 \le i \le m\}$ and $\underline{\rho}' := \{x_i' \leftarrow \underline{u}_i'(x) | \, x \in Y \text{ and } i : 1 \le i \le m\}$. Together with lemma 11.2 and $\underline{c} \preceq_h c$, we deduce the existence of a configuration $\tilde{c} = \tilde{u}_1 \cdots \tilde{u}_m$ such that $\underline{c}' \preceq_{id_m} \tilde{c}$ and $\underline{\Gamma}[\rho_h][\rho']$ evaluates to *true* for the substitutions ρ_h and ρ' respectively defined as $\left\{x_i \leftarrow u_{h(i)}(x) | \, x \in Y \text{ and } i : 1 \le i \le m\right\}$ and $\{x_i' \leftarrow \tilde{u}_i(x) | \, x \in Y \text{ and } i : 1 \le i \le m\}$. This means that $u_{h(k)} = \tilde{u}_k$ for each

[5] see chapter 9 for a definition of monotonic transition systems.

$k : 1 \leq k \neq i \leq m$, that $\left(u_{h(i)}, \tilde{u}_i, u_{h(j)}\right) \models \theta_1$ (observe $h(j) < h(i)$), and that the process state $u_{h(i)}$ is at state $\underline{q}$ while the process state $\tilde{u}_i$ is at state $\underline{q}'$. We choose $c' = u'_1 \cdots u'_n$ such that $u'_{h(k)} = \tilde{u}_k$ for each $k : 1 \leq k \leq m$, and $u'_l = u_l$ for each $l : 1 \leq l \leq n$ such that no $k : 1 \leq k \leq m$ verifies $h(k) = l$. We have $c \xrightarrow{t} c'$ and $\underline{c}' \preceq_h c'$. □

The only transition relations which do not preserve monotonicity are those involving universal global conditions. Intuitively, this is because a larger configuration may contain process states mapping variables or states to values that violate the global condition involved in the universal transition. Therefore, we propose an approximate transition system $\mathcal{A}(\mathcal{P}) = (C, \rightsquigarrow)$ that modifies the semantics of universal quantifiers in such a manner that monotonicity is maintained. In the new semantics, we remove all processes in the configuration which violate the condition of the universal quantifier. Below we describe how this is done.

The set C of configurations in $\mathcal{A}(\mathcal{P}) = (C, \rightsquigarrow)$ is identical to the one in the original transition system $\mathcal{T}(\mathcal{P})$ defined in section 11.1. We define $\rightsquigarrow$ as the union $(\longrightarrow \cup \rightsquigarrow_1)$ where $\rightsquigarrow_1$ (which reflects the approximation of universal quantifiers) is defined as follows. For a sequence of process states $u_1 \cdots u_m$, a formula $\theta \in \mathbb{F}(X \cup \circ{\cdot}Y \cup X^{next})$, and process states u, u', we use $(u_1 \cdots u_m) \ominus (\theta, u, u')$ to denote the sequence derived from $u_1 \cdots u_m$ by deleting all process states u_j such that $(u, u', u_j) \not\models \theta$. To explain this operation intuitively, we recall that a universal global condition requires that the current and next states of the current process (described by u and u' respectively) together with the state of each other process (described by u_j) should satisfy the formula θ. The operation then removes from $u_1 \cdots u_m$ each process whose state u_j does not comply with this condition.

Consider two configurations $c = u_1 \cdots u_n$ and $c' = u'_1 \cdots u'_{n'}$. Let t be a transition rule of the form depicted in figure 11.1, and such that θ is a universal global condition of the form $\Box\theta_1$ where $\Box \in \{\forall_L, \forall_{LR}, \forall_R\}$. We write $c \overset{t}{\rightsquigarrow}_1 c'$ to denote that there exist both i and i' (with $i : 1 \leq i \leq n$, $i' : 1 \leq i' \leq n'$ and $u_i = (q_i, v_i)$, $u'_i = (q'_i, v'_i)$) such that $q_i = \underline{q}$ and $q'_{i'} = \underline{q}'$ and both the following conditions hold:

- $u'_1 \cdots u'_{i'-1} = (u_1 \cdots u_{i-1}) \ominus (\theta_1, u_i, u'_{i'})$ if $\Box \in \{\forall_L, \forall_{LR}\}$, and
- $u'_{i'+1} \cdots u'_{n'} = (u_{i+1} \cdots u_n) \ominus (\theta_1, u_i, u'_{i'})$ if $\Box \in \{\forall_{LR}, \forall_R\}$.

Intuitively, the process taking the approximated universal transition has index i in c and index i' in c'. Notice that $\xrightarrow{t} \subseteq \overset{t}{\rightsquigarrow}_1$ in case t involves a universal global transition. The transition system $\mathcal{A}(\mathcal{P})$ over-approximates the original transition system $\mathcal{T}(\mathcal{P})$. The following lemma states that $\overset{t}{\rightsquigarrow}_1$ is monotonic. This is sufficient (when considered with lemma 11.3) to conclude that the approximate transition system $\mathcal{A}(\mathcal{P})$ is also monotonic.

Lemma 11.4. *Assume a transition t involving (only) a global universal condition. The resulting transition relation* $\overset{t}{\rightsquigarrow}_1$ *is monotonic.*

Proof. Let the transition t be of the form depicted in figure 11.1 and involving a global universal condition $\forall_L \theta_1$ where θ_1 is in $\mathbb{F}(X \cup o{\cdot}Y \cup X^{next})$. The other cases of global universal conditions can be easily deduced. We proceed in a similar manner to the proof of lemma 11.3. Consider three configurations $\underline{c} = \underline{u}_1 \cdots \underline{u}_m$, $\underline{c}' = \underline{u}'_1 \cdots \underline{u}'_{m'}$, and $c = u_1 \cdots u_n$ such that $\underline{c} \preceq_h c$ and $\underline{c} \overset{t}{\rightsquigarrow}_1 \underline{c}'$. We show the existence of a fourth configuration $c' = u'_1 \cdots u'_{n'}$ such that $c \overset{t}{\rightsquigarrow}_1 c'$ and $\underline{c}' \preceq c'$. Take $\underline{\Gamma}$ in lemma 11.2 to be:

$$\underline{\Gamma} = \left(\eta_{\underline{q},\underline{q}'}(q,q') \wedge \bigwedge_{j:1 \le j < i} \theta_1 \left[\rho^j\right] \right) [\rho] \wedge \bigwedge_{k:1 \le k \ne i \le m'} \left(\bigwedge_{x \in Y} (x_k = x'_k) \right)$$

where the substitution ρ is defined as the union $\{x \leftarrow x_i |\, x \in Y\} \cup \{x^{next} \leftarrow x'_i |\, x \in Y\}$, while substitution ρ^j corresponds to $\{o{\cdot}x \leftarrow x_j |\, x \in Y\}$. Let $\underline{h} : \overline{m'} \to \overline{m}$ be the strictly monotonic injection corresponding to deleting from $\underline{c}$ processes that do not satisfy the universal condition in $\underline{c} \overset{t}{\rightsquigarrow}_1 \underline{c}'$. The fact that $\underline{c} \overset{t}{\rightsquigarrow}_1 \underline{c}'$ implies the validity of $\underline{\Gamma}\left[\underline{\rho}\right]\left[\underline{\rho}'\right]$ for substitutions $\underline{\rho} := \left\{x_i \leftarrow \underline{u}_{\underline{h}(i)}(x) |\, x \in Y \text{ and } i : 1 \le i \le m'\right\}$ and $\underline{\rho}' := \{x'_i \leftarrow \underline{u}'_i(x) |\, x \in Y \text{ and } i : 1 \le i \le m'\}$. Together with lemma 11.2 and $\underline{c} \preceq_h c$, we deduce the existence of a configuration $\tilde{c} = \tilde{u}_1 \cdots \tilde{u}_{m'}$ such that $\underline{c}' \preceq_{id_{m'}} \tilde{c}$ and $\underline{\Gamma}[\rho_h][\rho']$ evaluates to *true* for the substitutions ρ_h and ρ' respectively defined by $\left\{x_i \leftarrow u_{h(\underline{h}(i))}(x) |\, x \in Y \text{ and } i : 1 \le i \le m'\right\}$ and $\{x'_i \leftarrow \tilde{u}_i(x) |\, x \in Y \text{ and } i : 1 \le i \le m'\}$. This means that (i) $u_{h(\underline{h}(k))} = \tilde{u}_k$ for each $k : 1 \le k \ne i \le m'$, that (ii) $\left(u_{h(\underline{h}(i))}, \tilde{u}_i, u_{h(\underline{h}(k))}\right) \models \theta_1$ for each $k : 1 \le k < i$, and that (iii) process states $u_{h(\underline{h}(i))}$ and $\tilde{u}_i$ are respectively at states $\underline{q}$ and $\underline{q}'$.

Observe that these three consequences imply the existence of a configuration denoted by $c' = u'_1 \cdots u'_{n'}$ (such that $c \overset{t}{\rightsquigarrow}_1 c'$) and of a strictly monotonic injection $\hbar : \overline{m'} \to \overline{n'}$ (such that $\tilde{u}_k = u'_{\hbar(k)}$). Intuitively, c' is obtained by adding to $\tilde{c}$ the missing process states in c that do not violate the universal condition. We have $c \overset{t}{\rightsquigarrow}_1 c'$ and $\underline{c}' \preceq_{id_{m'}} \tilde{c} \preceq_{\hbar} c'$. □

We use $c \rightsquigarrow c'$ to denote that $c \overset{t}{\rightsquigarrow} c'$ for some $t \in T$; and use $\overset{*}{\rightsquigarrow}$ to denote the reflexive transitive closure of $\rightsquigarrow$.

11.4 Constraints, Operations and Termination

In this section, we present the constraints we use in the scheme introduced in chapter 9. We also specify how to perform the operations necessary for the application of the scheme. For the rest of this section, we fix a parameterized systems $\mathcal{P} = (Q, X, T)$ and the induced approximate transition system $\mathcal{A}(\mathcal{P}) = (C, \rightsquigarrow)$.

Constraints

We define the *constraints* we use as a symbolic representation for sets of configurations. A *constraint* ϕ is a pair (m,ψ), where $m \in \mathbb{N}$ is a natural number, and $\psi \in \mathbb{F}(Y_1 \cup Y_2 \cup \cdots \cup Y_m)$. Intuitively, a configuration satisfying ϕ should contain at least m processes (indexed by $1,\ldots,m$). The constraint ϕ uses the elements of the set Y_i to refer to the local states and variables of process i. The values of these states and variables are constrained by the formula ψ. Formally, consider a configuration $c = u_1 \cdots u_n$ and a constraint $\phi = (m,\psi)$. Let $h : \overline{m} \mapsto \overline{n}$ be a strictly monotonic injection. We write $c \models_h \phi$ to denote the validity of the formula $\psi[\rho]$ where $\rho = \{x_i \leftarrow u_{h(i)}(x) |\ x \in Y \text{ and } i : 1 \leq i \leq m\}$. We write $c \models \phi$ to denote that $c \models_h \phi$ for some h; and define $[\![\phi]\!] = \{c |\ c \models \phi\}$. For a (finite) set of constraints Φ, we define $[\![\Phi]\!] = \bigcup_{\phi\in\Phi} [\![\phi]\!]$. The following lemma follows from the definitions.

Lemma 11.5. *For each constraint ϕ, the set $[\![\phi]\!]$ is upward closed.*

Proof. direct application of lemma 11.1. □

In all the examples we consider, the set F in the definition of the coverability problem in chapter 9 can be represented by a finite set Φ_F of constraints. The coverability question can then be answered by checking whether $I \xrightarrow{*} [\![\Phi_F]\!]$. We show how to perform the three operations on constraints used in the scheme presented in chapter 9, namely intersection with initial configurations, computing *Pre*, and checking entailment. We finish this section by explaining why termination of the analysis (on the approximate system) is guaranteed for linear parameterized systems with finite-processes (i.e. no numerical local variables).

Intersection with Initial States

For a constraint $\phi = (m,\psi)$, we have $(I \bigcap [\![\phi]\!]) \neq \emptyset$ if and only if $\psi[\rho_{initial}]$ is valid for the substitution $\rho_{initial} = \{x \leftarrow u_{init}(x) |\ x \in Y\}$.

Computing Pre

Given a constraint ϕ', We show how to compute the set $Pre(\phi')$. We will compute $Pre(\phi')$ as $\bigcup_{t\in T} Pre_t(\phi')$, where $[\![Pre_t(\phi')]\!] = \left\{c |\ \exists .c' \in [\![\phi']\!].\ c \overset{t}{\rightsquigarrow} c'\right\}$. Consider a transition rule of the form depicted in figure 11.1 and a constraint $\phi' = (m',\psi')$ where $\psi' \in \mathbb{F}(Y_1 \cup \ldots \cup Y_{m'})$. Below, we describe (and show the correctness of) how to compute $Pre_t(\phi')$ as a finite set of constraints. The definition has several cases depending on the condition θ (local, existential, or global condition).

First, we fix some notation. For a strictly monotonic injection $h : \overline{m} \to \overline{n}$ we define the function $pos_h : \overline{n} \to \overline{m+1}$ as follows. For each $i : 1 \leq i \leq n$, $pos_h(i)$ takes the (if any) smallest $j : 1 \leq j \leq m$ such that $i \leq h(j)$. If no such j exists (i.e. $h(m) < i$), $pos_h(i)$ takes $m+1$. For a formula φ in $\mathbb{F}(Y_1 \cup Y_2 \cup \cdots \cup Y_m)$, we write $\varphi \oplus i$ (for some $i : 1 \leq i \leq m+1$) to mean the formula in $\mathbb{F}(Y_1 \cup Y_2 \cup \cdots \cup Y_{m+1})$ such that $\varphi \oplus i = \varphi\left[\bigcup_{j:i\leq j\leq m} \{x_j \leftarrow x_{j+1} |\ x \in Y\}\right]$. Intuitively, $\varphi \oplus i$

corresponds to the insertion in φ of a (free) copy Y_i of the set Y, (i.e. insertion of a process at position i). For two indices $i, j : 1 \leq i, j \leq n+2$, such that $i < j$, we write $\varphi \oplus i, j$ to mean the formula $((\varphi \oplus i) \oplus j)$.

Moreover, given a strictly monotonic injection $h : \overline{m} \to \overline{n}$ and some $i : 1 \leq i \leq n$, we write $h \oplus i$ to mean the strictly monotonic injection defined as follows. If i is in the image of h, then $h \oplus i$ coincides with h. If i is not in the image of h, then $h \oplus i$ is defined from $\overline{m+1}$ to $\overline{n}$. In this case, there are two possibilities depending on whether there exists $j_0 : 1 \leq j_0 \leq m$ such that $i < h(j_0)$. If no such j_0 exists, then define $(h \oplus i)(j) = h(j)$ for each $j : 1 \leq j \leq m$, and $(h \oplus i)(i) = m+1$. Otherwise, let j_0^{min} be the minimum of these j_0. Define $(h \oplus i)(j)$ to take $h(j)$ for $j : 1 \leq j < j_0^{min} \leq m$; i for j_0^{min}; and $h(j-1)$ for $j : j_0^{min} < j \leq m+1$. Intuitively, the strictly monotonic injection $h \oplus i$ adapts h with respect to the possible insertion of process i in the constraint. We also write, for indices $i, j : 1 \leq i, j \leq n+2$, such that $i < j$, that $\psi \oplus i, j$ to mean the formula $((\psi \oplus i) \oplus j)$.

Local Conditions.
The condition θ is local, and hence in $\mathbb{F}(X \cup X^{next})$. We define $Pre_t(\phi')$ to be the smallest set containing the constraints of the two following forms:

- $\phi_i = (m', \psi_i)$ with, for each $i : 1 \leq i \leq m'$, ψ_i given by:

$$\exists Y_i^{next}. \left(\left(\eta_{\underline{q},\underline{q}'}(q, q') \wedge \theta \right) \left[\rho_1^i \right] \wedge \psi' \left[\rho_2^i \right] \right)$$

  Substitutions ρ_1^i and ρ_2^i are respectively defined by the sets $\{x^{next} \leftarrow x_i^{next} | \, x \in Y\} \cup \{x \leftarrow x_i | \, x \in Y\}$ and $\{x_i \leftarrow x_i^{next} | \, x \in Y\}$. Each constraint ϕ_i corresponds to the case where the process performing the transition is part of the definition of the constraint ϕ' (namely, it is the process with index i in ψ'). Substitution ρ_1^i renames current and next-value variables (Y and Y^{next}) appearing in $\eta_{\underline{q},\underline{q}'}(q, q') \wedge \theta$ to corresponding variables in Y_i and Y_i^{next}. Substitution ρ_2^i renames variables of process i in ψ' to match them with next-value variables of the same process in $\left(\eta_{\underline{q},\underline{q}'}(q, q') \wedge \theta \right) \left[\rho_1^i \right]$. Next-value variables are eliminated by projection (existential quantification). The set Y_i is used in the result to constrain values of local variables of i before it took the transition t.
- $\phi_{\underline{i}} = (m'+1, \psi_{\underline{i}})$ for each $i : 1 \leq i \leq m'+1$, with $\psi_{\underline{i}}$ given by:

$$\exists Y_i^{next}. \left(\left(\eta_{\underline{q},\underline{q}'}(q, q') \wedge \theta \right) \left[\rho_1^i \right] \wedge \left(\psi' \oplus i \right) \left[\rho_2^i \right] \right)$$

  where substitutions ρ_1^i and ρ_2^i are defined like in the previous case. The difference is that the process taking the transition is not part of ϕ'. This component is between processes (roughly) corresponding to indices $i-1$ and i in ϕ'. We therefore insert at the appropriate location a process in ϕ' obtaining $(\psi' \oplus i)$. Notice that, in this case, we have $\phi' \sqsubseteq \phi$. In other words, the newly generated constraint ϕ entails by the original constraint ϕ'. Therefore, the new constraint ϕ needs not be considered during the backwards reachability analysis scheme of chapter 9.

Lemma 11.6 (local predecessors)**.** *For any transition t involving (only) a local condition, we have* $[\![Pre_t(\phi')]\!] = \left\{c |\ \exists.c' \in [\![\phi']\!].\ c \overset{t}{\rightsquigarrow} c'\right\}$.

Proof. Can be carried in a similar manner to the existential or universal cases below. $\square$

Existential Conditions.

The condition θ is of the form $\Box\theta_1$, where θ_1 is in $\mathbb{F}(X \cup o{\cdot}Y \cup X^{next})$ and $\Box$ in $\{\exists_L, \exists_{LR}, \exists_R\}$. Suppose $\Box = \exists_L$, the other cases can be easily deduced. The main difference compared to local conditions is that we have to consider two processes. More precisely, in addition to the process (with index i) which performs the transition, we need to also consider a witness process (index j) which enables the transition. Define substitutions $\rho_1^i = \{x \leftarrow x_i |\ x \in Y\} \cup \{x^{next} \leftarrow x_i^{next} |\ x \in Y\}$, $\rho_2^j = \left\{o{\cdot}x \leftarrow x_j |\ x \in Y\right\}$ and $\rho_3^i = \{x_i \leftarrow x_i^{next} |\ x \in Y\}$. Substitutions ρ_1^i and ρ_3^i play the same role as ρ_1^i and ρ_2^i in the case of local conditions. Substitution ρ_2^j encodes the conditions imposed on the witness process (with index j). The definition of $Pre_t(\phi')$ consists of several cases corresponding to whether or not the processes i and j are parts of ϕ'. We define $Pre_t(\phi')$ to be the smallest set containing the following constraints.

- $\phi_{i,j} = (m', \psi_{i,j})$ for each $i, j : 1 \le j < i \le m'$, with $\psi_{i,j}$ given by:

$$\exists Y_i^{next}. \left(\left(\eta_{\underline{q},\underline{q}'}(q, q') \wedge \theta_1 \left[\rho_2^j\right]\right) \left[\rho_1^i\right] \wedge \psi' \left[\rho_3^i\right]\right)$$

  Each constraint $\phi_{i,j}$ corresponds to the case where the process performing the transition, and the witness process are parts of the definition of the constraint ϕ'. Formula $\left(\eta_{\underline{q},\underline{q}'}(q, q') \wedge \theta_1\right)$ is defined in a similar manner to the local case above (with the difference that θ is replaced by θ_1).

- $\phi_{i,\underline{j}} = \left(m' + 1, \psi_{i,\underline{j}}\right)$ for each $i, j : 1 \le j < i \le m' + 1$, with $\psi_{i,\underline{j}}$ given by:

$$\exists Y_i^{next}. \left(\left(\eta_{\underline{q},\underline{q}'}(q, q') \wedge \theta_1 \left[\rho_2^j\right]\right) \left[\rho_1^i\right] \wedge \left(\psi' \oplus j\right) \left[\rho_3^i\right]\right)$$

  Here, the process taking the transition is part of ϕ', while the witness process is not, and is therefore inserted.
  Notice that this case gives rise to a constraint $\phi_{i,\underline{j}}$ whose size is larger than the size of the original constraint ϕ'. Furthermore, in contrast to the case of local conditions, the new constraint $\phi_{i,\underline{j}}$ does not necessarily entail ϕ'. This makes the sizes of constraints which arise in the reachability analysis unbounded in general.

- $\phi_{\underline{i},j} = (m' + 1, \psi_{\underline{i},j})$ for each $i, j : 1 \le j < i \le m' + 1$, with $\psi_{\underline{i},j}$ given by:

$$\exists Y_i^{next}. \left(\left(\eta_{\underline{q},\underline{q}'}(q, q') \wedge \theta \left[\rho_2^j\right]\right) \left[\rho_1^i\right] \wedge \left(\psi' \oplus i\right) \left[\rho_3^i\right]\right)$$

  Here, the witness process is part of ϕ' while the process taking the transition is not, and is therefore added. In a similar manner to the case of local

conditions, the generated constraints entail ϕ' and can therefore be safely discarded in the reachability analysis.

- $\phi_{\underline{i},\underline{j}} = \left(m'+2, \psi_{\underline{i},\underline{j}}\right)$ for each $i, j : 1 \leq j < i \leq m'+2$, with $\psi_{\underline{i},\underline{j}}$ given by:

$$\exists Y_i^{next}. \left(\left(\eta_{\underline{q},\underline{q}'}(q,q') \wedge \theta\left[\rho_2^j\right]\right)\left[\rho_1^i\right] \wedge \left(\psi' \oplus j, i\right)\left[\rho_3^i\right]\right)$$

Here, neither of the process taking the transition, nor the witness process is part of ϕ'. Both are added. The generated constraints do entail ϕ' and can be safely discarded in the reachability analysis.

Lemma 11.7. *For any transition t involving (only) an existential global condition, we have $[\![Pre_t(\phi')]\!] = \left\{c \mid \exists .c' \in [\![\phi']\!].\ c \overset{t}{\rightsquigarrow} c'\right\}$.*

Proof. Recall $\overset{t}{\rightsquigarrow} = \overset{t}{\longrightarrow}$ for t existential. We show both directions for $t = \exists_L \theta_1$.

$\subseteq$: Fix $\phi' = (m', \psi')$, and assume $\phi = (m, \psi)$ with $(m, \psi) \in \left\{(m', \psi_{i,j}), \left(m'+1, \psi_{i,\underline{j}}\right), (m'+1, \psi_{\underline{i},j}), \left(m'+2, \psi_{\underline{i},\underline{j}}\right)\right\}$ as defined in $Pre_t(\phi')$. Let $c = u_1, \ldots, u_n$ be a configuration in $[\![\phi]\!]$. We show the existence of a configuration $c' = u_1', \ldots, u_n'$ in $[\![\phi']\!]$ such that $c \overset{t}{\longrightarrow} c'$
The fact that c is in $[\![\phi]\!]$ implies the existence of a strictly monotonic injection $h : \overline{m} \to \overline{n}$ such that $\psi[\rho_h]$ is valid for the substitution $\rho_h = \left\{x_i \leftarrow u_{h(i)}(x) \mid x \in Y \text{ and } 1 \leq i \leq m\right\}$. Recall that $\psi = \exists Y_i^{next}. \left(\left(\eta_{\underline{q},\underline{q}'}(q,q') \wedge \theta\left[\rho_2^j\right]\right)\left[\rho_1^i\right] \wedge \psi''\left[\rho_3^i\right]\right)$ with $\psi'' \in \mathbb{F}(Y_1 \cup Y_2 \cup \cdots \cup Y_m)$ equal to ψ' in the first case, to $\psi' \oplus j$ in the second, to $\psi' \oplus i$ in the third, and in the fourth to $(\psi' \oplus i) \oplus j$. We deduce the existence of a process state, call it u, such that $\left(\left(\eta_{\underline{q},\underline{q}'}(q,q') \wedge \theta\left[\rho_2^j\right]\right)\left[\rho_1^i\right] \wedge \psi''\left[\rho_3^i\right]\right)[\rho_h][\rho]$ is valid with $\rho = \{x_i^{next} \leftarrow u(x) \mid x \in Y\}$. Observe that $u_{h(i)}$ is at state $\underline{q}$ and u at state $\underline{q}'$. Also, $\left(u_{h(i)}, u, u_{h(j)}\right) \models \theta_1$. We deduce that $c \overset{t}{\longrightarrow} c'$ for $c' = u_1' \cdots u_n'$ with $u_k' = u_k$ for each $k : 1 \leq k \neq i \leq n$ and $u_i' = u$.
We show in the following that c' is in $[\![\phi']\!]$. Recall that $\left(\psi''\left[\rho_3^i\right]\right)[\rho_h][\rho]$ is valid. This says that $\psi''\left[\rho_h'\right]$ evaluates to *true* for $\rho_h' = \left\{x_i \leftarrow u_{h(i)}'(x) \mid x \in Y \text{ and } 1 \leq i \leq m\right\}$. We conclude $\psi'[\rho_\hbar]$ is valid for some $\rho_\hbar = \left\{x_i \leftarrow u_{\hbar(i)}'(x) \mid x \in Y \text{ and } 1 \leq i \leq m'\right\}$ where the strictly monotonic injection $\hbar$ depends on the considered case. More concretely, in the first case $\hbar = h$, while in the other three cases $\hbar \neq h$. In the second $h = \hbar \oplus j$ (j is not in image of $\hbar$), in the third $h = \hbar \oplus i$ (i is not in image of $\hbar$), and in the fourth $h = (\hbar \oplus i) \oplus j$ (i, j are not in image of $\hbar$, hence $(\hbar \oplus i) \neq h$).

$\supseteq$: Fix $c = u_1 \cdots u_n$, $c' = u_1' \cdots u_n'$ and ϕ' such that $c \overset{t}{\longrightarrow} c'$ and $\phi' = (m', \psi')$. Let $\underline{i}$ and $\underline{j}$ be the indices in ϕ' of respectively the process taking the transition, and the witness process. We show $c \in [\![\phi]\!]$ for some $\phi \in Pre_t(\phi')$.

The fact that c' is in $[\![\phi']\!]$ means $\psi'[\rho_{h'}]$ is valid for substitution $\rho_{h'}$ defined by $\left\{x_i \leftarrow u'_{h'(i)}(x) |\, x \in Y \text{ and } i : 1 \le i \le m'\right\}$ where $h' : \overline{m'} \to \overline{n}$ is a strictly monotonic injection. Fix formula ψ'' and strictly monotonic injection h depending on which of $\underline{i}$ and $\underline{j}$ is in image of h'. More concretely, define $\psi'' = \psi'$ and $h = h'$ if both of them are in image of h'; $\psi'' = \left(\psi' \oplus pos_{h'}(\underline{j})\right)$ and $h = \left(h' \oplus pos_{h'}(\underline{j})\right)$ if only $\underline{i}$ is in image of h'; $\psi'' = (\psi' \oplus pos_{h'}(\underline{i}))$ and $h = h' \oplus pos_{h'}(\underline{i})$ if only $\underline{j}$ is in image of h'; and finally $\psi'' = \left((\psi' \oplus pos_{h'}(\underline{i})) \oplus pos_{h'}(\underline{j})\right)$ and $h = \left((h' \oplus pos_{h'}(\underline{i})) \oplus pos_{h'}(\underline{j})\right)$ if neither of them is in image of h'. Define m to be equal to m' in the first case, $m'+1$ in the second and third cases, and $m'+2$ in the last case. We have $1 \le pos_{h'}(\underline{j}) < pos_{h'}(\underline{i}) \le m$ and $\psi'' \in \mathbb{F}(Y_1 \ldots \cup Y_m)$. Write i_0 and j_0 to respectively mean $pos_{h'}(\underline{i})$ and $pos_{h'}(\underline{j})$. Observe $h(i_0) = \underline{i}$ and $h(j_0) = \underline{j}$. Define the substitution $\rho_3^{i_0}$ by $\left\{x_{i_0} \leftarrow x_{i_0}^{next} |\, x \in Y\right\}$. By construction, we have that the formula $\left(\psi''\left[\rho_3^{i_0}\right]\right)\left[\rho'_h\right]\left[\rho'_{i_0}\right]$ is valid for $\rho'_h = \left\{x_i \leftarrow u'_{h(i)}(x) |\, x \in Y \text{ and } i : 1 \le i \le m\right\}$, and ρ'_{i_0} defined by $\left\{x_{i_0}^{next} \leftarrow u'_{h(i_0)}(x) |\, x \in Y\right\}$.

Three consequences follow from the fact that $c \xrightarrow{t} c'$. (i) $u_{\underline{i}}$ is at state $\underline{q}$ and $u'_{\underline{i}}$ is at state $\underline{q}'$; (ii) $u_k = u'_k$ for each $k : 1 \le k \ne \underline{i} \le n$; and (iii) $\left(u_{\underline{i}}, u'_{\underline{i}}, u_{\underline{j}}\right) \models \theta_1$. Recall the substitutions $\rho_1^{i_0} = \{x \leftarrow x_{i_0} |\, x \in Y\} \cup \left\{x^{next} \leftarrow x_{i_0}^{next} |\, x \in Y\right\}$, and $\rho_2^{j_0} = \left\{o{\cdot}x \leftarrow x_{j_0} |\, x \in Y\right\}$ defined in the description of the *Pre* computation above. We have $\left(\left(\eta_{\underline{q},\underline{q}'}(q,q') \wedge \theta\left[\rho_2^{j_0}\right]\right)\left[\rho_1^{i_0}\right]\right)\left[\rho'_h\right]\left[\rho'_{i_0}\right]$ is valid with substitutions ρ'_h and ρ'_{i_0} described above. Recall $1 \le j_0 < i_0 \le m$. We conclude $\psi\left[\rho'_h\right]$ is valid, and hence c is in $[\![\phi]\!]$.

□

Universal Conditions.

The condition θ is of the form $\Box\theta_1$, where θ_1 is in $\mathbb{F}(X \cup o{\cdot}Y \cup X^{next})$ and $\Box$ is in $\{\forall_L, \forall_{LR}, \forall_R\}$. Suppose $\Box = \forall_L$, the other cases can be easily deduced. We define $Pre_t(\phi')$ to be the smallest set containing the constraints:

- $\phi_i = (m', \psi_i)$ for each $i : 1 \le i \le m'$, with ψ_i given by:

$$\exists Y_i^{next}.\left(\left(\eta_{\underline{q},\underline{q}'}(q,q') \wedge \bigwedge_{1 \le j < i \le m'} \theta_1\left[\rho_2^j\right]\right)\left[\rho_1^i\right] \wedge \psi'\left[\rho_3^i\right]\right)$$

  The constraint ϕ_i corresponds to the case where the process (index i) performing the transition is part of the definition of the constraint ϕ'. The formula $\left(\eta_{\underline{q},\underline{q}'}(q,q') \wedge \theta_1\right)$ and the substitutions ρ_1^i, ρ_2^j, and ρ_3^i are identical to the existential case. The difference is that we apply θ_1 to all processes j to the left of i (and not only to one witness to the left like for $\Box = \exists_L$).

- $\phi_{\underline{i}} = (m'+1, \psi_{\underline{i}})$ for each $i : 1 \leq i \leq m'+1$, with $\psi_{\underline{i}}$ given by:

$$\exists Y_i^{next}. \left(\left(\eta_{\underline{q},\underline{q}'}(q,q') \wedge \bigwedge_{1 \leq j < i \leq m'+1} \theta_1 \left[\rho_2^j\right] \right) \left[\rho_1^i\right] \wedge \left(\psi' \oplus i\right) \left[\rho_3^i\right] \right)$$

The constraint $\phi_{\underline{i}}$ corresponds to the case where the process i is not part of the definition of the constraint ϕ'. The formula $\left(\eta_{\underline{q},\underline{q}'}(q,q') \wedge \theta_1\right)$ and the substitutions ρ_1^i, ρ_2^j, and ρ_3^i are identical to the previous case. This constraint entails ϕ' and can therefore be safely discarded in the reachability analysis.

Lemma 11.8. *For any transition t involving (only) a universal global condition, we have* $[\![Pre_t(\phi')]\!] = \left\{c \mid \exists . c' \in [\![\phi']\!].\ c \overset{t}{\rightsquigarrow} c'\right\}$.

Proof. We show both directions for $t = \forall_L \theta_1$.

$\subseteq$: Fix $\phi' = (m', \psi')$, and assume $\phi = (m, \psi)$ with $(m, \psi) \in \left\{(m', \psi_i), (m'+1, \psi_{\underline{i}})\right\}$ as defined by $Pre_t(\phi')$. Let $c = u_1, \ldots, u_n$ be a configuration in $[\![\phi]\!]$. We show the existence of a configuration $c' = u_1', \ldots, u_{n'}'$ in $[\![\phi']\!]$ such that $c \overset{t}{\rightsquigarrow} c'$.
The fact that c is in $[\![\phi]\!]$ implies the existence of a strictly monotonic injection $h : \overline{m} \to \overline{n}$ such that $\psi[\rho_h]$ is valid for the substitution ρ_h equal to $\left\{x_k \leftarrow u_{h(k)}(x) \mid x \in Y \text{ and } 1 \leq k \leq m\right\}$. There exists an $i : 1 \leq i \leq m$ such that $\left(\left(\eta_{\underline{q},\underline{q}'}(q,q') \wedge \bigwedge_{1 \leq j < i \leq m} \theta_1 \left[\rho_2^j\right]\right) \left[\rho_1^i\right] \wedge \psi'' \left[\rho_3^i\right]\right) [\rho_h]$ is valid, with ψ'' equal to ψ' if $m = m'$, or to $\psi' \oplus i$ if $m = m'+1$.
This means, there exists a process state u (define $\rho = \{x_i^{next} \leftarrow u(x) \mid x \in Y\}$) s.t. the formula $\left(\left(\eta_{\underline{q},\underline{q}'}(q,q') \wedge \bigwedge_{1 \leq j < i \leq m} \theta_1 \left[\rho_2^j\right]\right) \left[\rho_1^i\right] \wedge \psi'' \left[\rho_3^i\right]\right) [\rho_h] [\rho]$ is valid. This validity has the following consequences. In c, $u_{h(i)}$ is at state $\underline{q}$, while u is at state $\underline{q}'$. Also, $\left(u_{h(i)}, u, u_{h(j)}\right) \models \theta_1$ for each $i, j : 1 \leq j < i \leq m$. We define the configurations $c_h = u_{h(1)} \cdots u_{h(m)}$ and $c_h' = u_{h(1)}' \cdots u_{h(m)}'$ where $u_{h(k)}' = u_{h(k)}$ for each $k : 1 \leq k \neq i \leq m$, and $u_{h(i)}' = u$. By construction, we have $c_h \xrightarrow{t} c_h'$. Observe that c_h' is in $[\![(m', \psi')]\!]$ or in $[\![(m'+1, \psi' \oplus i)]\!]$. In both cases c_h' is in $[\![\phi']\!]$.
Let $c' = u_1' \cdots u_{n'}'$ be a configuration such that the three following conditions hold. (i) there is an $i' : 1 \leq i' \leq n'$ with $u_{i'}' = u$, and it is the case that (ii) $u_{i'+1}' \cdots u_{n'}' = u_{h(i)+1} \cdots u_n$, and that (iii) $u_1' \cdots u_{i'-1}' = u_1 \cdots u_{h(i)-1} \ominus \left(\theta_1, u_{h(i)}, u_{i'}'\right)$. Such a configuration exists, and verifies $c_h' \preceq c'$. Intuitively, c_h' is obtained by deleting in c processes that violate the universal condition. By construction of c', we have $c \overset{t}{\rightsquigarrow} c'$. Denotations of constraints are upward closed, and therefore $c' \in [\![\phi']\!]$.

$\supseteq$: Fix $c = u_1 \cdots u_n$, $c' = u_1' \cdots u_{n'}'$ and ϕ' such that $c \overset{t}{\rightsquigarrow} c'$ and $\phi' = (m', \psi')$. Let $\underline{i}$ and $\underline{i}'$ be the indices (respectively in c and in c') of the process taking the transition. We show $c \in [\![\phi]\!]$ for some $\phi \in Pre_t(\phi')$.
The fact that c' is in $[\![\phi']\!]$ means $\psi'[\rho_{h'}]$ is valid for substitution $\rho_{h'}$ defined by $\left\{x_i \leftarrow u_{h'(i)}'(x) \mid x \in Y \text{ and } i : 1 \leq i \leq m'\right\}$ where $h' : \overline{m'} \to \overline{n}$ is a strictly

monotonic injection. We let the formula ψ'' and the strictly monotonic injection h respectively be either: ψ' and h' if $\underline{i}'$ is in image of h', or $\psi' \oplus pos_{h'}(\underline{i}')$ and $h' \oplus \underline{i}'$ if $\underline{i}'$ is not in image of h'. In the first case we let m equal m', and in the second $m'+1$. Write i_0 to mean $pos_{h'}(\underline{i}')$. Observe $h(i_0) = \underline{i}$. Define the substitution $\rho_3^{i_0}$ as the set $\left\{x_{i_0} \leftarrow x_{i_0}^{next} \mid x \in Y\right\}$. Define also substitutions ρ'_{i_0} and ρ'_h respectively by $\left\{x_{i_0}^{next} \leftarrow u'_{h(i_0)}(x) \mid x \in Y\right\}$ and $\left\{x_i \leftarrow u'_{h(i)}(x) \mid x \in Y \text{ and } i : 1 \leq i \leq m\right\}$. The formula $\left(\psi''\left[\rho_3^{\underline{i}}\right]\right)\left[\rho'_h\right]\left[\rho'_{\underline{i}}\right]$ is valid as a consequence of $\psi'\left[\rho_{h'}\right]$ evaluating to *true*.
Three consequences follow from the fact that $c \overset{t}{\rightsquigarrow} c'$ where the process taking the approximate transition has index $\underline{i}$ in c and index $\underline{i}'$ in c'. (i) $u_{\underline{i}}$ is at state $\underline{q}$ and $u'_{\underline{i}'}$ is at state $\underline{q}'$; (ii) $u_{\underline{i}+1} \cdots u_n = u'_{\underline{i}'+1} \cdots u'_{n'}$ and (iii) $u'_1 \cdots u'_{\underline{i}'-1} = u_1 \cdots u_{\underline{i}-1} \ominus \left(\theta_1, u_{\underline{i}}, u'_{\underline{i}'}\right)$. Recall the substitutions $\rho_1^{i_0} = \{x \leftarrow x_{i_0} \mid x \in Y\} \cup \left\{x^{next} \leftarrow x_{i_0}^{next} \mid x \in Y\right\}$, and $\rho_2^{j_0} = \left\{o{\cdot}x \leftarrow x_{j_0} \mid x \in Y\right\}$ defined in the description of the *Pre* computation for approximate universal transitions. We have $\left(\left(\eta_{\underline{q},\underline{q}'}(q,q') \wedge \bigwedge_{1 \leq j_0 < i_0 \leq m} \theta_1\left[\rho_2^{j_0}\right]\right)\left[\rho_1^{i_0}\right]\right)\left[\rho'_h\right]\left[\rho'_{i_0}\right]$ is valid with substitutions ρ'_h and ρ'_{i_0} described above.
Let $\hbar : \overline{n'} \to \overline{n}$ be the strictly monotonic injection whose image are the indices in c of the processes that have not been deleted when deriving c' in $c \overset{t}{\rightsquigarrow} c'$. Define the substitutions ρ_h, respectively ρ_{i_0}, by replacing h by $\hbar \circ h$ in the definition of ρ'_h, respectively ρ'_{i_0}. This results in the validity of $\left(\left(\eta_{\underline{q},\underline{q}'}(q,q') \wedge \bigwedge_{1 \leq j_0 < i_0 \leq m} \theta_1\left[\rho_2^{j_0}\right]\right)\left[\rho_1^{i_0}\right]\right)\left[\rho_h\right]\left[\rho_{i_0}\right]$, and of $\left(\psi''\left[\rho_3^{\underline{i}}\right]\right)\left[\rho_h\right]\left[\rho_{\underline{i}}\right]$. We conclude $\psi\left[\rho_h\right]$ is valid, and hence c is in $[\![\phi]\!]$.

□

Entailment

Consider two constraints $\phi = (m, \psi)$, and $\phi' = (m', \psi')$. Let $\mathcal{H}(\phi, \phi')$ be the set of strictly monotonic injections $h : \overline{m} \to \overline{m'}$. We write ρ_h, to denote the substitution $\left\{x_i \leftarrow x_{h(i)} \mid x \in Y \text{ and } 1 \leq i \leq m\right\}$. For a configuration $c = u_1, \ldots, u_n$, we write ρ_c to denote the substitution $\{x_i \leftarrow u_i(x) \mid x \in Y \text{ and } 1 \leq i \leq n\}$. The following lemma gives a logical characterization which allows the computation of the entailment relation.

Lemma 11.9. *Given two constraints $\phi = (m, \psi)$, and $\phi' = (m', \psi')$, we have $\phi \sqsubseteq \phi'$ iff*

$$\forall y_1 \cdots y_k. \left(\psi'(y_1, \ldots, y_k) \Rightarrow \bigvee_{h \in \mathcal{H}(\phi, \phi')} \psi\left[\rho_h\right](y_1, \ldots, y_k)\right) \tag{11.1}$$

Proof. We show both directions:

$\Leftarrow$) Given c in $[\![\phi']\!]$, we show c in $[\![\phi]\!]$. Assume $c = u_1, \ldots, u_n$ with $(\psi'\left[\rho_{h'}\right])\left[\rho_c\right]$ valid for some strictly monotonic injection $h' : \overline{m'} \to \overline{n}$. By

assumption, there exists a strictly monotonic injection $h \in \mathcal{H}(\phi, \phi')$ such that $((\psi[\rho_h])[\rho_{h'}])[\rho_c]$ is valid. The function $h \circ h' : \overline{m} \to \overline{n}$ is also a strictly monotonic injection. In other words, the configuration c is in $[\![\phi]\!]$.

$\Rightarrow$) Suppose the implication 11.1 does not hold. We find a configuration c in $[\![\phi']\!]$ that is not in $[\![\phi]\!]$. The fact that the implication 11.1 does not hold entails the existence of at least a mapping $\hbar$ associating values to each variable in $y_1, \ldots, y_k$, and for which ψ' holds but not $\bigvee_{h \in \mathcal{H}(\phi,\phi')} \psi[\rho_h](y_1, \ldots, y_k)$. Reformulate such a mapping into the configuration $c = u_1, \ldots, u_m$ of size m. By construction, $(\psi[\rho_h])[\rho_c]$ does not hold for any injection $h \in \mathcal{H}(\phi, \phi')$. The configuration c is therefore outside $[\![\phi]\!]$.

$\square$

Termination

We finish this section by explaining why termination of the analysis (on the approximate system) is guaranteed for linear parameterized systems with finite-processes (i.e. no numerical local variables and $X = X^{\mathcal{B}}$). For such systems , the constraint system is *well quasi-ordered (WQO)*[6] with respect to $\sqsubseteq$. $(A, \preceq)$ is obviously a *WQO* for any finite set A and any *quasi-order* $\preceq$ on A. Let A^* be the set of words over A, and $\preceq^*$ be the sub-word relation. Higman's lemma [Hig52] states that $(A^*, \preceq^*)$ is also a WQO. Take A to be the finite set of quotient-sets of formulas in $\mathbb{B}(X \cup Q)$ under the usual equivalence relation. Let $\preceq$ be the implication relation on formulas in $\mathbb{B}(Y)$. By lemma 11.1, the relation $\sqsubseteq$ coincides with $\preceq^*$. We conclude that the constraint system is a WQO if the set $X^{\mathcal{N}}$ is empty. If $X^{\mathcal{N}}$ is not empty in $\mathcal{P} = (Q, X, T)$, one can build a sequence $c_1, c_2, \ldots$ of configurations unrelated by the ordering of section 11.2. Such a sequence of configurations would, for example, have $i+1$ processes in each configuration c_i, with each process sharing the value of its (possibly unique) numerical variable with the process to its right (if any). It is important that there is an unbounded number of values.

11.5 Additional Features

We add a number of features to the model introduced in the beginning of the current chapter. These features are modeled by generalizing the conditions allowed in the transitions. For all the new features, we can use the same constraint system as the one introduced in section 11.4; consequently checking entailment and intersection with initial configurations need not be modified.

Binary and N-ary Communication

In *binary communication* two processes perform a *rendez-vous*, changing states simultaneously. Such a transition can be encoded by considering a form

[6] Recall that for a set A and a quasi-order $\preceq$, the tuple $(A, \preceq)$ is a *well quasi-ordering* if for each infinite sequence $a_1, a_2, \ldots$ of elements of A, there exists $i < j$ with $a_i \preceq a_j$.

of existential global conditions that is more general than the one allowed in the model of section 11.1. More precisely we allow θ_1 in the definition of an existential global condition to constrain variables in the set $o \cdot Y^{next}$. A binary communication rule is in the set $\mathbb{F}(X \cup \mathsf{o} \cdot Y \cup X^{next} \cup \mathsf{o} \cdot Y^{next})$. Here, variables in X, X^{next} and $o \cdot Y$, $o \cdot Y^{next}$ represent the two processes involved in the rendez-vous. We can naturally generalize this scheme by introducing the use of existentially quantified conditions in the set $\mathbb{F}(X \cup X^{next} \cup \bigcup_{i=1}^{n} (o \cdot Y_i \cup o \cdot Y_i^{next}))$. This kind of quantification can be used to specify synchronization between $n+1$ distinct processes.

Shared Variables

We assume the presence of a finite set S of Boolean and numerical *shared variables* that can be read and written by all processes in the system. A transition may both check and modify S together with the local variables of the processes. Shared variables can be modeled as special processes. The updating of the value of a shared variable by a process can be modeled as a rendez-vous between the process and the variable.

Broadcast

A *broadcast* transition is initiated by a process, called the *initiator*. Together with the initiator, each other process inside the system responds simultaneously changing its local state and variables. We can model broadcast transitions by generalizing universally quantified conditions. The generalization is similar to the case of binary communication, i.e., we allow variables in $o \cdot Y^{next}$ to occur in the quantified formula. Such a formula is hence in $\mathbb{F}(X \cup \mathsf{o} \cdot Y \cup X^{next} \cup \mathsf{o} \cdot Y^{next})$.

Dynamic Creation and Deletion

We allow dynamic creation and deletion of processes. A *process creation* rule allows creation of a new process whose local state is specified and variables constrained. The newly created processes may be placed anywhere inside the array of processes. We also allow *process deletion* rules. Such a rule allows deletion of single processes with specified local states and constrained values on variables. For instance, the transition rule

$$\left[\ \cdot \rightarrow idle \ \rhd \ \neg(flag^{next})\ \right]$$

states that a process may be created with an initial state *idle* and a $flag$ value initialized to *false*. In a similar manner the transition

$$\left[\ stop \rightarrow \ \cdot \ \ \rhd \ flag\ \right]$$

is used to state that a process in state *stop* with its flag value set to $true$ can be deleted. The predecessor computations associated with these operations amount to projecting away (for creation), and inserting (at any position in the

array) a copy of Y constrained by the condition appearing in the transition (for deletion).

11.6 Conclusions

We have presented a method for approximate reachability analysis of systems which consist of an arbitrary number of processes organized in a linear array. Each process in the system manipulates a number of Boolean (finite) or numerical variables. Processes may communicate via global transitions, broadcast, rendez-vous, and shared variables. Chapter 13 exhibits several parameterized systems which may be described by the class defined above. The method introduced in this chapter has allowed to (fully) automatically check safety properties of these systems.

In the next chapter, we adapt the approach to the case of linear parameterized systems where components may manipulate (Boolean and numerical) local array variables.

12. Extension to Components with Array Variables

In this chapter, we extend the model introduced in chapter 11. In the following, we refer to the latter model as the *basic model*. The model we introduce here[1] can be enriched by the same additional features as in section 11.5. We briefly describe in section 12.5 how to use the model of this chapter to describe distributed parameterized systems.

Each component is again an extended automaton, but now using two sets of variables X and R. More specifically, the set X denotes the set of simple local variables and is similar to the local variables in the basic model.Variables in this set may range over the Booleans or the natural numbers. In addition, each component associates to each other component in the parameterized system a copy of the set of variables R. These are referred to as array local variables. Concretely, in an instance with n processes, each process associates $n-1$ copies of R, one for each other process in the instance. In other words, processes in the system are augmented with arrays that are indexed by the positions of the other processes in the system. Observe that, in an instance with n processes, there are n copies of the set X and $n(n-1)$ copies of the set R.

The transitions of the automaton are conditioned by the values of the simple local variables and by *global* conditions in which the process checks, together with its own array variables, the local states and simple variables, of all or some of the other processes inside the system. A transition may change the value of any local variable inside the process (possibly deriving the new values from those of the other processes). The aim is again to verify correctness of the systems for the whole family (regardless of the number n of processes inside the system).

Formally, a *parameterized system* $\mathcal{P}$ is a triple (Q, X, R, T), where the sets Q, X, R and T are finite and respectively correspond to the sets of *local states*, *simple local* variables, *array local* variables, and *transition* rules. The set X is partitioned into the sets $X^{\mathcal{B}}$ and $X^{\mathcal{N}}$, while R is partitioned into the sets $R^{\mathcal{B}}$ and $R^{\mathcal{N}}$. As usual, variables in $X^{\mathcal{B}}$ and $R^{\mathcal{B}}$ take their values in $\mathcal{B}$, while those in $X^{\mathcal{N}}$ and $R^{\mathcal{N}}$ take their values in $\mathcal{N}$.

A transition rule t is of the same form as in figure 11.1; with $\underline{q}, \underline{q} \in Q$ and θ either *local* or *global*. Intuitively, a process (*current*) takes the transition and changes its local state from $\underline{q}$ to $\underline{q}'$. In the meantime, the values of the simple

[1] third and last class of the linear parameterized systems introduced in chapter 9.

and array local variables of the process are updated according to θ. Below, we describe how we define local and global conditions.

We recall $Y = X \cup Q$, and use the sets Y^{next} and R^{next} to mean the sets of the *next-value* versions of (respectively) the variables in Y and R. A *local condition* is a formula in $\mathbb{F}(X \cup X^{next})$. The formula specifies (in a similar manner to 11.1) how local variables of the current process are updated with respect to their current values.

Global conditions check the values of simple local variables of the current process, local states and values of simple local variables of the other processes, together with the array local variables associated to them. We use, for each $x \in Y$, the variable $o{\cdot}x$ to refer to the simple local variable owned by another process. We also use, for each array variable r in R, the variable $o{\cdot}r$ to refer to the array local variable owned by the current process and associated to another process (pointed by $o{\cdot}$). We define the sets $o{\cdot}R = \{o{\cdot}r | r \in R\}$ and $o{\cdot}R^{next} = \{o{\cdot}r^{next} | r \in R\}$. Furthermore, a *global condition* is of the form $\Box\theta_1$ where $\Box$ is a *universal* or an *existential* quantifier, and $\theta_1 \in \mathbb{F}(X \cup X^{next} \cup o{\cdot}(R \cup R^{next} \cup Y))$. Recall that a universal quantifier is in $\{\forall_{LR}, \forall_L, \forall_R\}$; while an existential quantifier in $\{\exists_L, \exists_R, \exists_{LR}\}$. A global condition is said to be *universal* or *existential* depending on the type of the quantifier appearing in it.

As an example, assume $X^{\mathcal{N}} = \{count\}$ and $R^{\mathcal{B}} = \{check\}$ and consider the two following formulas:

$$\exists_L \ ((o{\cdot}count < count) \ \wedge \ o{\cdot}check^{next} \ \wedge \ count^{next} = count)$$
$$\forall_L \ (o{\cdot}check \ \wedge \ \neg o{\cdot}check^{next} \ \wedge \ count^{next} = count)$$

The first condition states that there is another process to the left (of the current process) with a simple numerical local variable $o{\cdot}count$ strictly smaller than the local variable *count* belonging to the current process. The entry (in the array local to the current process) corresponding to the other process (pointed by $o{\cdot}$) is then set to *true*. The local variable *count* is copied. The second formula states that the transition can be performed if the entries in the local array of the current process, and corresponding to all processes to the left, have their array variable *check* set to *true*. In such a case, these entries are reset to $false$, and the local variable *count* is copied. We explain the semantics of such transitions in the following section.

Outline

Section 12.1 defines the induced transition system. Section 12.2 generalizes the ordering on configurations of section 12.2. Section 12.3 proposes an over-approximated transition system that is monotonic with respect to the defined ordering. We show in section 12.4 that the approximate transition system satisfies the requirements described in chapter 9. Finally, we introduce, in section 12.5, how to extend the model of this chapter in order to describe distributed systems. We conclude this chapter in section 14.

12.1 Extended Transition System

In this section we describe the transition system $(C, \longrightarrow)$ induced by a parameterized system $\mathcal{P} = (Q, X, R, T)$. In an instance of $\mathcal{P}$, a configuration in C captures (for each component) the local states, the values of simple local variables, and the values of array local variables (corresponding to each other process in the instance). We define *simple local states* and *process states* in a similar manner to section 11.1. A simple local variable state is a mapping $v : X \mapsto \mathcal{B} \cup \mathcal{N}$ that respects the types of the variables. A process state $u = (q, v)$ is also a mapping $u : Y \mapsto \mathcal{B} \cup \mathcal{N}$, where $u(x) = v(x)$ for each $x \in X$, $u(q) = true$, and $u(q') = false$ for each $q' \in Q / \{q\}$. In a similar manner, we define an *array local state* w to be a mapping $R \to \mathcal{B} \cup \mathcal{N}$ which respects the types of the variables in R.

A *configuration* c is a tuple $\left([u_1, \ldots, u_n], \left[w_{1[2]}, \ldots, w_{n[n-1]}\right]\right)$ where $[u_1, \ldots, u_n]$ is a sequence of process states and $\left[w_{1[2]}, \ldots, w_{n[n-1]}\right]$ is a sequence of array local states. Intuitively, the above configuration corresponds to an instance of the system with n components. Here, the process state u_i captures the state and the values of the simple local variables of the component i in $\mathcal{P}$; while $w_{i[j]}$ captures the values of the array variables associated to component j in the local array owned by component i.

We define the transition relation $\longrightarrow$ on the set of configurations. Following the approach in section 11.1, we describe the semantics of local conditions in terms of two simple local states v and v'. Recall that a local condition θ is in $\mathbb{F}(X \cup X^{next})$. We write $(v, v') \models \theta$ to denote the validity of the formula $\theta\left[\rho^{loc}\right]$ where the substitution ρ^{loc} is union of $\{x \leftarrow v(x) |\, x \in X\}$ and $\{x^{next} \leftarrow v'(x) |\, x \in X\}$. For process states $u = (q, v)$ and $u' = (q', v')$, we again use $(u, u') \models \theta$ to denote that $(v, v') \models \theta$.

The semantics of global conditions are described as follows. Assume a formula θ in the set $\mathbb{F}(X \cup X^{next} \cup o \cdot (Y \cup R \cup R^{next}))$ representing a global condition. The semantics is given in terms of five mappings. Two simple local states (v, v'), two array local states (w, w') and one process state (u_j). The roles of v and v' are the same as for local conditions. While evaluating a global condition, a component i may consider the state and values of array variables associated with anther process j, together with values of simple local variables owned by process j. Both simple local variables and array local variables owned by the process evaluating the global condition (i.e. current) may get their values changed. We use u_j to refer to the process state of the other component. We use w and w', to respectively refer to the current and next values of the arrays variables corresponding to the considered component. We write $(v, v', w, w', u_j) \models \theta$ to denote the validity of the formula $\theta\left[\rho^{glob}\right]$ where the substitution ρ^{glob} is defined as the union of five sets of assignments: $\{x \leftarrow v(x) |\, x \in X\}$, $\{x^{next} \leftarrow v'(x) |\, x \in X\}$, $\{o \cdot r \leftarrow v(r) |\, r \in R\}$, $\{o \cdot r^{next} \leftarrow v'(r) |\, r \in R\}$ and $\{o \cdot x \leftarrow u_1(x) |\, x \in Y\}$. The

relation $(u,u',w,w',u_j) \models \theta$ is interpreted in a similar manner to the case of local conditions.

Now, we are ready to define the transition relation $\longrightarrow$. Let t be a transition rule of the form depicted in figure 11.1. Consider two configurations c, c' of the same size n. We write $c = \left(u_1,\ldots,u_n,\left[w_{1[2]},\ldots,w_{n[n-1]}\right]\right)$ and $c' = \left([u'_1,\ldots,u'_n],\left[w'_{1[2]},\ldots,w'_{n[n-1]}\right]\right)$ with $u_k = (q_k,v_k)$ and $u'_k = \left(q'_k,v'_k\right)$ for each $k : 1 \le k \le n$. We write $c \xrightarrow{t,i} c'$, for some $i : 1 \le i \le n$, to mean the following. States q_i and q'_i coincide with $\underline{q}$ and $\underline{q}'$ respectively. Furthermore, one of the following conditions is met:

1. θ is a **local** condition and $(u_i,u'_i) \models \theta$ while $u_{k_1} = u'_{k_1}$, and $w_{k_2[k_3]} = w'_{k_2[k_3]}$ for all $k_1,k_2,k_3 : 1 \le k_1 \ne k_2 \le n$ with $k_1 \ne i$.
2. θ is a **universal** global condition and both following conditions are satisfied. First, $(v_i,v'_i,w_{i[j]},w'_{i[j]},u_j) \models \theta$ **for each** $j : 1 \le j < i$ if $\Box \in \{\forall_L,\forall_{LR}\}$, **and each** $j : i < j \le n$ if $\Box \in \{\forall_{LR},\forall_R\}$. Second, $u_{k_1} = u'_{k_1}$, $w_{k_1[i]} = w'_{k_1[i]}$, $w_{k_1[k_2]} = w'_{k_1[k_2]}$, and for each $k_1,k_2 : 1 \le k_1,k_2 \le n$ such that k_1,k_2 and i are mutually different; and also $w_{i[k_3]} = w'_{i[k_3]}$, for each $k_3 : 1 \le k_3 < i$ if $\Box = \forall_R$, and each $k_3 : i < k_3 \le n$ if $\Box = \forall_L$.
3. θ is an **existential** global condition and both of the following conditions are satisfied. First, $(v_i,v'_i,w_{i[j]},w'_{i[j]},u_j) \models \theta$ **for some** $j : 1 \le j < i$ if $\Box \in \{\exists_L,\exists_{LR}\}$, **or some** $j : i < j \le n$ if $\Box \in \{\exists_R,\exists_{LR}\}$. Second, $u_{k_1} = u'_{k_1}$, $w_{k_1[i]} = w'_{k_1[i]}$, $w_{i[k_3]} = w'_{i[k_3]}$ and $w_{k_1[k_2]} = w'_{k_1[k_2]}$ for each $k_1,k_2,k_3 : 1 \le k_1,k_2,k_3 \le n$ such that i,k_1,k_2,k_3 are mutually different, with $k_3 \ne j$.

We use $c \xrightarrow{t} c'$ to denote that $c \xrightarrow{t,i} c'$ for some $i : 1 \le i \le n$, and $c \longrightarrow c'$ to denote that $c \xrightarrow{t} c'$ for some $t \in T$.

In a similar manner to 11.1, we assume (for a $\mathcal{P} = (Q,X,R,T)$), that prior to starting the execution of the system, each process is in an (identical) *initial* process state $u_{init} = (q_{init},v_{init})$, and each element of each array is in an (identical) *initial* array state r_{init}. The set I of initial configurations contains configurations of the form $([u_{init},\ldots,u_{init}],[w_{init},\ldots,w_{init}])$ with n copies of u_{init} and $n(n-1)$ copies of w_{init} (n is some natural number). This set is infinite.

12.2 Ordering on Configurations

We extend in the following the ordering on configurations introduced in section 12.2. Consider two configurations, $c = \left([u_1,\ldots,u_m],\left[w_{1[2]},\ldots,w_{m[m-1]}\right]\right)$ and $c' = \left([u'_1,\ldots,u'_n],\left[w'_{1[2]},\ldots,w'_{n[n-1]}\right]\right)$, where $u_i = (q_i,v_i)$ for each $i : 1 \le i \le m$, and $u'_i = (q'_i,v'_i)$ for each $i : 1 \le i \le n$. We write $c \preceq c'$ to denote that there exists a strictly monotonic injection $h : \overline{m} \to \overline{n}$ such that the following four conditions are satisfied for each (mutually different) $i_1,j_1,i_2,j_2 : 1 \le i_1,j_1,i_2,j_2 \le m$.

1. $q_{i_1} = q'_{h(i_1)}$.

2. $v_{i_1}(x)$ iff $v'_{h(i_1)}(x)$ for each $x \in X^{\mathcal{B}}$
3. $w_{i_1[j_1]}(r)$ iff $w'_{h(i_1)[h(j_1)]}(r)$ for each $r \in R^{\mathcal{B}}$.
4. For any $x, y \in X^{\mathcal{N}}$, $r, s \in R^{\mathcal{N}}$ and $\sim \in \{<, \leq\}$, each one of the following implications does hold:
   - $v_{i_1}(x) \sim_k v_{i_2}(y)$ entails there exists $l : k \leq l$ such that $v'_{h(i_1)}(x) \sim_l v'_{h(i_2)}(y)$.
   - $w_{i_1[j_1]}(r) \sim_k v_{i_2}(x)$ entails the existence of an $l : k \leq l$ such that $w'_{h(i_1)[h(j_1)]}(r) \sim_l v'_{h(i_2)}(x)$.
   - $v_{i_1}(x) \sim_k w_{i_2[j_2]}(r)$ entails there exists an $l : k \leq l$ such that $v'_{h(i_1)}(x) \sim_l w'_{h(i_2)[h(j_2)]}(r)$.
   - $w_{i_1[j_1]}(r) \sim_k w_{i_2[j_2]}(s)$ entails there exists an $l : k \leq l$ such that $w'_{h(i_1)[h(j_1)]}(s) \sim_l w'_{h(i_2)[h(j_2)]}(r)$.

In other words, there is a mapping from (from left to right of) the processes in c to the processes in c' such that the following holds. The local states and the values of the Boolean variables coincide in the corresponding processes (Conditions 1, 2 and 3). Concerning numerical variables, the ordering preserves equality of variables, while gaps between variables in c' are at least as large as the gaps between the corresponding variables in c (Condition 4). Observe this ordering coincides with the one defined in section 11.1 in case the set of local variables R in $\mathcal{P}$ is empty.

We derive in the following section an approximate transition systems such that monotonicity (with respect to $\preceq$) is enforced.

12.3 Approximation

Following section 11.3, we introduce an over-approximation $\rightsquigarrow$ of the transition relation $\longrightarrow$. The purpose is again to enforce monotonicity with respect to the ordering $\preceq$ defined on configurations. The set C of configurations in $\mathcal{A}(\mathcal{P}) = (C, \rightsquigarrow)$ is identical to the one in the original transition system $\mathcal{T}(\mathcal{P})$ presented in section 12.1. We only modify the semantics of transitions involving universal global conditions. We define $\rightsquigarrow = (\longrightarrow \cup \rightsquigarrow_1)$, where $\overset{t}{\rightsquigarrow}_1$ reflects the approximation of universal quantifiers. Roughly speaking, in the new semantics, we (again) remove all processes in the configuration which violate the condition of the universal quantifier. Below we describe how this is done.

First, we define what is meant by removing a process from a configuration. For configuration $c = \left([u_1, \ldots, u_n], \left[w_{1[2]}, \ldots, w_{n[n-1]}\right]\right)$ and an index $i : 1 \leq i \leq n$, we define $c \ominus i$ to be the configuration $c' = \left(\left[u'_1, \ldots, u'_{n-1}\right], \left[w'_{1[2]}, \ldots, w'_{n-1[n-2]}\right]\right)$ defined in the following way. Define the strictly monotonic injection $h : \overline{n-1} \to \overline{n}$ such that $h(j) = j$ if $j < i$, and $h(j) = j+1$ otherwise. The configuration c' verifies the following. for each $i, j : 1 \leq i, j \leq n-1$, the process state u'_j and the array local state $w'_{i[j]}$ respectively coincide with $u_{h(j)}$ and $w_{h(i)[h(j)]}$. Intuitively, $c \ominus i$ corresponds to deleting the process i together with the variables associated to it by the other

processes in the instance. We generalize this notion to sets of indices, and write $c \ominus \xi$, where ξ is some set of indices (or linear array positions) to mean the successive application, from the largest to the smallest, for each element i in ξ, of the deletion of process i.

Assume the transition t depicted in figure 11.1, and where θ is a universal global condition of the form $\Box\theta_1$ with $\Box$ in the set $\{\forall_L, \forall_R, \forall_{LR}\}$. Let c and $\tilde{c}$ (respectively) correspond to both configurations $\left([u_1, \dots, u_m], \left[w_{1[2]}, \dots, w_{m[m-1]}\right]\right)$ and $\left([\tilde{u}_1, \dots, \tilde{u}'_n], \left[\tilde{w}_{1[2]}, \dots, \tilde{w}_{n[n-1]}\right]\right)$. We write $c \overset{t}{\rightsquigarrow}_1 \tilde{c}$ to denote the existence of some $i : 1 \leq i \leq m$, and of a configuration $c' = \left([u'_1, \dots, u'_n], \left[w'_{1[2]}, \dots, w'_{n[n-1]}\right]\right)$ such that the following holds:

1. $q_i = \underline{q}$ and $q'_i = \underline{q}'$,
2. $j \in \xi$ iff $(u_i, u'_i, w_{i[j]}, w'_{i[j]}, u_j) \not\models \theta$ and either:
   - $\Box \in \{\forall_L, \forall_{LR}\}$ and $j : 1 \leq j < i$, or
   - $\Box \in \{\forall_R, \forall_{LR}\}$ and $j : i < j \leq n$
3. $\tilde{c} = c' \ominus \xi$.

In other words, we have $c \overset{t}{\rightsquigarrow}_1 \tilde{c}$ if we can obtain $\tilde{c}$ from another configuration c' as follows. There is a process i at state $\underline{q}$ in c, and at state $\underline{q}'$ in c' (Conditions 1), and $\tilde{c}$ is obtained by deleting in c' each process j (together with the entries corresponding to it in the arrays of the other processes) that violates the global condition depending on the quantifier $\Box$ (that is $j < i$ if $\Box$ in $\{\forall_L, \forall_{LR}\}$ and $i < j$ if $\Box$ in $\{\forall_R, \forall_{LR}\}$) as expressed by Condition 2, and Condition 3 above.

Lemma 12.1. *The approximate transition system $(C, \rightsquigarrow)$ is monotonic with respect to $\preceq$.*

Proof. Similar to the proof of lemma 12.1. The main difference consists in taking into account all the local array variables in addition to the process states and to the simple local variables. $\Box$

12.4 Constraints and Operations

In this section, we adapt the constraints introduced in section 11.4 to take into account the array variables of the processes. We also specify how to perform the operations necessary for the application of the scheme in chapter 10. We will not exhibit the proofs of the corresponding lemmas since they respectively identical (except for the manipulation of the array variables) to those in section 11.4. For the rest of this section, we fix a parameterized system $\mathcal{P} = (Q, X, R, T)$ and the induced approximate transition system $\mathcal{A}(\mathcal{P}) = (C, \rightsquigarrow)$.

Constraints

Recall that for each natural number $i \in \mathcal{N}$ we make a copy Y_i such that $x_i \in Y_i$ if $x \in Y$. We also make, for each $j \neq i$, a copy R_{ij} of R such that $r_{ij} \in R_{ij}$ if $r \in R$. A *constraint* ϕ is a pair (m, ψ), where $m \in \mathcal{N}$ is a natural number, and $\psi \in \mathbb{F}(\cup_{1 \leq i \leq n} Y_i \cup \cup_{1 \leq i \neq j \leq n} R_{ij})$. Intuitively, a configuration satisfying ϕ should contain at least m processes (indexed by $1, \ldots, m$). The constraint ϕ uses the elements of the set Y_i to refer to the local states and variables of process i. In the same way, ϕ uses the elements of the set R_{ij} to refer to the array variables owned by process i and associated with process j. The values of these states and variables are constrained by the formula ψ. Formally, consider a configuration $c = \left([u_1, \ldots, u_n], \left[w_{1[2]}, \ldots, w_{n[n-1]}\right]\right)$ and a constraint $\phi = (m, \psi)$. Let $h : \overline{m} \mapsto \overline{n}$ be a strictly monotonic injection. We write $c \models_h \phi$ to denote the validity of the formula $\psi[\rho]$ where ρ is union of $\left\{x_i \leftarrow u_{h(i)}(x) |\, x \in Y \text{ and } 1 \leq i \leq m\right\}$, and $\left\{r_{ij} \leftarrow w_{h(i)[h(j)]}(r) |\, r \in R \text{ and } 1 \leq i \neq j \leq m\right\}$. We write $c \models \phi$ to denote that $c \models_h \phi$ for some h; and define $[\![\phi]\!] = \{c |\, c \models \phi\}$. For a (finite) set of constraints Φ, we define $[\![\Phi]\!] = \bigcup_{\phi \in \Phi} [\![\phi]\!]$. The following lemma states, that the set of configurations denoted by a constraint is upward closed.

Lemma 12.2. *For each constraint ϕ, the set $[\![\phi]\!]$ is upward closed.*

Proof. Similar to the proof of lemma 12.2. □

In all the examples we consider, the set F of bad configurations (chapter 10) can (here also) be represented by a finite set Φ_F of constraints. The coverability question can then be answered by checking whether $I \xrightarrow{*} [\![\Phi_F]\!]$. The three operations on constraints used in the scheme of chapter 10, namely computing *Pre*, entailment, and intersection with initial states are performed in a similar manner to section 12.4. Below, we show how to compute *Pre*.

Computing Pre

We describe the computation of $Pre(\phi')$ as a finite set of constraints. Assume a constraint $\phi' = (m, \psi')$ where $\psi' \in \mathbb{F}(\cup_{1 \leq i \leq n} Y_i \cup \cup_{1 \leq i \neq j \leq n} R_{i[j]})$. Recall $Pre(\phi')$ is defined as the set $\bigcup_{t \in T} Pre_t(\phi')$, where $[\![Pre_t(\phi')]\!] = \left\{c |\, \exists . c' \in [\![\phi']\!].\, c \overset{t}{\rightsquigarrow} c'\right\}$. The main difference compared to section 12.4 is that we have to take into account the array variables owned by the process taking a global transition.

We extend the operator $\oplus$ (introduced in section 12.4) to the case of constraint formulas in $\mathbb{F}(\cup_{1 \leq i \leq n} Y_i \cup \cup_{1 \leq i \neq j \leq n} R_{i[j]})$. Fix such a formula ψ, and some $i : 1 \leq i \leq n+1$. Define the strictly monotonic injection $h_i : \overline{n-1} \rightarrow \overline{n}$ such that $h_i(j) = j$ if $j < i$, and $h_i(j) = j+1$ otherwise. We write $\psi \oplus i$ to mean the formula in $\mathbb{F}(\cup_{1 \leq i \leq n+1} Y_i \cup \cup_{1 \leq i \neq j \leq n+1} R_{i[j]})$ such that $\psi \oplus i = \psi\left[\bigcup_{j : 1 \leq j \leq n} \left\{x_j \leftarrow x_{h_i(j)} |\, x \in Y\right\}\right]\left[\bigcup_{j : 1 \leq j,k \leq n} \left\{r_{jk} \leftarrow w_{h_i(j)[h_i(k)]} |\, r \in R\right\}\right]$. Intuitively, $\psi \oplus i$ corresponds to the insertion in ψ of a (free) set of variables. More concretely, one copy of Y and $2n$ copies of R at positions defined by the strictly monotonic injection h_i above. Again, $\psi \oplus i$ corresponds (like

for section 12.4) to the insertion of a new process (i.e. with unconstrained variables and states) at position i. For two indices $i, j : 1 \leq i < j \leq n+2$, we write $\psi \oplus i, j$ to mean the formula $((\psi \oplus i) \oplus j)$.

Back to our *Pre* computation, we recall the local condition case is similar to the corresponding case in the basic model. Let t be the transition rule of figure 11.1 where θ is a global condition of the form $\Box\theta_1$, with θ_1 a condition in $\mathbb{F}(X \cup X^{next} \cup o\cdot(Y \cup R \cup R^{next}))$. We recall the definition of the formula $\eta_{\underline{q},\underline{q}'}(q, q')$ in chapter 11.

Fix a natural $m : 1 \leq m$; for each $i, j : 1 \leq i, j \leq m$, the five substitutions ρ_1^i, ρ_2^j, ρ_3^i, ρ_4^{ij} and ρ_5^{ij} follow the scheme (except for array variables) of the substitutions used in the *Pre* computations in section 11.4. Concretely, ρ_1^i is defined by the set $\{x \leftarrow x_i |\, x \in Y\} \,\cup\, \{x^{next} \leftarrow x_i^{next} |\, x \in Y\}$; substituiton ρ_2^j by the set $\left\{x \leftarrow x_j |\, x \in Y\right\}$; substitution ρ_3^i by the set $\{x_i \leftarrow x_i^{next} |\, x \in Y\}$; substitution ρ_4^{ij} by the set $\left\{r \leftarrow r_{ij} |\, r \in R\right\} \,\cup\, \left\{r^{next} \leftarrow r_{ij}^{next} |\, r \in R\right\}$; and finally substitution ρ_5^{ij} by the set $\left\{r_{ij} \leftarrow r_{ij}^{next} |\, r \in R\right\}$. We use these substitutions in the following.

Existential Conditions.

We look at the case where $\Box = \exists_L$; the $\Box = \exists_{LR}$ and $\Box = \exists_R$ cases are similar. We use $i, \underline{i}$ and $j, \underline{j}$ to respectively denote the indices (in the constraint ϕ') of the process taking the transition, and of the witness process[2]. In a similar manner to the existential case section 11.4, the definition of $Pre_t(\phi')$ consists in four cases corresponding to the situations where the processes i and j are: both represented in ϕ'; or only process i is represented in ϕ' (let $\underline{j}$ be); or only the process j is represented in ϕ', or neither of them is represented in ϕ'. More precisely, we define $Pre_t(\phi')$ to be the smallest set containing the following constraints:

- A constraint (m, ψ_{ij}) for each $i, j : 1 \leq j < i \leq m$ and corresponding to the case where both processes are part of the definition of the constraint ϕ'. Each formula ψ_{ij} is given by:

$$\exists\left(Y_i^{next} \cup R_{i[j]}^{next}\right).\left(\left(\eta_{\underline{q},\underline{q}'}(q,q') \wedge \theta_1\left[\rho_2^j\right]\left[\rho_4^{ij}\right]\right)\left[\rho_1^i\right] \wedge \psi'\left[\rho_3^i\right]\left[\rho_5^{ij}\right]\right)$$

- A constraint $\left(m+1, \psi_{i\underline{j}}\right)$ for each $i, \underline{j} : 1 \leq \underline{j} < i \leq m+1$ and corresponding to the case where only the process taking the transition is part of the definition of the constraint ϕ'. Each formula $\psi_{i\underline{j}}$ is given by:

$$\exists\left(Y_i^{next} \cup R_{i[\underline{j}]}^{next}\right).\left(\left(\eta_{\underline{q},\underline{q}'}(q,q') \wedge \theta_1\left[\rho_2^{\underline{j}}\right]\left[\rho_4^{i\underline{j}}\right]\right)\left[\rho_1^i\right] \wedge (\psi' \oplus \underline{j})\left[\rho_3^i\right]\left[\rho_5^{i\underline{j}}\right]\right)$$

[2] Intuitively, Let i_h and j_h respectively denote (in the configuration $c' \models_h \phi'$) the indices of the process taking the transition, and of the witness process. We will write i to mean $h^{-1}(i_h)$ if i_h in image of h. Otherwise, we write $\underline{i}$ to mean $pos_h(i_h)$ as defined in sections 11.4 (and j to mean $h^{-1}(j_h)$ if j_h in image of h. Otherwise, we write $\underline{j}$ to mean $pos_h(j_h)$.)

- A constraint $(m+1,\psi_{\underline{i}j})$ for each $\underline{i},j : 1 \leq j < \underline{i} \leq m+1$ and corresponding to the case where only the witness process is represented in the constraint ϕ'. Each formula $\psi_{\underline{i}j}$ is given by:

$$\exists\left(Y_{\underline{i}}^{next} \cup R_{\underline{i}[j]}^{next}\right).\left(\left(\eta_{\underline{q},\underline{q}'}(q,q') \wedge \theta_1\left[\rho_2^j\right]\left[\rho_4^{\underline{i}j}\right]\right)\left[\rho_1^{\underline{i}}\right] \wedge (\psi' \oplus \underline{i})\left[\rho_3^{\underline{i}}\right]\left[\rho_5^{\underline{i}j}\right]\right)$$

- A constraint $\left(m+2,\psi_{\underline{i}\underline{j}}\right)$ for each $\underline{i},\underline{j} : 1 \leq \underline{j} < \underline{i} \leq m+2$ and corresponding to the case where neither the process taking the transition, nor the witness process are represented in the constraint ϕ'. Each formula $\psi_{\underline{i}\underline{j}}$ is given by:

$$\exists\left(Y_{\underline{i}}^{next} \cup R_{\underline{i}[\underline{j}]}^{next}\right).\left(\left(\eta_{\underline{q},\underline{q}'}(q,q') \wedge \theta_1\left[\rho_2^{\underline{j}}\right]\left[\rho_4^{\underline{i}\underline{j}}\right]\right)\left[\rho_1^{\underline{i}}\right] \wedge (\psi' \oplus \underline{j},\underline{i})\left[\rho_3^{\underline{i}}\right]\left[\rho_5^{\underline{i}\underline{j}}\right]\right)$$

Universal Conditions.

We look at the case where $\square = \forall_L$; the $\square = \forall_{LR}$ and $\square = \forall_R$ cases are similar. Two cases follow depending on whether the process taking the transition is represented in ϕ'. More precisely, we define $Pre_t(\phi')$ to be the smallest set containing the following constraints:

- A constraint $\phi_i = (m,\psi_i)$ for each $i : 1 \leq i \leq m$ and corresponding to the case where the process taking the transition is part of the constraint ϕ'. Each formula ψ_i is given by:

  $$\exists\left(Y_i^{next} \cup R_{i[j]}^{next}\right).\Gamma_t \wedge \Gamma_{\psi'}$$

  where the formulas $\Gamma_t, \Gamma_{\psi'}$ above are defined by:

  $$\Gamma_t = \left(\eta_{\underline{q},\underline{q}'}(q,q') \wedge \bigwedge\nolimits_{1\leq j<i\leq m} \theta_1\left[\rho_2^j\right]\right)\left[\rho_4^{ij}\right]\left[\rho_1^i\right]$$
  $$\Gamma_{\psi'} = \psi'\left[\rho_3^i\right]\left[\bigcup\nolimits_{1\leq j<i\leq m} \rho_5^{ij}\right]$$

- A constraint $\phi_{\underline{i}} = (m+1,\psi_{\underline{i}})$ for each $\underline{i} : 1 \leq \underline{i} \leq m$ and corresponding to the case where the process taking the transition is not part of the constraint ϕ'. Each formula $\psi_{\underline{i}}$ is given by:

  $$\exists\left(Y_{\underline{i}}^{next} \cup R_{\underline{i}[j]}^{next}\right).\Gamma_t \wedge \Gamma_{\psi'}$$

  where the formulas $\Gamma_t, \Gamma_{\psi'}$ above are defined by:

  $$\Gamma_t = \left(\eta_{\underline{q},\underline{q}'}(q,q') \wedge \bigwedge\nolimits_{1\leq j<\underline{i}\leq m+1} \theta_1\left[\rho_2^j\right]\right)\left[\rho_4^{\underline{i}j}\right]\left[\rho_1^{\underline{i}}\right]$$
  $$\Gamma_{\psi'} = (\psi' \oplus \underline{i})\left[\rho_3^{\underline{i}}\right]\left[\bigcup\nolimits_{1\leq j<\underline{i}\leq m+1} \rho_5^{\underline{i}j}\right]$$

Lemma 12.3. *We have* $[\![Pre_t(\phi')]\!] = \left\{c \mid \exists.c' \in [\![\phi']\!].\ c \overset{t}{\rightsquigarrow} c'\right\}$ *for any transition t with a local condition, a global existential condition, or a global universal condition.*

Proof. Similar to the proofs of lemma 11.6, lemma 11.7, and lemma 11.8. □

In the following section, we enrich the model of section 12.1.

12.5 Towards Distribution

It is straightforward (considering the similar constraint systems) to adapt to the model of section 12.1, all the additional features described in section 11.5. In this section we introduce features we use later to handle parameterized distributed algorithms. These are protocols where the processes can only communicate by message passing (since they may be distributed in different physical locations). The distributed algorithms we look at here are those where each component may send and receive messages (to and from each other component in the system) through (bounded) channels with finite or numerical contents. We use these features in the examples of chapter 13 to check safety for a number of challenging distributed parameterized systems.

Channels

An important ingredient in any distributed system is the communication medium. The class of parameterized systems we have introduced in 12 allows the modeling of channels as bounded buffers. We introduce this notion using a typical example.

Suppose we are given a distributed protocol where processes have one local numerical variable "*val*". Say processes may send to each other messages of the form $(message, value)$ where message is in $\{req, ack\}$ and *value* is numerical. We model channels of length one by assuming two extra local array variables $channel_m$ (containing variables ranging over $\{\varepsilon, req, ack\}$) and $channel_v$ (containing variables ranging over $\mathcal{N}$). These will play the role of the medium between pairs of processes.

We can model the operation of sending the message (req, val), using the formula $send\,(o, req, val)$ in $\mathbb{F}\,(X \cup X^{next} \;\cup\; o\cdot(R \;\cup\; R^{next}))$ and defined by:

$$send\,(o, req, val) = \left(\begin{array}{c} o\cdot channel_m = \varepsilon \\ \wedge\; o\cdot channel_m^{next} = req \;\wedge\; o\cdot channel_v^{next} = val \end{array} \right)$$

Typically the formula $send\,(o, m_0, v_0)$, may be used in an existential transition of the form:

$$\left[\; \underline{q} \to \underline{q}' \;\rhd\; \exists_R\,(send\,(o, m_0, v_0)) \;\right]$$

to denote sending the message to some process (here to the right of the sender), or using a universal transition of the form

$$\left[\; \underline{q} \to \underline{q}' \;\rhd\; \forall_{LR}\,(send\,(o, m_0, v_0)) \;\right]$$

to denote a broadcast (here to both processes to the left and to the right of the sender). On the other side of the channels, a receiver needs to be able to "pick" its messages. This is done by allowing variables in two new copies of R (say $s\cdot R\,[o]\,, s\cdot R\,[o]^{next}$) to appear in global conditions. This approach is similar to the one used to add broadcast and binary communications in section 11.5.

Intuitively, a process i may read and modify the content $w_{j[i]}$ in the arrays (cells pointing to i) belonging to process j. Back to our example, a receiver may perform a transition like

$$\left[\ \underline{q} \rightarrow \underline{q}' \ \rhd\ \exists_R \left(get\left(o, m_0, v_0\right)\right)\ \right]$$

where the condition $get\left(o, req, val\right)$, is defined by

$$get\left(o, req, val\right) = \left(\begin{array}{l} \quad s{\cdot}channel\left[o\right]_m = req \\ \wedge\ s{\cdot}channel\left[o\right]_m^{next} = \varepsilon\ \wedge\ val^{next} = s{\cdot}channel\left[o\right]_v \end{array}\right)$$

Channels of longer size can be modeled by more copies of the arrays, one for each channel place (buffer cell).

The "is left" and "is right" array

Consider a global universal condition $\forall_{LR}\theta_1$. Some systems require that the evaluation of the condition considers whether they apply to the processes to the left or to those to the right of the component evaluating the transition.

A typical example is the following. Say each process has one numerical local variable *clock*. Say process i has priority if the value of its *clock* variable is strictly smaller than the value of each *clock* variable belonging to a process to the left, and smaller or equal to each clock value belonging to a process to the right. It is not possible to obtain the same behavior by checking first to (say) the left (with $\forall_L(clock < o{\cdot}clock)$), and then to the right (with $\forall_R(clock \leq o{\cdot}clock)$). The reason is that while checking in the second direction, a process in the first direction (compared to the process evaluating the condition) may alter its *clock* variable. One way to achieve this effect and forbid undesired interleavings is to use a shared global variable like those introduced in section 11.5.

This can be handled in a more natural way by introducing two arrays of Booleans "is right" and "is left" (that we respectively denote by 1_L and 1_R). The previous example can then be written

$$\forall_{LR}\ \left(\left(clock < o{\cdot}clock\right)\ \vee\ \left(\left(clock \leq o{\cdot}clock\right)\ \wedge\ o{\cdot}1_R\right)\right)$$

The values of these arrays can be set during an initialization sequence that each process performs before getting to q_{init}. More concretely, assume two extra states q_{init_0} and q_{init_1} and both transitions

$$t_L : \left[\ q_{init_0} \rightarrow q_{init_1}\ \rhd\ \forall_L \left(o{\cdot}1_L^{next} \wedge \neg o{\cdot}1_R^{next}\right)\ \right]$$

$$t_R : \left[\ q_{init_1} \rightarrow q_{init}\ \rhd\ \forall_R \left(\neg o{\cdot}1_R^{next} \wedge o{\cdot}1_L^{next}\right)\ \right]$$

We assume neither of $1_L, 1_R$ can be altered after q_{init}, or read by other components then their owner.

We use these notions to model some challenging distributed algorithms in section 13.

12.6 Conclusions

We have extended the method of chapter 11 in order to handle linear parameterized systems where each process manipulates a number of arrays of Boolean and numerical variables. Processes may communicate via global transitions, broadcast, rendez-vous, and shared variables. We described how these systems can naturally be used to describe particular distributed systems. These are systems where each component may send and receive messages (to and from each other component in the system) through bounded channels (buffers) with bounded or numerical contents. The following chapter 13 describes several non-trivial examples of parameterized systems using the formalism of this chapter. Chapter 13 also reports on the encouraging results permitting fully automatic model checking of these systems.

13. Examples of Linear Parameterized Systems

In this chapter we discuss the application of our verification method. This method permitted the automatic checking of an important number of parameterized systems. We introduce in the following some of these systems, and divide the presentation in four sections. The first section that follows describes parameterized mutual exclusion algorithms where components manipulate bounded variables (first class in 9). Section 13.2 describes (simplified versions of) parameterized cache coherence protocols, also with bounded variables. Section 13.3 concerns checking two non-trivial formulations of the Bakery algorithm by Lamport. Finally, section 13.4 concerns two known parameterized mutual exclusion algorithms that have been designed for distributed communication. The fully automatic check has terminated on all the examples presented below.

Fix a parameterized system $\mathcal{P}$. We allow conditions obtained as conjunctions and disjunctions of local and global conditions. We use the following assumptions in order to improve the readability of the descriptions. The *next-value* forms (x^{next} with $x \in (Q \cup X \cup R)$) that are not mentioned in a conjunction (of the disjunctive normal form) are assumed to equal their current form (i.e. $x^{next} = x$). For instance, assume we consider formulas in $\mathbb{F}(a, b, a^{next}, b^{next})$, where a, b range over the Booleans. We write $a \lor (b \land a^{next})$ to mean $(a \land a^{next} = a \land b^{next} = b) \lor (b \land a^{next} \land b^{next} = b)$.

Given a Boolean variable q, and a state $\underline{q} \in Q$, we mean by the expression $\text{only}\left(q = \underline{q}\right)$ the formula:

$$\text{only}\left(q = \underline{q}\right) = \left(\left(q = \underline{q}\right) \land \bigwedge_{p \in Q / \{\underline{q}\}} \left(q \neq \underline{q}\right) \right)$$

We start with the description of parameterized systems with finite components.

13.1 Bounded Parameterized Mutex Algorithms

In the parameterized systems that follow, a number of components compete for a shared resource. A typical safety property for these systems is the one of mutual exclusion, i.e. at any given time, there is at most one component

that accesses the shared resource. A process is allowed to access the shared resource when at a state referred to as the *critical section*. The correctness of the mutual exclusion algorithm, with respect to this property, amounts then to ensuring that at most one component at a time is at its critical section state.

13.1.1 Burns Algorithm

Burns mutual exclusion algorithm [LPS93] can be viewed as a parameterized system where each process has a local Boolean variable $flag$. Local states range over $\{q1,\ldots,q7\}$ with $q6$ playing the role of critical section. In this algorithm, a process checks values of $flag$ variables belonging to processes to its right or left (not both). A process interested in accessing the shared resource starts by resetting its $flag$ to false (transition t_1). The process checks twice whether other processes to its left are competing for the critical section by reading their $flag$ values (transitions t_3 and t_6).

Table 13.1: Burns Mutex Algorithm

Burns mutual exclusion algorithm: a component operates on a single Boolean variable and can have seven local states $q1,\ldots,q7$ with $q6$ modeling access to the critical section.

States:	**Local:**
$Q = \{q1,\ldots,q7\}$ originally $q1$	$flag$ is in $\mathcal{B}$ originally $false$

Transition rules:

$t_1 : q1$	$\rightarrow$	$q2$	$\triangleright$	$\neg(flag^{next})$
$t_2 : q2$	$\rightarrow$	$q1$	$\triangleright$	$\exists_L(flag)$
$t_3 : q2$	$\rightarrow$	$q3$	$\triangleright$	$\forall_L(\neg flag)$
$t_4 : q3$	$\rightarrow$	$q4$	$\triangleright$	$(flag^{next})$
$t_5 : q4$	$\rightarrow$	$q1$	$\triangleright$	$\exists_L(flag)$
$t_6 : q4$	$\rightarrow$	$q5$	$\triangleright$	$\forall_L(\neg flag)$
$t_7 : q5$	$\rightarrow$	$q6$	$\triangleright$	$\forall_R(\neg flag)$
$t_8 : q6$	$\rightarrow$	$q7$	$\triangleright$	$\neg(flag^{next})$
$t_9 : q7$	$\rightarrow$	$q1$	$\triangleright$	$true$

Final states:

$\text{only}(q_1 = q6) \wedge \text{only}(q_2 = q6)$

If another process is competing for the resource, the process backs-off to state $q1$ (transitions t_2 and t_5). After the double check, several competing pro-

cesses may still end up waiting at state $q5$. The priority is then given to the process most to the right (transition t_7). After accessing the critical section in $q6$, a process stops blocking the access to the resource by setting its $flag$ to false (transition t_8) and moves back to state $q1$.

Unsafe configurations correspond to at least two processes in their critical section state $q6$.

13.1.2 Dijkstra Algorithm

A process takes its states in $\{q1,\dots,q7\}$ where $q6$ represents the critical section. In the original algorithm [LPS93], each process owns a local variable $flag$ taking its values in $\{1,2,3\}$. The processes also share a $pointer$ variable they can (i) set to there position in the array or (ii) use to read the local $flag$ belonging to the process pointed to by $pointer$. We model the $pointer$ variable by a boolean variable p local to each component. Setting the $pointer$ variable amounts to a broadcast where the initiator sets its p variable to true, and simultaneously resets all p variables belonging to the other processes to false (transition t_3).

Table 13.2: Dijkstra Mutex Algorithm

Dijkstra mutual exclusion algorithm: a component operates on a boolean and a finite variable, and takes its states in $q1,\dots,q7$ with $q6$ modeling the critical section.

States:
$Q = \{q1,\dots,q7\}$ originally $q1$

Local:
p is in $\mathcal{B}$ originally $false$
$flag$ is in $\{0,1,2\}$ originally 0

Transition rules:

$t_1:$	$q1$	$\rightarrow$	$q2$	$\triangleright$	$flag^{next} = 1$
$t_2:$	$q2$	$\rightarrow$	$q3$	$\triangleright$	$\forall_{LR}(\circ \cdot flag = 0 \vee \neg \circ \cdot p)$
$t_3:$	$q3$	$\rightarrow$	$q4$	$\triangleright$	$p^{next} \wedge \forall_{LR}(\neg(o \cdot p^{next}))$
$t_4:$	$q4$	$\rightarrow$	$q5$	$\triangleright$	$flag^{next} = 2$
$t_5:$	$q5$	$\rightarrow$	$q1$	$\triangleright$	$\exists_{LR}(\circ \cdot flag = 2)$
$t_6:$	$q5$	$\rightarrow$	$q6$	$\triangleright$	$\forall_{LR}(\circ \cdot flag \neq 2)$
$t_7:$	$q6$	$\rightarrow$	$q7$	$\triangleright$	$flag^{next} = 0$
$t_8:$	$q7$	$\rightarrow$	$q1$	$\triangleright$	$true$

Final states:

only $(q_1 = q6) \wedge$ only $(q_2 = q6)$

Given a set of processes in their initial states and competing for the critical section. There are roughly two possible scenarios:

- If only one process (call it i) fires t_2 followed by t_3 before any other processes *makes a move*, then the remaining processes will be blocked at $q2$. Process i can then fire the rest of the transitions thus accessing the critical section. The transition t_2 will eventually be enabled for other process once i fires t_7.
- In case some processes fire t_2, then t_3 and t_4 ending in $q5$. Among these processes, the last (say i) that had fired t_3 is the only process with p equal to $true$. Observe now that all processes at $q5$ have the variable $flag$ equal to two. Therefore, among these processes, only the *slowest* process (call it j) can fire t_5 since the rest of the processes would by then have fired t_6 and moved back to $q1$. These are the only processes able to reach $q6$ in the next "round".

Unsafe configurations correspond to at least two processes in their critical section state $q6$.

13.1.3 Szymanski Algorithm

Szymanski mutual exclusion algorithm [Szy90, GZ98] induces a parameterized system with states in $\{q1, \ldots, q7\}$ and three Boolean variables a, s and w. A process is in the critical section whenever its local state is $q7$.

There are three stages a requesting process may pass by: the *doorway* ($q3$), the *waiting room* ($q4$) and the *inner sanctum*($q5$, $q6$ and $q7$). The variable a indicates whether the process is interested in accessing the shared resource, variable s whether it has passed the doorway but is not in the waiting room, and variable w whether it is in the waiting room (after the doorway). A process is in the doorway if it took transition t_2 and is at state $q3$. To get to the doorway, a process needs to take the transition t_1. This transition is enabled only if there are no processes in the doorway or in the inner sanctum. After the doorway, a process competing for the critical section goes to the waiting room ($q4$) if there are other processes in the doorway. By doing so, the process stops blocking transition t_1 for processes interested in the access but which still didn't get to the doorway (i.e. processes at $q0$ and $q1$). If there is no (more) process at the doorway, the process bypasses the waiting room and goes to the inner sanctum. This has two consequences: t_1 is now blocked until all processes in the waiting room and the inner sanctum have accessed and left the critical section; and this enables the transition t_5 for the processes in the waiting room to get to the inner sanctum. Once all the processes get to the inner sanctum, the processes most

to the left are given priority to access the resource. The transition t_1 is blocked until all processes in the the inner sanctum have left the critical section (t_8).

Table 13.3: Szymanski Mutex Algorithm.

Szymanski mutual exclusion algorithm: a component operates on three boolean variable and takes its states in $q0,\dots,q7$ with $q7$ modeling the critical section.

States:	**Local:**
$Q=\{q0,\dots,q7\}$ originally $q1$	a,w,s are *Booleans* all *originally* false

Transition rules:

$t_0 : q0$	$\rightarrow$	$q1$	$\triangleright$	a^{next}
$t_1 : q1$	$\rightarrow$	$q2$	$\triangleright$	$\forall_{LR}(\neg o{\cdot}s)$
$t_2 : q2$	$\rightarrow$	$q3$	$\triangleright$	$s^{next} \wedge w^{next}$
$t_3 : q3$	$\rightarrow$	$q5$	$\triangleright$	$\neg(w^{next}) \wedge \forall_{LR}(\neg o{\cdot}a \vee o{\cdot}w)$
$t_4 : q3$	$\rightarrow$	$q4$	$\triangleright$	$\neg(s^{next}) \wedge \exists_{LR}(o{\cdot}a \wedge \neg o{\cdot}w)$
$t_5 : q4$	$\rightarrow$	$q5$	$\triangleright$	$s^{next} \wedge \neg(w^{next}) \wedge \exists_{LR}(o{\cdot}s \wedge \neg o{\cdot}w)$
$t_6 : q5$	$\rightarrow$	$q6$	$\triangleright$	$\forall_{LR}(\neg o{\cdot}w)$
$t_7 : q6$	$\rightarrow$	$q7$	$\triangleright$	$\forall_{L}(\neg o{\cdot}s)$
$t_8 : q7$	$\rightarrow$	$q0$	$\triangleright$	$\neg(a^{next}) \wedge \neg(s^{next})$

Final states:

$\text{only}(q_1 = q7) \wedge \text{only}(q_2 = q7)$

Unsafe configurations correspond to at least two processes in their critical section state $q7$.

13.1.4 The Java Meta-Locking Algorithm

The concurrent object-oriented programming language Java provides synchronization operations for the access to each *synchronized method* or *synchronized statement*. To ensure fairness and efficiency, every object maintains some *synchronization data* (FIFO queue of the threads requesting the object). The Java meta-locking algorithm is the protocol that controls the access to the synchronization data of every object. The Meta-locking protocol is a distributed algorithm observed by every object and thread. The algorithm ensures mutually exclusive access to the synchronization data of every object. The pattern followed by a synchronized method invoked by a thread is as follows:

- The thread gets the object *meta-lock* if no other thread is accessing the synchronization data (*fast path*), otherwise it waits for a *hand-off*
- The thread manipulates the synchronization data.
- The thread releases the meta-lock if no other thread is waiting (*fast path*), otherwise it hands off the meta-lock to a waiting thread.

We consider the parameterized model of the meta-locking algorithm defined in [RR04]. The model is defined for a single object in which synchronization data have been abstracted away. The model consists of the parallel composition of (i) an object, (ii) a *hand-off* process, and (iii) an arbitrary number of threads.

Each thread has five possible states: $idle$, *own* (the thread owns the meta-lock), *hin* (the thread competes to acquire the meta-lock via a hand-in), *hout* (the thread is ready to hand-off the meta-lock), and *wait* (the thread waits for an acknowledgment to acquire the meta-lock). The hand-off process models the races between acquiring and releasing threads via four possible states. This is modeled with a shared variable $hoff$ for "hand off" that takes values in $\{0,1,2,3\}$. The object has one Boolean control variable that is true exactly if there exists a thread possessing the meta-lock. This is modeled by a shared Boolean variable *ob* for "object busy". The object keeps also track of the number of threads waiting to acquire the meta-lock on the object. We keep track of the number of waiting threads by assigning a process to *unit* for each waiting thread and to *null* for each non waiting thread. This is an abstraction of the FIFO queue contained in the object synchronization data.

Table 13.4: Java Meta-Locking Algorithm.

Parameterized model of the Java Meta-locking algorithm defined for a single object in which synchronization data have been abstracted away. The model consists of the parallel composition of an object, a *hand-off* process, and an arbitrary number of threads.

States:	**Shared:**
$Q = \{idle, own, hin, hout, wait\}$	*busy* is in $\mathcal{B}$ originally $false$
$\cup\{null, unit\}$ originally $idle, null$	$hoff$ is in $\{0,\dots 3\}$ originally 0

Transition rules:

t_1 : *idle*	$\rightarrow$	*own*	$\triangleright$	$\neg busy \wedge busy^{next}$
t_2 : *idle*	$\rightarrow$	*hin*	$\triangleright$	$\exists_{LR}(o{\cdot}null \wedge o{\cdot}unit^{next})$
t_3 : *own*	$\rightarrow$	*idle*	$\triangleright$	$busy \wedge \neg(busy^{next}) \wedge \forall_{LR}(\neg o{\cdot}unit)$
t_4 : *own*	$\rightarrow$	*hout*	$\triangleright$	$busy \wedge \exists_{LR}(o{\cdot}unit \wedge o{\cdot}null^{next})$
t_5 : *hin*	$\rightarrow$	*wait*	$\triangleright$	$hoff = 0 \wedge hoff^{next} = 1$

$t_6 :$ *hout*	$\rightarrow$	*idle*	$\triangleright$	$hoff = 0 \wedge hoff^{next} = 2$
$t_7 :$ *hout*	$\rightarrow$	*idle*	$\triangleright$	$hoff = 1 \wedge hoff^{next} = 3$
$t_8 :$ *hin*	$\rightarrow$	*wait*	$\triangleright$	$hoff = 2 \wedge hoff^{next} = 3$
$t_9 :$ *wait*	$\rightarrow$	*own*	$\triangleright$	$hoff = 3 \wedge hoff^{next} = 0$

Final configurations:

$\text{only}\,(q_1 = own) \wedge \text{only}\,(q_2 = own)$,
$\text{only}\,(q_1 = hout) \wedge \text{only}\,(q_2 = hout)$,
$\text{only}\,(q_1 = own) \wedge \text{only}\,(q_2 = hout)$,
$\text{only}\,(q_1 = hout) \wedge \text{only}\,(q_1 = own)$

Transitions t_1 and t_3 correspond to a thread acquiring the meta-lock when the object is not busy and releasing it when there are no other threads waiting (fast path). Each time a thread wants to acquire the meta-lock while the object is busy, it moves to state *hin* (transition t_2), while (simultaneously) a process in state *null* moves to *unit*. When a thread releases the meta-lock while another thread is waiting (transition t_4), the process in state *unit* moves to state *null*. The transitions t_5, t_6, t_7, t_8 and t_9 model the synchronization between a thread in state *hout* releasing the meta-lock, and a thread in state *hin* that has to wait for an acknowledgment to use the meta-lock.

The unsafe configurations correspond to at least two processes at states *hout* or *own*.

13.2 Bounded Parameterized Coherence Protocols

We introduce in the following simplified versions of several cache coherence protocols as possible examples for parameterized systems with bounded variables (first of the three linear classes). Cache are described here as a finite state machine with one copy of data. The transitions of the machine correspond to *coherence* actions or rules. These are applied as consequences of *read hit* or *miss*, or a *write hit* or *miss*. The coherence actions defined by a cache protocol correspond either to a single cache changing its state, or to several caches simultaneously changing their state.

Follows descriptions [Han93, Del00a, Del00b] of *Illinois*, *Dec Firefly*, and *Futurebus+* coherence protocols. We also describe the the directory based *German* protocol which was first introduced in [PRZ01]. For each protocol, we state final constraints denoting the configurations violating the coherence in terms of coexisting caches in particular states.

13.2.1 Illinois Coherence Protocol

The states of a cache range in this protocol over *invd* (invalid), *dirt* (dirty), *shrd* (shared) and *vlid* (valid). Initially all caches are in state *invd*. These states have the following meanings:

- invalid: cache content is not up-to-date.
- dirty: cache contains a modified copy of the data, i.e the data in main memory is obsolete and the other caches are invalid.
- shared: cache has a copy of the data that is consistent with the memory and with other caches.
- valid: cache contains an exclusive copy of the data that is consistent with the memory.

There are no coherence actions in case of a read hit. The cases of read miss, write hit, and write miss are described bellow.

Four coherence actions on a *read miss*:

- if there is another cache in state dirty, the latter supplies the data, writes it back to memory and both caches move simultaneously to state shared (t_1).
- if there is a cache in state shared or valid, then the latter supplies the data, and all caches at state valid move simultaneously to state shared (t_5).
- if there is no cached copy, then the current cache gets a copy from memory and moves to state valid (t_2).
- transitions t_4, t_7, and t_8 all model a cache line replacement.

Two coherence actions on a *write hit*:

- if the cache is in state valid, then it moves to state dirty (t_9).
- if the cache is in state shared, then it moves to state dirty and all other copies of the data are *invalidated* (t_6).

On a *write miss* all other copies are *invalidated* and the current cache moves to state dirty (t_3).

Table 13.5: Illinois Coherence Protocol

In the Illinois Protocol cache coherence protocol, each cache can be in one of the states *invd*,*dirt*,*shrd*,*vlid*. These respectively correspond to *invalid*, *dirty*, *shared*, and *valid*.

States:

$Q = \{invd, dirt, shrd, vlid\}$ *originally invd*

Transition rules:

t_1 : $invd$	$\to$	$shrd$	$\triangleright$	$\exists_{LR}\,(o{\cdot}dirt \wedge o{\cdot}shrd^{next})$
t_2 : $invd$	$\to$	$vlid$	$\triangleright$	$\forall_{LR}\,(o{\cdot}invd)$
t_3 : $invd$	$\to$	$dirt$	$\triangleright$	$\forall_{LR}\,(o{\cdot}invd^{next})$

$t_4 :$ *dirt*	$\rightarrow$	*invd*	$\triangleright$	*true*
$t_5 :$ *invd*	$\rightarrow$	*shrd*	$\triangleright$	$\exists_{LR}\,(o{\cdot}vlid \,\vee\, o{\cdot}shrd)$ $\wedge\, \forall_{LR}\,(\neg o{\cdot}vlid \,\vee\, (o{\cdot}vlid \,\wedge\, o{\cdot}shrd^{next}))$
$t_6 :$ *shrd*	$\rightarrow$	*dirt*	$\triangleright$	$\forall_{LR}\,(\neg o{\cdot}shrd \,\vee\, (o{\cdot}shrd \,\wedge\, o{\cdot}invd^{next}))$
$t_7 :$ *shrd*	$\rightarrow$	*invd*	$\triangleright$	*true*
$t_8 :$ *vlid*	$\rightarrow$	*invd*	$\triangleright$	*true*
$t_9 :$ *vlid*	$\rightarrow$	*dirt*	$\triangleright$	*true*

Final configurations:

only $(q_1 = dirt)$ only $(q_2 = dirt)$,
only $(q_1 = dirt)$ only $(q_2 = shrd)$,
only $(q_1 = shrd)$ only $(q_2 = dirt)$

In the Illinois protocol, unsafe configurations are those where a cache in state dirty coexists with another in state shared or dirty.

13.2.2 DEC Firefly Coherence Protocol

The states of a cache range in this protocol over *invd* (invalid), *dirt* (dirty), *excl* (exclusive), and *shrd* (shared). A cache is in state exclusive if it has the only copy of the data in the memory. A cache is in state shared if there are other copies of the data. Finally, a cache is in state dirty if it has a copy that is not consistent with the memory. Initially all processes are at state invalid.

The coherence actions are as follows. Three coherence actions in case of a read miss:

- if there is another cache in state dirty, the latter supplies the data, writes it back to the memory and both caches move to state shared (t_1).
- if there is a cache in state shared or exclusive, then the latter supplies the data, and all caches in state exclusive go to state shared (t_3).
- if there is no cached copy , (i.e., all other caches are in state invalid), then the current cache gets a copy from memory and moves to state exclusive (t_2).

Two coherence actions in case of a write hit:

- if the cache is in state exclusive, then it moves to state dirty (t_4).
- if the cache is the only one in state shared, then it moves to state exclusive (t_6).

For a write miss, the cache moves to state dirty and simultaneously all other copies are invalidated, i.e., all other caches go to state dirty (t_5). Observe that no coherence actions are needed for a read hit.

Table 13.6: Dec Firefly Coherence Protocol

In the Dec Firefly cache coherence protocol, each cache can be in one of the states *invd*, *excl*, *shrd*, *dirt*. These correspond to *invalid*, *exclusive*, *shared*, and *dirty* states.

States:
$Q = \{invd, excl, shrd, dirt\}$ *originally invd*

Transition rules:

t_1 : *invd*	$\rightarrow$	*shrd*	$\triangleright$	$\exists_{LR}\,(o{\cdot}dirt \;\wedge\; o{\cdot}shrd^{next})$
t_2 : *invd*	$\rightarrow$	*excl*	$\triangleright$	$\forall_{LR}\,(o{\cdot}invd)$
t_3 : *invd*	$\rightarrow$	*shrd*	$\triangleright$	$\exists_{LR}\,(o{\cdot}excl \;\vee\; o{\cdot}shrd)$ $\wedge\,\forall_{LR}\,(\neg o{\cdot}excl \;\vee\; (o{\cdot}excl \;\wedge\; o{\cdot}shrd^{next}))$
t_4 : *excl*	$\rightarrow$	*dirt*	$\triangleright$	$true$
t_5 : *invd*	$\rightarrow$	*dirt*	$\triangleright$	$\forall_{LR}\,(o{\cdot}invd^{next})$
t_6 : *shrd*	$\rightarrow$	*excl*	$\triangleright$	$\forall_{LR}\,(\neg o{\cdot}shrd)$

Final configurations:

$\text{only}\,(q_1 = dirt) \;\wedge\; \text{only}\,(q_2 = shrd)$,
$\text{only}\,(q_1 = dirt) \;\wedge\; \text{only}\,(q_2 = excl)$,
$\text{only}\,(q_1 = shrd) \;\wedge\; \text{only}\,(q_2 = dirt)$,
$\text{only}\,(q_1 = excl) \;\wedge\; \text{only}\,(q_2 = dirt)$,
$\text{only}\,(q_1 = excl) \;\wedge\; \text{only}\,(q_2 = excl)$,
$\text{only}\,(q_1 = dirt) \;\wedge\; \text{only}\,(q_2 = dirt)$

In the DEC Firefly protocol, a configuration is unsafe if (i) a cache in state dirty coexists with a cache in state shared, exclusive, or dirty; and (ii) two caches coexist in state exclusive.

13.2.3 Futurebus+ Coherence Protocol

We derive for the Futurebus [Han93] coherence protocol a parameterized version from [Del00b]. This protocol uses the signal *transaction flag* whose state (asserted or not) depends on the transaction taking effect. The transaction flag gives the requesting cache the option of taking data immediately into an exclusive state, if that data is not copied in any other cache.

In our parameterized model, Read cycles are described in terms of states in invalid (*invld*), shared unmodified (*shunm*), exclusive unmodified (*exunm*), exclusive modified (*exmod*), pending read (*pread*). We also write (*pexcl*) to mean that a cache in state exclusive modified is ready to respond with the data to a

read command; and (*punm*) to mean that caches in states shared unmodified or exclusive unmodified are ready to assert the transaction flag. Write cycles are described via two additional local states. The state (*pwrit*) indicates caches that have issued a read modified command and wait for a data acknowledgment; the state (*pewr*) indicates that a cache in state exclusive modified is preparing to issue the data acknowledgment. In our model we assume the transaction flag is only up if there exists a cache in a pending exclusive state (*pexcl*) or a pending shared state (*punm*), (i.e., if ($\exists_{LR}(pexcl \vee punm)$) holds). Furthermore, to avoid trivial inconsistencies we assume that a cache in state invalid can issue a read shared or a read modified command only if there are no pending write commands ($\forall_{LR}(\neg pwrit)$).

1. Read hits do not cause state transitions.
2. When a cache issues a read shared command and goes to state pending read (*pread*), all caches in state shared unmodified (*shunm*) or exclusive unmodified (*exunm*) go to state pending unmodified (*punm*) and prepare to set the transaction flag. A cache in state exclusive modified (*exmod*) goes to state pending exclusive (*pexcl*) and prepares (in transition t_1) to send a data acknowledgment (ready signal).
3. If the data acknowledgment is sent by a cache in a pending exclusive state (*pexcl*) or by main memory, and if the transaction flag has been asserted then caches at state pending read (*pread*) go to state shared unmodified (*shunm*) in transitions t_2 and t_3.
4. If the transaction flag has not been asserted and more than one cache has pending read requests, then all caches move to states shared unmodified (*shunm*) in transition t_4.
5. If the transaction flag has not been asserted and only one cache is in a pending read state (*pread*), then it changes to state exclusive unmodified (*exunm*) in transition t_5.
6. When a cache issues a read modified command and goes to a pending write state (*pwrit*), the cache in state exclusive modified (*exmod*) goes to state pending exclusive write (*pewr*), while all other caches get invalidated (transition t_6).
7. When a data acknowledgment is issued then the cache goes to exclusive modified (*exmod*) in transitions t_7 and t_8.
8. Write hits in states exclusive unmodified (*exunm*) and exclusive modified (*exmod*) need no coherence actions (transitions t_9 and t_{10}).
9. When the cache is in state shared unmodified (*shunm*), all other caches in the same state are invalidated (transition t_{11}).

Table 13.7: Futurebus Coherence Protocol

In the Futurebus coherence protocol, each cache can be in one of the following states: invalid ($invld$), shared unmodified ($shunm$), exclusive unmodified ($exunm$), exclusive modified ($exmod$), pending read ($pread$), pending exclusive ($pexcl$), pending shared unmodified ($punm$), pending write ($pwrit$), and pending exclusive write ($pewr$).

States:

$Q = \{pread, shunm, punm, pexcl, exunm, exmod, pewr, pwrit, invld\}$

originally invld

Transition rules:

$$t_1 : invld \rightarrow pread \triangleright \forall_{LR}(\neg o \cdot pwrit) \wedge \forall_{LR}\begin{pmatrix} \neg o \cdot exmod & \vee \\ \neg o \cdot shunm \vee \neg o \cdot exunm & \vee \\ o \cdot exmod \wedge o \cdot pexcl^{next} & \vee \\ o \cdot shunm \wedge o \cdot punm^{next} & \vee \\ o \cdot exunm \wedge o \cdot punm^{next} & \end{pmatrix}$$

$$t_2 : pexcl \rightarrow shunm \triangleright \forall_{LR}\begin{pmatrix} \neg o \cdot pread & \vee \\ o \cdot pread \wedge o \cdot shunm^{next} & \end{pmatrix}$$

$$t_3 : punm \rightarrow shunm \triangleright \forall_{LR}\begin{pmatrix} \neg o \cdot pread \vee \neg o \cdot punm & \vee \\ o \cdot pread \wedge o \cdot shunm^{next} & \vee \\ o \cdot punm \wedge o \cdot shunm^{next} & \end{pmatrix}$$

$$t_4 : pread \rightarrow shunm \triangleright \exists_{LR}(o \cdot pread) \wedge \forall_{LR}(\neg(o \cdot punm \vee o \cdot pexcl)) \wedge \forall_{LR}\begin{pmatrix} \neg o \cdot pread & \vee \\ o \cdot pread \wedge o \cdot shunm^{next} & \end{pmatrix}$$

$$t_5 : pread \rightarrow exunm \triangleright \forall_{LR}(\neg(o \cdot shunm \vee o \cdot pexcl \vee o \cdot pread))$$

$$t_6 : invld \rightarrow pwrit \triangleright \forall_{LR}(\neg o \cdot pwrit) \wedge \forall_{LR}\begin{pmatrix} \neg o \cdot invld & \vee \\ \neg o \cdot exmod \wedge o \cdot pewr^{next} & \vee \\ o \cdot exmod \wedge o \cdot invld^{next} & \end{pmatrix}$$

$$t_7 : pewr \rightarrow invld \triangleright \forall_{LR}\begin{pmatrix} \neg o \cdot pwrit & \vee \\ o \cdot pwrit \wedge o \cdot exmod^{next} & \end{pmatrix}$$

$$t_8 : pwrit \rightarrow exmod \triangleright \forall_{LR}(\neg o \cdot pewr) \wedge \forall_{LR}\begin{pmatrix} \neg o \cdot pwrit & \vee \\ o \cdot pwrit \wedge o \cdot exmod^{next} & \end{pmatrix}$$

t_9 : *exunm*	$\rightarrow$	*exmod*	$\triangleright$	*true*
t_{10} : *exmod*	$\rightarrow$	*exmod*	$\triangleright$	*true*
t_{11} : *shunm*	$\rightarrow$	*exmod*	$\triangleright$	$\forall_{LR}\begin{pmatrix} \neg o{\cdot}shunm & \vee \\ o{\cdot}shunm \wedge o{\cdot}invld^{next} & \end{pmatrix}$

Final configurations:

$\text{only}\,(q_1 = shunm) \wedge \text{only}\,(q_2 = exmod)$,
$\text{only}\,(q_1 = shunm) \wedge \text{only}\,(q_2 = exunm)$,
$\text{only}\,(q_1 = exmod) \wedge \text{only}\,(q_2 = shunm)$,
$\text{only}\,(q_1 = exmod) \wedge \text{only}\,(q_2 = exunm)$,
$\text{only}\,(q_1 = exunm) \wedge \text{only}\,(q_2 = shunm)$,
$\text{only}\,(q_1 = exunm) \wedge \text{only}\,(q_2 = exunm)$,
$\text{only}\,(q_1 = exunm) \wedge \text{only}\,(q_2 = exmod)$,
$\text{only}\,(q_1 = pwrit) \wedge \text{only}\,(q_2 = pwrit)$,
$\text{only}\,(q_1 = pwrit) \wedge \text{only}\,(q_2 = pread)$,
$\text{only}\,(q_1 = pread) \wedge \text{only}\,(q_2 = pwrit)$

For this model, we were able to automatically verify that no more than one cache is in an exclusive state, that different caches cannot be simultaneously in exclusive and shared states, that there cannot be simultaneously caches that emitted write and read commands or more than one write command.

13.2.4 German Coherence Protocol

In this protocol [PRZ01], a central controller, denoted by *Home*, is used to manage the access of an arbitrary, but finite, number of *clients* to a cache line. In the considered model, each process corresponds to a client, while Home bounded local variables are modeled as shared variables (its actions are represented in each process).

A client can be in one of the three states: invalid (*invalid*), shared (*shared*), or exclusive (*exclusive*). A client is in state invalid if it does not have access to the cache line. A client is in state shared if it has been granted the access (by Home) possibly with other clients (also in state shared). Home can also grant the access exclusively to a client (state exclusive).

Each client communicates with Home via three channels: ch_1, ch_2 and ch_3. Since the channels are considered to be of length one, each of them can be represented by a local variable ch_i for $channel_i$.

Table 13.8: German Coherence Protocol

States:
$Q = \{invl, shrd, excl\}$ *originally invl*
$any \in Q$

Local:
ch_1 is in $\{\varepsilon, reqShr, reqExc\}$ *originally* ε
ch_2 is in $\{\varepsilon, graShr, graExc, invldt\}$ *originally* ε
ch_3 is in $\{\varepsilon, invAck\}$ *originally* ε
$curClt, sList, iList$ are *Booleans originally false*

Shared:
$excGrted$ is *Boolean originally false*
$currComm$ is in $\{\varepsilon, reqShr, reqExc\}$ *originally* ε

Transition rules:

$home_0 : any \rightarrow any \triangleright currComm = reqShr \wedge \neg excGrted \wedge ch_2 = \varepsilon \wedge curClt \wedge currComm^{next} = \varepsilon \wedge sList^{next} \wedge ch_2^{next} = graShr$

$home_1 : any \rightarrow any \triangleright curClt \wedge \forall_{LR}(\neg(o{\cdot}sList)) \wedge currComm = reqExc \wedge ch_2 = \varepsilon \wedge currComm^{next} = \varepsilon \wedge sList^{next} \wedge ch_2^{next} = graExc \wedge excGrted^{next}$

$home_2 : any \rightarrow any \triangleright currComm = \varepsilon \wedge ch_1 \neq \varepsilon \wedge currComm^{next} = ch_1 \wedge iList^{next} = sList \wedge ch_1^{next} = \varepsilon \wedge curClt^{next} \wedge \forall_{LR}\left(\begin{array}{l} (o{\cdot}iList)^{next} = o{\cdot}sList \quad \wedge \\ \neg(o{\cdot}curClt^{next}) \end{array} \right)$

$home_3 : any \rightarrow any \triangleright currComm = reqShr \wedge excGrted \wedge iList \wedge ch_2 = \varepsilon \wedge \neg(iList^{next}) \wedge ch_2^{next} = invldt$

$home_4 : any \rightarrow any \triangleright currComm = reqExc \wedge iList \wedge ch_2 = \varepsilon \wedge \neg(iList^{next}) \wedge ch_2^{next} = invldt$

$home_5 : any \rightarrow any \triangleright currComm \neq \varepsilon \wedge ch_3 = invAck \wedge \neg(sList^{next}) \wedge \neg(excGrted^{next}) \wedge ch_3^{next} = \varepsilon$

$client_1$: $invl$	$\rightarrow$	$invl$	$\triangleright$	$ch_1 = \varepsilon \wedge ch_1^{next} = reqShr$
$client_2$: $invl$	$\rightarrow$	$invl$	$\triangleright$	$ch_1 = \varepsilon \wedge ch_1^{next} = reqExc$
$client_3$: $shrd$	$\rightarrow$	$shrd$	$\triangleright$	$ch_1 = \varepsilon \wedge ch_1^{next} = reqExc$
$client_4$: any	$\rightarrow$	$invl$	$\triangleright$	$ch_2 = invldt \wedge ch_3 = \varepsilon$ $\wedge\ ch_2 = \varepsilon \wedge ch_3^{next} = invAck$
$client_5$: any	$\rightarrow$	$shrd$	$\triangleright$	$ch_2 = graShr \wedge ch_2^{next} = \varepsilon$
$client_6$: any	$\rightarrow$	$excl$	$\triangleright$	$ch_2 = graExc \wedge ch_2^{next} = \varepsilon$

Final states:

$\text{only}\,(q_1 = excl) \wedge \text{only}\,(q_2 = shrd)$,
$\text{only}\,(q_1 = excl) \wedge \text{only}\,(q_2 = excl)$,
$\text{only}\,(q_1 = shrd) \wedge \text{only}\,(q_2 = excl)$

Depending on the content of the channels and the values of the shared variables, Home may perform one of the following actions.

1. If Home is idle ($currComm = \varepsilon$) and receives a request via ch_1, then Home updates $currComm$ with the received request, empties ch_1, selects the sender to be the current client and copies the content of the sharer list to the invalidation list. The client selection and the list copying are modeled with a broadcast ($home_2$).
2. In case ch_2 is empty, the current command is a shared request and the exclusive access has not been granted, then Home sends a grant for a shared access to the current client via ch_2 and adds the client to the sharer list and becomes idle ($home_0$).
3. In case ch_2 is empty, the current command is an exclusive request and the sharer list is empty, then Home sends a grant for the exclusive access to the current client via ch_2, adds the client to the sharer list, sets the exclusive flag and becomes idle ($home_1$).
4. If the current command is either a shared request (while the exclusive flag is set) or an exclusive request, ch_2 is empty, then Home sends an invalidation message every process through ch_2 and removes these processes from the invalidation list ($home_3$ and $home_4$).
5. If the current command is a request for either a shared or an exclusive access and Home receives an invalidation acknowledgment from a client via ch_3, in this case Home removes a client from the sharer list, resets the exclusive flag and empties ch_3 ($home_5$).

The safety properties we checked are: (i) no two clients are simultaneously granted an exclusive access, (ii) no client in state shared coexists with a client in state exclusive.

13.3 Unbounded Parameterized Mutex Algorithms

We show in this section how to model three (non-trivial) formulations for the Bakery algorithm by Lamport [Lam74]. The algorithm is a well-known solution to the critical section problem where an arbitrary and finite number of processes compete for a shared resource.

The rationale behind the algorithm is as follows. Each process has a local numerical variable *ticket* in which it stores a ticket. Initially *ticket* equals zero. When a process is interested in entering the critical section, it sets its *ticket* to a value strictly greater than the values of the tickets belonging to all other processes in the system (*choosing phase*). After this phase, a process waits until its ticket is less than all positive tickets belonging to other processes (*entry phase*). In the *releasing phase* , the process resets its ticket to zero.

```
var ticket: shared array [0..n-1] of integer;

Process P[i] :=
  loop forever
    ticket[i] := max(ticket[0],...,ticket[n-1])+1;
    for j := 0 to n-1 do begin
      while ticket[j]<>0
            and (ticket[j],j) < (ticket[i],i)
          do skip;
    end;
    (* critical section *)
    ticket[i] := 0;
    (* non-critical section *);
```

Figure 13.1: Pseudo-code for Bakery algorithm

In the following, we specify three different formulations for the above algorithm. In the first formulation, we introduce an auxiliary variable to break down the choosing phase in two steps. The second formulation is obtained by modifying the test in the *entry phase* of the first version. This modification introduces a bug (i.e., violation of mutual exclusion becomes possible). In the third formulation, we break even more the *choosing phase* and the *entry phase*, and consider all possible interleavings.

13.3.1 Bakery with Races

A process can have one of four states: *idle*, *choose*, *wait*, and *use*; where the last one represents the critical section.

In addition to the numerical variable *ticket* described above, we break down the atomicity of the *entry phase* by means of an additional numerical variable *aux*. The *entry phase* is here performed in two steps. In the first step,

the auxiliary *aux* variable belonging to a process interested in accessing its critical section takes a value strictly larger than the values of all other *tickets* in the system (transition t_1). In the second step, the variable *ticket* takes the value of *aux* (transition t_2). The process gains access to the critical section if neither of the other processes is choosing its *ticket* value, or has priority in terms of *ticket* values (transition t_3). Here, ties are broken using the position in the array (hence the use of the "isright" Boolean array introduced in 12.5). The process puts zero in its *ticket* variable while releasing the shared resource (transition t_4).

Table 13.9: Bakery Mutex

Bakery mutual exclusion algorithm. Each process has two local numerical variables: *ticket* and *aux*. The second variable is used to break the atomicity of the *entry phase* . A process can have one of four states: *idle*, *choose*, *wait*, and *use*; where the last state represents the critical section.

States:
$Q = \{idle, choose, wait, use\}$
originally *idle*

Local:
$ticket, aux$ are in $\mathbb{N}$ *originally zero*

Transition rules:

t_1 : $idle$	$\rightarrow$	$choose$	$\triangleright$	$\forall_{LR}\,(o{\cdot}ticket < aux^{next})$
t_2 : $choose$	$\rightarrow$	$wait$	$\triangleright$	$ticket^{next} = aux$
t_3 : $wait$	$\rightarrow$	use	$\triangleright$	$\forall_{LR}\left(\begin{array}{l}\neg o{\cdot}choose\ \wedge \\ \left(\begin{array}{l} o{\cdot}ticket = 0 \\ \vee\ ticket < o{\cdot}ticket \\ \vee\ (ticket = o{\cdot}ticket \wedge 1_R)\end{array}\right)\end{array}\right)$
t_3 : use	$\rightarrow$	$idle$	$\triangleright$	$ticket^{next} = 0$

Final states:
$\text{only}\,(q_1 = use)\ \wedge\ \text{only}\,(q_2 = use)$

The set of bad configurations correspond to two processes in their state *use*.

13.3.2 Bakery with a Bug

This version is almost identical to the previous formulation of the Bakery algorithm. The only difference is that other processes are not required, when evaluating the condition to enter critical section, to be in other states than *choose*.

Table 13.10: Bakery Mutex with a Bug

Bakery mutual exclusion algorithm. Each process has two local numerical variables: *ticket* and *aux*. The second variable is use to break the atomicity of the *entry phase* . A process can have one of four states: *idle*, *choose*, *wait*, and *use*; where the last one represents the critical section.

States:	**Local:**
$Q = \{idle, choose, wait, use\}$ originally idle	$ticket, aux$ are in $\mathbb{N}$ *originally zero*

Transition rules:

t_1 : *idle*	$\rightarrow$	*choose*	$\triangleright$	$\forall_{LR}\,(o{\cdot}ticket < aux^{next})$
t_2 : *choose*	$\rightarrow$	*wait*	$\triangleright$	$ticket^{next} = aux$
t_3 : *wait*	$\rightarrow$	*use*	$\triangleright$	$\forall_{LR}\begin{pmatrix} o{\cdot}ticket = 0 & \vee \\ ticket < o{\cdot}ticket & \vee \\ (ticket = o{\cdot}ticket \wedge 1_R) & \end{pmatrix}$
t_3 : *use*	$\rightarrow$	*idle*	$\triangleright$	$ticket^{next} = 0$

Final states:

only $(q_1 = use)$ $\wedge$ only $(q_2 = use)$

The set of bad configurations correspond to two processes in their state *use*. This algorithm contains a bug. Consider a system with two processes *one* and (to its right) *two*. Say both processes choose five as value for their respective auxiliary variables. Process *two* can access the critical section; followed by process *one* to its left (transition t_3).

13.3.3 Bakery with all Interleavings

The formulation presented here allows all interleavings in the *choosing phase* and the *entry phase*. Each process has two local numerical variables: *ticket* and *aux* that play similar roles as in the previous formulations. In addition, each process owns a local Boolean array it uses to mark other processes when evaluating the global conditions appearing in transitions t_1 and t_3 in the first formulation. Using these arrays, it becomes possible to evaluate locally universal global conditions.

A process can take one of the five states: *idle*, *choose*, *wait*, *eval*, and *use*; where the last state represents the critical section.

Table 13.11: Bakery Mutex with all Interleavings

Bakery mutual exclusion algorithm with all interleavings in both the *choosing phase* and the *entry phase* . Each process uses a local array *check* of Booleans to mark other processes when evaluating the entry condition and choosing a value for *ticket*. A process can have one of five states: *idle*, *choose*, *wait*, and *use*; where the last state represents the critical section.

States:
$Q = \{idle, choose, wait, use\}$
originally idle

Local:
$ticket, aux$ are in $\mathcal{N}$ *originally zero*
check is an array of *Booleans*
originally false

Transition rules:

Rule	From		To		Condition
t_1 :	$idle$	$\rightarrow$	$choose$	$\triangleright$	$aux^{next} = ticket \wedge \forall_{LR}\ (\neg(o\cdot check^{next}))$
t_2 :	$choose$	$\rightarrow$	$choose$	$\triangleright$	$\exists_{LR}(\neg o\cdot check \wedge o\cdot check^{next} \wedge o\cdot ticket \leq aux) \vee \exists_{LR}\begin{pmatrix} \neg o\cdot check \wedge o\cdot check^{next} \\ \wedge\, aux < o\cdot ticket \\ \wedge\, aux^{next} = o\cdot ticket \end{pmatrix}$
t_3 :	$choose$	$\rightarrow$	$wait$	$\triangleright$	$aux < ticket^{next} \wedge \forall_{LR}\ (o\cdot check \wedge \neg(o\cdot check^{next}))$
t_5 :	$wait$	$\rightarrow$	$wait$	$\triangleright$	$\exists_{LR}\begin{pmatrix} \neg check \wedge check^{next} \wedge \\ \neg o\cdot choose \wedge \\ \begin{pmatrix} o\cdot ticket = 0 \vee ticket < o\cdot ticket \\ \vee\, (o\cdot ticket = ticket \wedge 1_R) \end{pmatrix} \end{pmatrix}$
t_6 :	$wait$	$\rightarrow$	use	$\triangleright$	$\forall_{LR}\ (o\cdot check)$
t_7 :	use	$\rightarrow$	$idle$	$\triangleright$	$aux^{next} = 0$

Final states:

$\text{only}\,(q_1 = use) \wedge \text{only}\,(q_2 = use)$

The set of bad configurations correspond to two processes in their state *use*.

13.4 Originally Distributed Mutex Algorithms

We describe in this section two well know algorithms: The distributed mutual exclusion algorithm by Lamport [Lam78], and its adaptation by Ricart and Agrawala [RA81]. These algorithms are tailored for processes communicating

by message passing. For each access to the critical section, the first algorithm requires $3(n-1)$ exchanged messages (where n is the size of the instance). The second algorithm lowers this number to $2(n-1)$. We explain in the following how to describe these protocols in our framework.

13.4.1 Distributed Mutex by Lamport

A process in this model has three states: *idle*, *wait* and *use* (for critical section). Each process i has a logical clock $clock_i$ that ranges over the natural numbers, and a local queue Q_i in which i stores received timestamped requests. Mutual exclusion is guaranteed by letting only the process with the *earliest* request access the critical section. The processes can only communicate by message passing. Here *earliest* is in the sens of the *happened-before* [Lam78] partial ordering combined with process positions in the array to brake ties (total ordering). This order is also used to define heads of the queues in the processes.

Initially, all processes are at their *idle* state with empty queues and clock values equal to the minimal variable *zero*. A timestamped message is a tuple (m,ts) where m is either a request, an acknowledgment or a release, and ts the associated timestamp value. In each of the following three steps, process i strictly increases its *clock* value before proceeding with the rest of the actions in the step.

- *idle*: process i broadcasts the timestamped request $(req_i, clock_i)$, stores a copy of $[i, clock_i]$ in its local queue Q_i, and moves to state *wait*;
- *wait*: process i accesses its critical section (state *use*) if replies have been collected from all other processes, and if its own request is at the top of its queue Q_i;
- *use*: process i releases the critical section (moves back to state *idle*), removes its own request from the queue Q_i, and broadcasts a timestamped release message $(rel_i, clock_i)$ (where $clock_i$ is the new clock value).

In each of the following four transitions, a process j (that receives a message with timestamp ts) updates its local clock to a value strictly larger than $\max(clock, ts)$, and depending on the message type, does the following:

- a *request* (req_i, ts) is received by process j from process i; process j adds (i, ts) to its own local queue Q_j, and sends back a reply message $(ack_j, clock_j)$ timestamped with the new clock value;
- an *acknowledgment* (ack_i, ts) is received by process j from process i, only update the local clock;
- a *release* (rel_i, ts) is received by process j from process i; process j removes all requests associated with i from its local queue Q_j.

Our model of the distributed Lamport algorithm is depicted in table 13.12. In this parameterized system we use a numerical variable to model the *clock*; and the combination of a local array *Queue* of numerical variables and a local numerical variable *queue* to model a local queue Q_i. The array *Queue* is used for storing requests from other processes, while the variable *queue* for storing

self requests. Communication channels are modeled via two pairs (q_{ch_1}, ts_{ch_1}) of local arrays. These correspond to bounded channels (of size two).

We define here formulas to augment the readability of the model. Assume in the following *tstamp* to be a numerical variable and *msg* to be a bounded variable taking its values in $\{req, ack, rel\}$. In the following, formulas written between brackets are to be copied as is. The assumption stating that non mentioned next-value variables are copied applies on formulas resulting from combination of formulas within parenthesis. For instance, assume three Boolean variables: a, b and c. We mean:

$$\left(\left[a^{next}\right]\right) \wedge \left(\left[c^{next}\right]\right) = a^{next} \wedge c^{next} \wedge (b^{next} = b)$$

Queue

The formula $push(s, tstamp)$ is defined by $[queue^{next} = tstamp]$; it corresponds to a process pushing in its local queue the value of variable *tstamp* associated with its proper index. We write $push(o, tstamp)$ to mean $[o{\cdot}Queue^{next} = tstamp]$; this corresponds to a process pushing in its local queue the value of variable *tstamp* associated with another process "o". The formula $pop(s)$ is defined by $[queue^{next} = zero]$ and corresponds to a process popping from its local queue the value associated with its index; in a similar manner, $pop(o)$ is defined by $[o{\cdot}Queue^{next} = zero]$ and corresponds to a process popping from its local queue the value associated with another process "o".

Precedence

The formula $aheadof(o)$ evaluates to true if the process evaluating is, in its local queue, ahead of process "o", where the latter sent a request that was pushed in the queue. This formula is defined by:

$$aheadof(o) = [(o{\cdot}1_L \wedge queue < o{\cdot}Queue) \vee (o{\cdot}1_R \wedge queue \leq o{\cdot}Queue)]$$

Communication

We use $send(o, msg, tstamp)$ and $get(o, msg, tstamp)$ to mean sending and receiving a message respectively. These are defined by:

$$send(o, msg, tstamp) = \begin{bmatrix} o{\cdot}q_{ch_1} = \varepsilon \wedge o{\cdot}ts_{ch_1} = zero \\ \wedge\, o{\cdot}q_{ch_1}^{next} = msg \wedge o{\cdot}ts_{ch_1}^{next} = tstamp \end{bmatrix}$$

$$get(o, msg, tstamp) = \begin{bmatrix} s{\cdot}q_{ch_2}[o] = msg \wedge s{\cdot}ts_{ch_2}[o] = tstamp \\ \wedge\, s{\cdot}q_{ch_2}[o]^{next} = \varepsilon \wedge s{\cdot}ts_{ch_2}[o]^{next} = zero \end{bmatrix}$$

where $s{\cdot}channel[o]$ are as introduced in section 12.5.

Finally, in order to non-deterministically "move" the messages in the channels we define both formulas $advance(s, o)$ and $copy(s, o)$. Formula

$advance(s,o)$ corresponds to moving forward the content in the channel from the current process to the other process "o"; and is defined by

$$advance(s,o) = \left[\begin{array}{lclc} o{\cdot}q_{ch_1} = msg & \wedge & o{\cdot}q_{ch_2} = \varepsilon & \wedge \\ o{\cdot}ts_{ch_1} = tstamp & \wedge & o{\cdot}ts_{ch_2} = zero & \wedge \\ o{\cdot}q^{next}_{ch_1} = \varepsilon & \wedge & o{\cdot}q^{next}_{ch_2} = msg & \wedge \\ o{\cdot}ts^{next}_{ch_1} = zero & \wedge & o{\cdot}ts^{next}_{ch_2} = tstamp & \end{array}\right]$$

The second formula corresponds to a channel where the content has not changed, and is defined by:

$$copy(s,o) = \left[\begin{array}{lclc} o\cdot q^{next}_{ch_1} = o\cdot q_{ch_1} & \wedge & o\cdot q^{next}_{ch_2} = o\cdot q_{ch_2} & \wedge \\ o\cdot ts^{next}_{ch_1} = o\cdot ts_{ch_1} & \wedge & o\cdot ts^{next}_{ch_2} = o\cdot ts_{ch_2} & \end{array}\right]$$

Table 13.12: Lamport Distributed Mutex

Lamport Distributed Mutex

States:
$Q = \{idle, wait, use\}, any \in Q$

Local:
$clock$ is a $natural$ $originally\ zero$
$replied$ is an array of $Booleans$ $originally\ false$
$queue$ is a $natural$ $originally\ zero$
$Queue$ is an array of $naturals$ $originally\ zero$
ts_{ch_1}, ts_{ch_2} are arrays of $naturals$ $originally\ zero$
q_{ch_1}, q_{ch_2} are arrays of $\{\varepsilon, req, ack, rel\}$ $originally\ \varepsilon$

Acquiring and releasing the ressource:

$t_1:\ idle$	$\rightarrow$	$wait$	$\triangleright$	$(clock < clock^{next}) \wedge (push(s, clock^{next})) \wedge (\forall_{LR}(send(o, req, clock^{next})))$
$t_2:\ wait$	$\rightarrow$	use	$\triangleright$	$(clock < clock^{next}) \wedge (\forall_{LR}(o{\cdot}replied \wedge aheadof(o)))$
$t_3:\ use$	$\rightarrow$	$idle$	$\triangleright$	$(clock < clock^{next}) \wedge (\forall_{LR}(send(o, rel, clock^{next})))$

Handling incoming messages:

$t_4:\ any$	$\rightarrow$	any	$\triangleright$	$\exists_{LR}\left(\begin{array}{l} (get(o, req, s{\cdot}ts_{ch_2}[o])) \\ \wedge\ (push(o, s{\cdot}ts_{ch_2}[o])) \\ \wedge\ (send(o, ack, clock^{next})) \\ \wedge\ (clock < clock^{next}) \\ \wedge\ (s{\cdot}ts_{ch_2}[o] < clock^{next}) \end{array}\right)$

$t_5:$	any	$\rightarrow$	any	$\triangleright$	$\exists_{LR}\left(\begin{array}{l}\left(get\left(o, ack, s{\cdot}ts_{ch_2}[o]\right)\right) \\ \wedge\, o{\cdot}replied^{next} \\ \wedge\, (clock < clock^{next}) \\ \wedge\, \left(s{\cdot}ts_{ch_2}[o] < clock^{next}\right)\end{array}\right)$
$t_6:$	any	$\rightarrow$	any	$\triangleright$	$\exists_{LR}\left(\begin{array}{l}\left(get\left(o, rel, s{\cdot}ts_{ch_2}[o]\right)\right) \\ \wedge\, (pop(o)) \\ \wedge\, (clock < clock^{next}) \\ \wedge\, \left(s{\cdot}ts_{ch_2}[o] < clock^{next}\right)\end{array}\right)$
$t_7:$	any	$\rightarrow$	any	$\triangleright$	$\forall_{LR}((advance(s,o)) \vee (copy(s,o)))$

Final configurations:

only $(q_1 = use \wedge q_2 = use)$

The set of bad configurations corresponds to configurations where two processes are in state *use*.

13.4.2 Distributed Mutex by Ricart-Agrawala

The Ricart-Agrawala Algorithm [RA81] is a modification of the distributed mutual exclusion algorithm introduced above. The modification consists in sending a reply upon reception of a request if the sender is "earlier" with respect to precedence relation introduced earlier. If the process that sent the request is "later" according to this relation, then its request is put on hold, and the receiver will send the acknowledgment when releasing the critical section.

We describe the protocol in table 13.13. Each process has two numerical variables $clock$ and $last$. A process keeps in $last$ the value of $clock$ last time it broadcasted a request. A process has also two local Boolean arrays $replied$ and $deferred$. The array $replied$ plays the same role as in the previous algorithm. The array $deferred$ is used to remember the pending requests.

The $send(o, msg, tstamp)$, $get(o, msg, tstamp)$, $advance(s,o)$, and $copy(s,o)$ formulas are identical to those in our description of the previous algorithm. We use $before$ and $after$ to mean the following formulas :

$$\begin{aligned} before(o, tstamp) &= [(o{\cdot}1_L \wedge last < tstamp) \vee (o{\cdot}1_R \wedge last \leq tstamp)] \\ after(o, tstamp) &= [(o{\cdot}1_L \wedge tstamp \leq last) \vee (o{\cdot}1_R \wedge tstamp < last)] \end{aligned}$$

Table 13.13: Ricart-Agrawala Distributed Mutex

Ricart-Agrawala Distributed Mutex

States:
$Q = \{idle, wait, use\}, any \in Q$

Local:
$clock, last$ are $naturals$ $originally\ zero$
$replied, deferred$ are arrays of $Booleans$ $originally\ false$
ts_{ch_1}, ts_{ch_2} are arrays of $naturals$ $originally\ zero$
q_{ch_1}, q_{ch_2} are arrays of $\{\varepsilon, req, ack, rel\}$ $originally\ \varepsilon$

Acquiring and releasing the ressource:

$$t_1 : idle \rightarrow wait \triangleright (clock < clock^{next}) \wedge (last^{next} = clock^{next}) \wedge (\forall_{LR} (send(o, req, last^{next})))$$

$$t_2 : wait \rightarrow use \triangleright (clock < clock^{next}) \wedge (\forall_{LR} (o \cdot replied))$$

$$t_3 : use \rightarrow idle \triangleright (clock < clock^{next}) \wedge \forall_{LR} \left(\begin{array}{l} \left(\begin{array}{l} \neg o \cdot deferred \\ \wedge \neg (o \cdot replied^{next}) \end{array} \right) \\ \vee \left(\begin{array}{l} o \cdot deferred \\ \wedge \neg (o \cdot deferred^{next}) \\ \wedge (send(o, ack, clock^{next})) \\ \wedge \neg o \cdot replied^{next} \end{array} \right) \end{array} \right)$$

Handling incoming messages:

$$t_4 : any \rightarrow any \triangleright \exists_{LR} \left(\begin{array}{l} (get(o, ack, s \cdot ts_{ch_2}[o])) \wedge \\ \wedge (clock < clock^{next}) \\ \wedge (s \cdot ts_{ch_2}[o] < clock^{next}) \\ \wedge o \cdot replied^{next} \end{array} \right)$$

$$t_5 : idle \rightarrow idle \triangleright \exists_{LR} \left(\begin{array}{l} (get(o, req, s \cdot ts_{ch_2}[o])) \\ \wedge (clock < clock^{next}) \\ \wedge (s \cdot ts_{ch_2}[o] < clock^{next}) \\ \wedge (send(o, ack, clock^{next})) \end{array} \right)$$

$$t_6 : wait \rightarrow wait \triangleright \exists_{LR} \left(\begin{array}{l} (get(o, req, s \cdot ts_{ch_2}[o])) \\ \wedge (clock < clock^{next}) \\ \wedge (s \cdot ts_{ch_2}[o] < clock^{next}) \\ \wedge (after(o, s \cdot ts_{ch_2}[o])) \\ \wedge (send(o, ack, clock^{next})) \end{array} \right)$$

$$t_7 : wait \rightarrow wait \triangleright \exists_{LR} \left(\begin{array}{l} (get(o, req, s \cdot ts_{ch_2}[o])) \\ \wedge (clock < clock^{next}) \\ \wedge (s \cdot ts_{ch_2}[o] < clock^{next}) \\ \wedge (before(o, s \cdot ts_{ch_2}[o])) \\ \wedge o \cdot deferred^{next} \end{array} \right)$$

$$t_8 : \mathit{use} \quad \rightarrow \quad \mathit{use} \quad \triangleright \quad \exists_{LR} \begin{pmatrix} \left(get\left(o, req, s{\cdot}ts_{ch_2}[o]\right)\right) \\ \wedge\ \left(clock < clock^{next}\right) \\ \wedge\ \left(s{\cdot}ts_{ch_2}[o] < clock^{next}\right) \\ \wedge\ o{\cdot}deferred^{next} \end{pmatrix}$$

$$t_9 : \mathit{any} \quad \rightarrow \quad \mathit{any} \quad \triangleright \quad \forall_{LR}\left(\left(advance(s,o)\right) \vee \left(copy(s,o)\right)\right)$$

Final configurations:

$q_1 = use \wedge q_2 = use$

The set of bad configurations corresponds to configurations where two processes are in state *use*.

13.5 Experimental Results

Based on the approach introduced in this Part II, we have implemented a prototype [Rez08] and run it on all parameterized algorithms and protocols we were able to model. The prototype tool applies the method uniformally and mechanically on any parameterized system belonging to one of the three linear classes introduced in chapter 9. The tool starts from specifications similar to those used to introduce the examples in the previous sections of this chapter. The backwards reachability analysis is performed on constraints denoting upward closed sets of configurations.

We use in the current version of the prototype, the well known data structure of *Difference Bound Matrices* for manipulating numerical variables. Our prototype relied on a home made implementation of these data structures. This gave extremely encouraging results and permitted automatic verification of safety properties for most of the linear examples we tried. A distributed version of the mutual exclusion algorithm by Szymanski was the only case that generated a *false positive*. We report on the obtained results, using a 1.6 Ghz laptop with 1G of memory.

Recall that several "bad" constraints have sometimes been introduced for the same parameterized system. Taking the disjunction of the constraints may ease the calculations (intuitively, the backwards algorithm may save some iterations). For the cases where several "bad" constraints are stated, we present in table 13.14 those corresponding to the heaviest calculations in terms of number of constraints.

We state, for each one of the considered parameterized systems, whether the model at hand has been verified by our prototype. We give the required

	verified	iter	Φ	t	m
Burns algorithm	$\surd$	16	40	<0.05	< 5
Szymanski algorithm	$\surd$	61	5085	85	10
Dijkstra algorithm	$\surd$	18	89	<0.05	< 5
Java Lock algorithm	$\surd$	22	376	3.2	< 5
Illinois coherence	$\surd$	10	46	<0.05	<5
Dec Firefly coherence	$\surd$	14	31	0.1	<5
Futurebus coherence	$\surd$	10	46	0.1	<5
German coherence	$\surd$	34	10492	232	15
Racing Bakery algorithm	$\surd$	11	29	<0.05	<5
Bogus Bakery algorithm	$\times$	7	67	<0.05	<5
All interl. Bakery algorithm	$\surd$	13	42	0.1	<5
Distr. Lamport	$\surd$	30	4676	85	18
Distr. Ricart-Agrawala	$\surd$	32	1205	13	<5

Table 13.14: *For each linear parameterized system introduced in this chapter, we (fully automatically) check safety as defined by the constraints denoting bad configurations and introduced together with the system at hand. We state whether each system has been verified, and enumerate the number of iterations, the number of constraints (in the final set resulting from the fixpoint analysis), together with the required time in seconds and memory in megabytes. The prototype returns a counter example when checking the Bakery algorithm with a Bug (marked by $\times$).*

time in seconds, memory in megabytes, number of iterations[1], and number of constraints obtained as representation of the final upward closed (of bad configurations).

We are not aware of previous fully automatic verifications for protocols like German cache coherence, Java Meta Lock, Lamport Bakery algorithm, Lamport Distributed mutex, or the Ricart and Agrawala distributed algorithm.

[1]This number corresponds to the sum of the steps in in the following optimized backwards reachability scheme: first compute (until a fixpoint is reached) all predecessors with a minimal size (and save those generated with a larger size). Then repeat on the saved constraints with a larger size. This ensures that larger constraints will be used for computations if they are not to be eliminated by other constraints of a smaller size.

14. Conclusions

In this Part, we have presented a simple, light-weight, and efficient method permitting the automatic model checking of safety properties for systems which consist of an arbitrary number of processes. We uniformly apply our method on three different classes of linear parameterized systems. In the first class, the domains of the variables manipulated by the components are finite. In the second class, the variables can take numerical values. In the third class, the variables can also be arrays of linear and numerical variables.

Our method works in two steps. The first step consists in automatically generating an over-approximation for the transition system induced by the parameterized system at hand. A key notion in this automatic over-approximation is to enforce monotonicity. Together with some other requirements (effectiveness of some predecessor computations on upward closed sets, etc.) this allows the performance of the second step. In the second step we perform a backwards reachability analysis that starts from a set of bad configurations representing the violation of the safety property to be checked. The analysis is continued until a fixpoint is reached. The analysis on the approximate transition system is guaranteed to terminate (based on *well quasi-ordering* results) on approximate transition systems resulting from linear parameterized systems belonging to the first class. Based on the method, we have automatically model checked several non-trivial parameterized systems from each class we considered. In fact among the examples we considered for all the three classes, only one example generated a false positive. The method permitted the fully automatic verification of parameterized systems like the Bakery [Lam74] and the Distributed [Lam78] algorithms by Lamport, and the Distributed algorithm by Ricart and Agrawala [RA81].

Part III:
Reducing Termination to Reachability

Termination (liveness) properties are harder to verify than safety properties. For finite-state systems and some parameterized systems, safety properties can be verified by computing (some approximation of) the set of reachable states. Verifying liveness properties, requires at least a repeated search through the state space in enumerate model checkers [Hol97]. In symbolic model checkers, a natural but more expensive technique is to compute the transitive closure of the transition relation (see Part I). Multiple fairness requirements can make the situation even more complicated. For general infinite-state systems, the difference between safety and liveness properties is even larger. For some classes of systems, such as lossy channel systems, checking safety properties is decidable [AJ96b], whereas checking liveness properties is undecidable [AJ96a].

The general approach for proving liveness involves finding auxiliary assertions associated with well-founded ranking functions and helpful directions. Finding such ranking functions is not easy, and automation requires techniques adapted to specific data domains. The technique of finite-state abstractions has been difficult to apply when proving liveness properties since abstractions may introduce spurious loops that do not preserve liveness. A recent approach, namely the one of transition predicate abstraction [PR04], extends the framework of predicate abstraction and involves constructing an abstraction of the transition relation. The application of this approach to the case of parameterized systems is however still unclear.

This Part III assumes that the liveness property to be checked has been transformed to the property of termination for a system. This transformation is standard for many classes of liveness properties, including the so-called progress properties (of form $\Box(P \Longrightarrow \Diamond Q)$). Our approach proves termination for (parameterized and non-parameterized) concurrent systems with arbitrary (weak) fairness (aka justice) requirements; reachability analysis are the only non-trivial computations on the system. The approach consists in a main method

complemented by a (simple) complementary method. Neither of the proposed methods relies on finding an explicit ranking function or on computing the transitive closure of the transition relation. For a simple deterministic (non-concurrent) system, the set of states in which termination is guaranteed can be calculated as the set of states that are backwards reachable from some terminated state. We generalize this observation in the main method. In this method, a central technique for handling concurrency is the use of commutativity properties between different transitions of the system. The main method checks Termination by backwards reachability analysis, which computes the set of states that are backwards reachable from the set of terminated configurations under a particular transition relation, which we call a *convergence relation*. The main method is in general not complete. It computes an under-approximation of the set of configurations from which termination is guaranteed. If this under-approximation does not include the configurations for which one intends to prove termination, there are several ways to increase the power of the method. One way is to repeat the backwards reachability analysis, letting the computed under-approximation play the role of terminated configurations. One then exploits the fact that our convergence relation increases when the set of terminated configurations increases: a repeated reachability analysis will therefore improve the under-approximation. Another way is to apply other techniques (e.g. based on ranks or transitive closure computation) to prove termination for the remaining states of interest. Here, we propose a simple complementary method. Roughly, the second method relies on the following idea. If a finite number of weak fair actions (i.e. guaranteed to be infinitely often disabled or executed) are all disabled only in terminated configurations, and if each one of them can neither be disabled, nor executed twice without encountering a terminated configuration, then we are guaranteed (by fairness) to reach the set of terminated configurations.

Computing (or under-approximating) the set of (backwards) reachable states is conceptually easier than finding ranking functions or computing the transitive closure. Thus, liveness properties can be established for a class of systems, provided that there is a powerful way to compute sets of (backwards) reachable states. To show the usefulness of our approach, we apply it to examples of parameterized and non-parameterized infinite systems.

Outline

This last Part is organized as follows. We describe in chapter 15 fair parameterized systems and introduce basic notions for general infinite fair transition systems. Then we introduce our main and complementary methods. In chapter 16, we discuss the application of our approach on three representative termination problems for infinite-state systems. We conclude this Part in Chapter 17.

15. Fair Termination as Reachability

In this chapter, we present our approach for proving terminations using reachability analysis. The termination is stated for systems with arbitrary (weak) fairness requirements.

The approach consists in two methods, and is in general not complete. A main method computes an under-approximation of the set of configurations from which termination is guaranteed. One way to increase the power of the method is to repeat the backwards reachability analysis, using the computed under-approximation as terminated configurations. One can also use complementary proof methods; here, we present a complementary proof method which is particularly useful for parameterized systems. Both proof methods use reachability analysis as the only non-trivial computation on the system to be checked.

First, we introduce basic notions and fix some notations we will be using in the rest of this Part. Then, we give an overview of the main proof method and introduce the intuition behind it in section 15.2. We formalize this intuition and introduce the required calculations in section 15.3. In section 15.4, we present our complementary proof method. We conclude this chapter in section 15.5

15.1 Fair Transition Systems

We consider fair transition systems. Such a system may contain a set of actions with (weak) fairness requirements (aka justice), as in, e.g., UNITY [CM88]. Formally, a *fair transition system* FTS is a triple $(C, \longrightarrow, \mathcal{A})$, where :

- C is a set of *configurations*,
- $\longrightarrow \subseteq C \times C$ is a *transition relation* on C. We require that the identity relation is included in $\longrightarrow$.
- $\mathcal{A}$ is a finite or countable set of *fair actions* (each of which is a subset of $\longrightarrow$) required to be deterministic.

An *action* is any subset of the transition relation. We write $C \longrightarrow C'$ for $(c, c') \in \longrightarrow$. For an action α, we use $c \xrightarrow{\alpha} c'$ to denote $(c, c') \in \alpha$. An action α is *enabled* in a configuration c if there is some configuration c' such that $c \xrightarrow{\alpha} c'$. The set of configurations in which the action α is enabled is denoted $En(\alpha)$. If S is a set of configurations, then $\alpha \wedge S$ denotes the set of pairs (c, c') of configurations such that $c \xrightarrow{\alpha} c'$ and $c \in S$. For a set $\mathcal{B}$ of actions, let $\mathcal{B} \wedge S$ denote $\{\alpha \wedge S | \ \alpha \in \mathcal{B}\}$. A *computation* of FTS from a configuration $c \in S$ is

an infinite sequence of configurations $c_0\ c_1\ c_2\ \dots$ such that, (i) $c = c_0$; (ii) $c_i \longrightarrow c_{i+1}$ for each $i : 0 \leq i$; and (iii) for each fair action $\alpha \in \mathcal{A}$, there are infinitely many $i \geq 0$ where either $c_i \xrightarrow{\alpha} c_{i+1}$ or $c_i \notin En(\alpha)$.

Remark 15.1. (Fair parameterized systems) One way to introduce fairness for a parameterized system $\mathcal{P} = (Q, X, T)$ (see section 11.1), and on its induced transition system $(C, \longrightarrow)$, is the following. Fairness requirements are added by choosing a subset T_{Fair} of T. This results in a subset $\left\{ \xrightarrow{t} \mid t \in T_{Fair} \right\}$ of the transition relation $\longrightarrow$. This subset is taken to be the set $\mathcal{A}$ of fair actions. Observe that if each transition t in T_{Fair} is deterministic, then the set of fair actions $\mathcal{A}$ can (see remark 15.2) naturally be considered to be deterministic.

For a set S' of configurations and action α, let $Pre(\alpha, S')$ be the set of configurations c such that $c \xrightarrow{\alpha} c'$ for some $c' \in S'$. For a set of actions $\mathcal{B}$, let $Pre^*(\mathcal{B}, S')$ be the union of S' and the set of configurations c such that $c \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} c'$ for some $c' \in S'$ and $\alpha_1, \dots, \alpha_n \in \mathcal{B}$.

Termination

Let FTS be a program $(C, \longrightarrow, \mathcal{A})$ and $F \subseteq S$ be a set of *terminated* configurations. We assume F to be *stable*, i.e., that $c \in F$ and $c \longrightarrow c'$ implies $c' \in F$. Define $\Diamond F$ as the set of configurations c such that any computation of FTS from c contains a configuration in F. In other words, $\Diamond F$ is the set of configurations from which termination is guaranteed, in the sense that each computation from c will eventually reach F. In this Part III we present methods for computing (an under-approximation of) $\Diamond F$. We can also consider many classes of liveness properties, e.g., progress properties (of form $\Box(P \implies \Diamond Q)$), by first transforming them to termination properties. There exist standard techniques for such reductions. For example, a program satisfies $\Box(P \implies \Diamond Q)$ if $\Diamond Q$ includes configurations that can be reached from an initial configuration in a sequence of transitions that visits P, but have not yet visited Q.

Remark 15.2. (Non deterministic case) The restriction that each fair action is deterministic can often be circumvented by representing a nondeterministic action as a union of several deterministic ones.

Our definition of fair transition system does not mention initial configurations.

Remark 15.3. (initial configurations) When initial configurations are given, a typical use of our techniques will be to first compute the set of reachable configurations (or an over-approximation), and let them be the configurations of the program as defined above.

15.2 Intuition of the Main Proof Method

In this section, we give an overview of our main method for computing a (good) under-approximation of the set $\Diamond F$, where F is a set of configurations

of a program $FTS = (C, \longrightarrow, \mathcal{A})$. The inspiration for this method is the simple observation that when FTS is a deterministic program with only one action α that is fair, then $\Diamond F$ is the set $Pre^*(\alpha, F)$. Our goal is therefore a technique for proving termination and liveness properties, where the only nontrivial computation is a predecessor calculation, i.e., computing $Pre^*(\mathcal{B}, S)$ for some set of configurations S and actions $\mathcal{B}$.

Our method works by computing a so-called *convergence relation*, here denoted $\hookrightarrow_F$, on the configurations of FTS; this is a relation with the property that if $c \hookrightarrow_F \underline{c}$ and $\underline{c} \in \Diamond F$ then also $c \in \Diamond F$. From this property it follows that $Pre^*(\hookrightarrow_F, F) \subseteq \Diamond F$ for any convergence relation $\hookrightarrow_F$. The construction of $\hookrightarrow_F$ depends in general on F. Since $\hookrightarrow_F$ will be employed in a predecessor calculation, it is natural to allow the use of predecessor calculations also in the construction of $\hookrightarrow_F$ itself, but to avoid computations of transitive closures or other more powerful techniques.

Our main technique for constructing $\hookrightarrow_F$ uses a commutativity argument to infer that it satisfies the required properties. To explain its intuition, consider the following simple program, which consists of two deterministic processes executing in parallel.

$$
\begin{array}{llll}
\alpha_1 : & x := x - 1 & \text{if} & x > 0 \\
\alpha_2 : & y := y - 1 & \text{if} & y > 0
\end{array}
$$

Variables x and y assume values in the natural numbers. For $i = 1, 2$, process i repeatedly performs action α_i. Both α_1 and α_2 are fair actions. The transition relation is the union of both actions plus the identity relation. The set F of terminated configurations is the single configuration with $x = y = 0$.

In this example, our method computes $\hookrightarrow_F$ as $\alpha_1 \cup \alpha_2$. Our method implicitly ascertains that $\hookrightarrow_F$ is a convergence relation using a commutativity argument. To understand why α_1 is in $\hookrightarrow_F$, assume that $c \xrightarrow{\alpha_1} \underline{c}$ and $\underline{c} \in \Diamond F$. Consider any computation from c. If it goes first to $\underline{c}$ we are done. Otherwise, it first consists of a sequence of executions of action α_2. During this sequence, α_1 remains enabled, and so must eventually (by fairness) be executed, leading to some configuration $\underline{c}'$. Now observe that since α_1 and α_2 commute, $\underline{c}'$ is reachable from $\underline{c}$. Since $\underline{c} \in \Diamond F$ we infer, using the fact that $\Diamond F$ is a stable set, that $\underline{c}' \in \Diamond F$ and hence that $c \in \Diamond F$. We conclude that termination is guaranteed for all configurations in $Pre^*(\hookrightarrow_F, F)$, which here is the set of all configurations.

The above method can prove termination for many programs with a regular structure. It is in general incomplete. For programs where the above method computes a too small under-approximation of $\Diamond F$, we offer the following two ways to proceed.

The backwards reachability computation can be repeated several times. If one computation produces an under-approximation G of $\Diamond F$, the next application of our method will compute $\Diamond G$ using a convergence relation $\hookrightarrow_G$ that

is larger than in the first computation, since it depends on G instead of F. Let us illustrate this by altering the above program by changing the guard of α_1 into $0 < x \leq y \vee y = 0$. This destroys commutativity between α_1 and α_2 in case $y = x$. However, a first backwards reachability computation will produce the set G consisting of configurations with $0 \leq x \leq 1$ or with $0 \leq y < x$ as an under-approximation to $\Diamond F$. A second backwards reachability computation thereafter reveals that all configurations are in $\Diamond G$, hence also in $\Diamond F$.

15.3 Main Proof Method

In this section, we formalize the methods for calculating (an under-approximation of) the set $\Diamond F$ by backwards reachability analysis, presented in the previous section. We first present the general approach, and then our main technique.

Assume a fair transition system $(C, \longrightarrow, \mathcal{A})$. Let F be a stable set of terminated configurations. Define a *convergence relation* on S for F to be a relation $\hookrightarrow_F$ on S such that whenever $c \hookrightarrow_F \underline{c}$ and $\underline{c} \in \Diamond F$ then also $c \in \Diamond F$. The point of convergence relations is that if $\hookrightarrow_F$ is a convergence relation for F, then $Pre^*(\hookrightarrow_F, F) \subseteq \Diamond F$, i.e., we can use predecessor calculation to prove that termination is guaranteed from a set of configurations. Larger convergence relations allow to prove termination for larger sets of configurations. Furthermore, even if we cannot precisely calculate $Pre^*(\hookrightarrow_F, F)$, any under-approximation of this set is also in $\Diamond F$.

To apply these ideas, we need techniques to compute sufficiently powerful convergence relations. Any number of convergence relations can be combined into one, since the union of two convergence relations is again a convergence relation. Now we present our main technique, which is based on a commutativity argument. Fix for the rest of this section a deterministic fair action α, and a set of configurations F.

Definition 15.1. *(Left moving configurations) Define the* left moving configurations for (α, F)*, denoted* $Left(\alpha, F)$*, as the set of configurations c satisfying: whenever there are configurations* $\underline{c}, c'$ *with* $c' \notin F$ *such that* $c \longrightarrow \underline{c} \xrightarrow{\alpha} c'$*, then there is a configuration* $\underline{c}'$ *such that* $c \xrightarrow{\alpha} \underline{c}' \longrightarrow c'$.

Intuitively, α can "move left" of $\longrightarrow$, and still reach the same configuration. The definition is illustrated in Figure 15.1.

We make use of the set of left moving configurations for action α and set F of configurations to introduce another set we use in our main method, and defined below.

Definition 15.2. *(Helpful configurations) Define the* α-helpful configurations, *denoted* $Helpful(\alpha, F)$*, as the largest set S of configurations such that* $S \subseteq ((En(\alpha) \cap Left(\alpha, F)) \cup F)$*, and whenever* $c \in S$ *and* $c \longrightarrow c'$ *then either* $c \xrightarrow{\alpha} c'$*, or* $c' \in F$*, or* $c' \in S$.

$$\forall c', \underline{c} \qquad \begin{array}{ccc} \exists \underline{c}' & \longrightarrow & c' \notin F \\ \uparrow \alpha & & \uparrow \alpha \\ c & \longrightarrow & \underline{c} \end{array}$$

Figure 15.1: A configuration c is in $\mathit{Left}(\alpha, F)$ if whenever there are configurations $\underline{c}, c'$ with $c' \notin F$ such that $c \longrightarrow \underline{c} \xrightarrow{\alpha} c'$, then there is a configuration $\underline{c}'$ such that $c \xrightarrow{\alpha} \underline{c}' \longrightarrow c'$.

Intuitively, a configuration is α-helpful if the properties that α is enabled and left moving remain true when any sequence of transitions not in α are taken, unless F is reached. The above concepts can be used to define a convergence relation as follows.

Theorem 15.1. *Let α be a fair action of $(C, \longrightarrow, \mathcal{A})$ and F be a stable set of configurations. Then the relation $\stackrel{\alpha}{\hookrightarrow}_F$, defined by:*

$$\stackrel{\alpha}{\hookrightarrow}_F \;\equiv\; \alpha \wedge \mathit{Helpful}(\alpha, F)$$

is a convergence relation for F.

Proof. Assume that $c \stackrel{\alpha}{\hookrightarrow}_F \underline{c}$ and $\underline{c} \in \Diamond F$. Consider any computation $c_0\ c_1\ c_2\ \ldots$ from $c = c_0$. We show that it contains a configuration in F.

- If there is a k with $c_k \in F$ we are done.
- Otherwise, if there is a k with $c_k \xrightarrow{\alpha} c_{k+1}$, let k be the least such index. By induction, using the definition of $\mathit{Helpful}(\alpha, F)$, we infer that $c_i \in \mathit{Helpful}(\alpha, F)$, hence $c_i \in \mathit{En}(\alpha)$ and $c_i \in \mathit{Left}(\alpha, F)$ for $i = 0, \ldots, k$. Let $\underline{c}_i$ be the unique configuration with $c_i \xrightarrow{\alpha} \underline{c}_i$, in particular $c_{k+1} = \underline{c}_k$. By induction we infer, using the definition of $\mathit{Left}(\alpha, F)$, that $\underline{c}_i$ is reachable from $\underline{c}$ for all i with $0 \leq i \leq k$. In particular, $c_{k+1} = \underline{c}_k$ is reachable from $\underline{c}$. From $\underline{c} \in \Diamond F$ and the stability of F, we infer $c_{k+1} \in \Diamond F$ and hence the computation must contain a configuration in F. An illustration of this argument is provided in Figure 15.2.
- Otherwise, we infer by induction over k, using $c \in \mathit{Helpful}(\alpha, F)$, that α is enabled in all configurations of the computation. By fairness, α will eventually be executed, and we are back to the previous case.

□

Corollary 15.1. $\mathit{Pre}^*\left(\left\{\stackrel{\alpha}{\hookrightarrow}_F \mid \alpha \in \mathcal{A}\right\},\ F\right) \subseteq \Diamond F$

In order to show how termination can be proven by backwards reachability analysis, we must finally explain how to compute $\mathit{Helpful}(\alpha, F)$, or an underapproximation of it, by backwards reachability analysis. We first observe that:

$$\mathit{Left}(\alpha, F) = \neg \mathit{Pre}((\longrightarrow \circ\, \alpha) \,/\, (\alpha \circ \longrightarrow), \neg F)$$

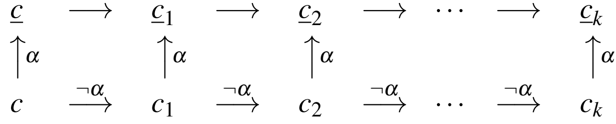


Figure 15.2: $(c,\underline{c}) \in \overset{\alpha}{\hookrightarrow}_F$. The α-successor of any successor of c, is either a successor of $\underline{c}$, or in F.

Proposition 15.1. *The set Helpful* (α,F) *is the complement of the set*

$$Pre^*\left((\longrightarrow/\alpha) \wedge \neg F, (\neg Left(\alpha,F) \cup \neg En(\alpha)) \cap \neg F\right)$$

Proof. According to Definition 15.2, a configuration c is not in $Helpful(\alpha,F)$ if and only if there is a sequence of transitions from c, none of which is in α or visits a configuration in F, which leads to a configuration neither in F nor in $En(\alpha) \cap Left(\alpha,F)$; exactly what the proposition formalizes. □

15.4 A Complementary Termination Method

In this section, we present a proof method we use to complement the method in section 15.2. We applied this complementary method on the results obtained by the main proof rule in order to conclude termination for non-trivial examples of parameterized systems. The simplicity of this complementary method speaks in favor of the usefulness of the main proof method.

This method is particularly useful for parameterized systems, where termination is often achieved by letting a selected subset of the processes execute a fixed sequence of actions. The proof method assumes we select a finite number $\mathcal{B}$ of fair actions from the fair transition system at hand. For a parameterized system the finite set of fair actions $\mathcal{B}$ is typically defined using a subset of the set T_{Fair} as explained in Remark 15.1. Observe that for such systems, the set $\mathcal{B}$ is unboundedly large, but still finite.

The idea is to establish that a configuration c is in $\Diamond F$ if computations starting at c satisfy the following three conditions: (i) whenever an action in $\mathcal{B}$ is enabled, it remains so until executed or F is reached; and (ii) each action in $\mathcal{B}$ may be executed at most once before F is reached; finally (iii) the set F includes all configurations where the actions in $\mathcal{B}$ are all disabled. Intuitively, the number of actions in $\mathcal{B}$ which have not yet been executed implicitly defines a *ranking function*, and corresponds to a distance (to F) that can not increase and is bounded to decrease by fairness.

Let us define the involved properties formally. Fix a fair transition system $(C,\longrightarrow,\mathcal{A})$. Let F be a set of terminated configurations, and α be a fair deterministic action. We define two sets of configurations we use in the proof method.

Definition 15.3. *(Persistent configurations) Define the* α-persistent configurations*, denoted by* $Persist(\alpha, F)$*, as the largest set* S *of configurations such that* $S \subseteq (En(\alpha) \cup F)$*, and whenever* $c \in Persist(\alpha, F)$ *and* $c \longrightarrow c'$ *then either* $c \xrightarrow{\alpha} c'$*, or* $c' \in F$*, or* $c' \in S$*.*

Intuitively, $Persist(\alpha, F)$ corresponds to the condition that if action α is enabled, it remains so until executed; The set $Persist(\alpha, F)$ can be computed (like for $Helpful(\alpha, F)$ in proposition 15.1) as the complement of the set:

$$Pre^* ((\longrightarrow / \alpha) \wedge \neg F, \neg En(\alpha) \cap \neg F)$$

Definition 15.4. *(Twice* α *outside* F*) Define* $Twice(\alpha, F)$ *to be the set containing all configurations* c *for which a computation* $c_0 c_1 \ldots$ *starting at* c *verifies both conditions: (i) there exist* $i < j$ *with* $c_i \xrightarrow{\alpha} c_{i'}$ *and* $c_j \xrightarrow{\alpha} c_{j'}$*; and (ii)* $c_k \notin F$ *for each* $0 : 1 \leq k \leq j$*.*

Intuitively, $Twice(\alpha, F)$ characterizes the complement of the set of configurations from which α may be executed at most once before F. It is easy to see that this set coincides with S_5 defined by $S_5 = Pre^* (\longrightarrow \wedge \neg F, S_4)$, and $S_4 = Pre(\alpha \wedge \neg F, S_3)$, $S_3 = Pre^* (\longrightarrow \wedge \neg F, S_2)$, $S_2 = Pre(\alpha \wedge \neg F, S_1)$, $S_1 = Pre^* (\longrightarrow \wedge \neg F, S_0)$, with $S_0 = \neg F$. Observe there are three reachability calculations here.

Finally we capture the last condition using the set $After(\mathcal{B}, F)$ and defined by:

Definition 15.5. *(Disabled entails termination) For a set of fair actions* $\mathcal{B}$*, the set* $After(\mathcal{B}, F)$ *contains all configurations* c *such that for any computation* $c_0 c_1 \ldots$ *starting at* c*, if all actions in* $\mathcal{B}$ *are disabled in* c_i*, then* $c_i \in F$*.*

This set can be obtained as the complement of the set :

$$Pre^* \left(\longrightarrow \wedge \neg F, \neg F \cap \bigcap_{\alpha \in \mathcal{B}} (\neg En(\alpha)) \right)$$

Care must be taken to handle the parameters correctly when performing the predecessor calculations. Now we state the termination proof method.

Theorem 15.2. *Let* $\mathcal{B}$ *be a finite set of fair actions of* $(C, \longrightarrow, \mathcal{A})$*, and let* F *be a set of configurations in* S*. Then*

$$\left[After(\mathcal{B}, F) \cap \bigcap_{\alpha \in \mathcal{B}} (\neg Twice(\alpha, F) \cap Persist(\alpha, F)) \right] \subseteq \Diamond F$$

Proof. Let c be a configuration in the set defined by the left-hand side. Consider a computation from c. Assume that it contains no configuration in F. Then, since $c \in After(\mathcal{B}, F)$ it also contains no configuration in which all actions in $\mathcal{B}$ are disabled. This means that at any configuration in the computation, some action α is enabled. Since $c \in Persist(\alpha, F)$ the action α will remain

enabled until it is executed, and thereafter (since $c \in \neg Twice(\alpha, F)$) never be executed again. This implies that after a finite number of computation steps, all actions in $\mathcal{B}$ have been executed. This contradicts the previous conclusion that thereafter some action in $\mathcal{B}$ is enabled, and will eventually be executed. $\square$

15.5 Conclusions

We have presented in this chapter a method that applies for general transition systems to compute configurations for which termination is guaranteed under weak fairness. The method combines commutativity and fairness arguments in order to obtain termination. Our main method is in general incomplete. In case the set of configurations identified as terminating is not large enough, one could repeatedly apply the method. Another way is to use a complementary proof rule. We presented such a proof rule that is particularly tailored for parameterized systems. In the next chapter, we discuss the application of these methods on particular termination problems.

16. Applications on Infinite Systems

We show the usefulness of our approach and apply it to prove fair termination on three examples, each from a different family of infinite systems. As a first example, we look at the parameterized mutual exclusion algorithm of Szymanski and consider proving the property of starvation-freedom for it. Termination of an integer program constitutes the second example. In the third example, we show progress for a system (namely the *alternating bit* protocol) composed of finite-state machines communicating over unbounded lossy FIFO channels.

16.1 Parameterized systems

We briefly introduce how to use the methods in chapter 15 to verify individual starvation freedom for the mutual exclusion algorithm of Szymanski (see section 13.1).

Recall that the algorithm of Szymanski can be viewed as a parameterized system. The required calculations have been performed on the regular model checking tool [Nil02]. In this tool, automata and transducers are used to manipulate sets of configurations and transition (chapter 6). We use in the following regular expressions to represent sets of configurations. We take advantage of the possibility, in this framework, to compute the set of reachable configurations, and propose (table 16.1) a state-based version of the algorithm. In this version, components have no local variables. Their states range over $\{1,\ldots,7\}$, with state (7) corresponding to the critical section.

In the following, we associate a constant bit to each process (see section 6.1). We set this bit non deterministically to one of the processes in the system, and show starvation freedom for it. This makes sure the property is individual and not communal. We represent the states of the processes for which the bit is set by $\underline{1},\ldots,\underline{7}$.

Table 16.1: State Based Algorithm of Szymanski.

Szymanski mutual exclusion algorithm: in this state based formulation, components take states in $0,\ldots,7$ with 7 modeling the critical section.

States:

$Q = \{1,\ldots,7\}$ originally 1

Transition rules:

$t_1 :$	1	$\rightarrow$	2	$\triangleright$	$\forall_{LR}(124)$
$t_2 :$	2	$\rightarrow$	3	$\triangleright$	$true$
$t_3 :$	3	$\rightarrow$	4	$\triangleright$	$\forall_{LR}(34)$
$t_4 :$	3	$\rightarrow$	5	$\triangleright$	$\exists_{LR}(125)$
$t_5 :$	4	$\rightarrow$	5	$\triangleright$	$\exists_{LR}(5)$
$t_6 :$	5	$\rightarrow$	6	$\triangleright$	$\forall_{LR}(1567)$
$t_7 :$	6	$\rightarrow$	7	$\triangleright$	$\forall_{L}(1)$
$t_8 :$	7	$\rightarrow$	1	$\triangleright$	$true$

Final configurations:

$\underline{F} = \{1,5,6,7\}^* \cdot \{\underline{7}\} \cdot \{1,5,6,7\}^*$

In this system, all transitions (except t_1) are fair. We fix $\underline{F}$ to be the set $\{1,5,6,7\}^* \cdot \underline{7} \cdot \{1,5,6,7\}^*$, where a non deterministically chosen process is at its critical section (7). We are interested in computing the set of configurations from which this process is guaranteed to access its critical section, i.e., to reach $\underline{F}$.

For the algorithm of table 16.1 above, this set corresponds to all reachable configurations in $\{1,...,7\}^* \cdot \underline{q} \cdot \{1,...,7\}^*$ where q is in $\{2,3,4,5,6,7\}$. For this example, our main proof rule computes a strict under-approximation of the set $\Diamond\underline{F}$. We complete the calculations, and obtain the totality of the set $\Diamond\underline{F}$ using the complementary proof rule of section 15.4.

A verification method based on Corollary 15.1 and Theorem 15.2 was implemented in the framework of Regular Model Checking [AJNS04], and applied to a number of well-known parameterized mutual exclusion protocols.

Verification Procedure

For each protocol, we have modeled $\underline{F}$ as the set of states where a process is in its critical section. We have computed an under-approximation $\underline{G}$ of $\Diamond\underline{F}$ using the method of section 15.3, and thereafter applied the complementary rule described in section 15.4 to compute $\Diamond\underline{G}$. To ensure that predecessors are reachable states, we computed the set of (forwards) reachable states, and restricted the actions to it.

In the experiments, the choice of which rule to apply, and according to what sequence, was not automatized. However, the approach may be fully automated by e.g. applying the rules alternatively. Concerning starvation freedom for the algorithm of Szymanski, three successive applications of Corollary 15.1

establish starvation freedom for almost all system configurations where an individual process is waiting, i.e., not at state (1). However, Corollary 15.1 cannot prove starvation freedom for system states where processes at state (1) coexist with processes at state (2). The reason for this is that action t_2 may disable the actions t_1, thereby destroying commutativity. By using also one application of Theorem 15.2, starvation freedom is shown for each process, from all reachable system configurations, where the process is waiting.

16.2 Other Applications

We prove termination for non-parameterized infinite systems. In this section we illustrate our main proof method by applying it on examples from two different families of systems. The first family of systems we consider is the one of integer programs. Programs in this family may be regarded as some extended finite-state machines manipulating a finite number of numerical variables (observe variables values make the state space infinite in general). Termination of such a program typically depends on the values of the manipulated variables. We also use the method on the well-known *Alternating Bit* protocol as a member of the second family. Here, finite-state machines communicate over lossy FIFO channels. The state space of this protocol is infinite because the channels are assumed to be unbounded.

The necessary reachability calculations for all described examples in this section are made by hand. These calculations are however simple, and should be easily carried out with exact tools like those developed in [BLP06, BG99].

16.2.1 Unbounded Variables: an Integer Program

Consider the program written, in the action-based syntax of the example in section 15.2, as follows.

$$\begin{array}{lllll} \alpha_1 & : & y := y+1 & \text{if} & x=1 \\ \alpha_2 & : & x := 0 & \text{if} & \mathit{true} \\ \alpha_3 & : & y := y-1 & \text{if} & x=0 \wedge y>0 \end{array}$$

Variable y takes its values in the natural numbers, and variable x takes its values in $\{0,1\}$. Both α_2 and α_3 are fair actions. The transition relation is the union of all three actions plus the identity relation. The set F of terminated states is the single state with $x = y = 0$. It is well-known that a standard termination proof for this program will require a ranking function whose range is larger than the natural numbers. This suggests that we need at least two iterations of our method to compute the set $\Diamond F$. We describe each iteration below.

In the first iteration we compute $\mathit{Helpful}(\alpha_2, F)$ and $\mathit{Helpful}(\alpha_3, F)$ (α_1 is not a fair action). These computations are summarized in the table below.

	$En(\alpha_i)$	$\mathit{Left}(\alpha_i, F)$	$\mathit{Helpful}(\alpha_i, F)$
α_2	*true*	$x = 0$	$x = 0$
α_3	$x = 0 \wedge y > 0$	$x = 0 \vee y = 0 \vee y = 1$	$x = 0$

We explain the entries of the table for α_2. The corresponding entries for α_3 can be explained in a similar manner. The set $\mathit{Left}(\alpha_2, F)$ includes all states c where $x = 0$. This is since either (i) $y = 0$ in which case $c \in F$; or (ii) $y > 0$, which means that α_1 is not enabled, and α_2 commutes with α_3. On the other hand, $\mathit{Left}(\alpha_2, F)$ does not include any state c with $x = 1$, as follows. We have $c \xrightarrow{\alpha_1} \xrightarrow{\alpha_2} c'$, for some c' with $y > 0$. Obviously, $c' \notin F$ and furthermore it is not the case that $c \xrightarrow{\alpha_2} \xrightarrow{\alpha_1} c'$ since α_2 disables α_1. This means we have violated the condition for being a left mover.

The set $\mathit{Helpful}(\alpha_2, F)$ includes all states where $x = 0$; such a state c belongs to $\mathit{Left}(\alpha_2, F)$. The action α_2 is enabled from c. Furthermore, the action α_1 is disabled, while the execution of α_3 from c again leads to a state satisfying $\mathit{Helpful}(\alpha_2, F)$.

By Corollary 15.1, the following set is in $\Diamond F$:

$$G \equiv \mathit{Pre}^*\left((\alpha_2 \wedge \mathit{Helpful}(\alpha_2, F)) \cup (\alpha_3 \wedge \mathit{Helpful}(\alpha_3, F)), F\right) \quad \equiv \quad x = 0$$

In the second iteration we compute $\mathit{Helpful}(\alpha_i, G)$ for $i = 2, 3$ in the same way. The interesting difference is that $\mathit{Left}(\alpha_2, G)$, which is *true*, is larger than $\mathit{Left}(\alpha_2, F)$, since any execution of α_2 leads to G. Hence also $\mathit{Helpful}(\alpha_2, G)$, which is *true*, is larger than $\mathit{Helpful}(\alpha_2, F)$.

	$En(\alpha_i)$	$\mathit{Left}(\alpha_i, G)$	$\mathit{Helpful}(\alpha_i, G)$
α_2	*true*	*true*	*true*
α_3	$x = 0 \wedge y > 0$	*true*	$x = 0$

By Corollary 15.1, the following set is in $\Diamond G$, hence in $\Diamond F$:

$$\mathit{Pre}^*\left((\alpha_2 \wedge \mathit{true}) \cup (\alpha_3 \wedge \mathit{Helpful}(\alpha_3, F)), G\right) \quad \equiv \quad \mathit{true}$$

16.2.2 Unbounded Channels: Alternating Bit Protocol

This protocol consists of finite-state processes that communicate over unbounded and lossy FIFO channels. It is decidable whether such a protocol satisfies a safety property [AJ96b], but undecidable whether a protocol satisfies a liveness property [AJ96a]. Using our technique, we can prove liveness properties for some of these protocols.

The alternating bit protocol involves a *sender* and a *receiver* that communicate over two channels c_M and c_A. Channel c_M is used to transmit messages

from the *sender* to the *receiver*, and channel c_A to transmit acknowledgments from the *receiver* to the *sender*. Both channels are FIFO and faulty in the sense that messages may be lost but not reordered. The purpose of the protocol is to transmit messages from the *sender* to the *receiver* in correct order, in spite of the fact that the channels can lose messages. Corruption of messages can also be taken into account by modeling it as a loss (some mechanism will detect and discard a corrupted message). Each channel is "fair" in the sense that if infinitely many messages are input, then infinitely many messages will be delivered.

We describe the operations of *sender* and *receiver* in the protocol. At one end of the channels, the *sender* constructs a message m_b by adding a sequence number b in $\{0,1\}$ to a pending message m, and sends it on the channel c_M to the *receiver*. The *sender* waits for an acknowledgment a_b with the same sequence number on the channel c_A. If a_b arrives, the procedure is repeated with the next pending message but with an inverted sequence number $(1-b)$. If no acknowledgment a_b arrives within a specified time period the *sender* retransmits the message m_b. Retransmissions are repeated until an acknowledgment a_b with a corresponding sequence number arrives. Acknowledgments with non-corresponding sequence numbers are discarded. On the other end of the channels, the *receiver* receives messages m_b from the incoming channel c_M. A message m_b is delivered if the corresponding sequence number b was expected. After delivery of m_b, the *receiver* sends on channel c_A an acknowledgment a_b with the same sequence number to the *sender*. The *receiver* expects a message with an inverted sequence number $(1-b)$. Messages with non-expected sequence numbers are discarded.

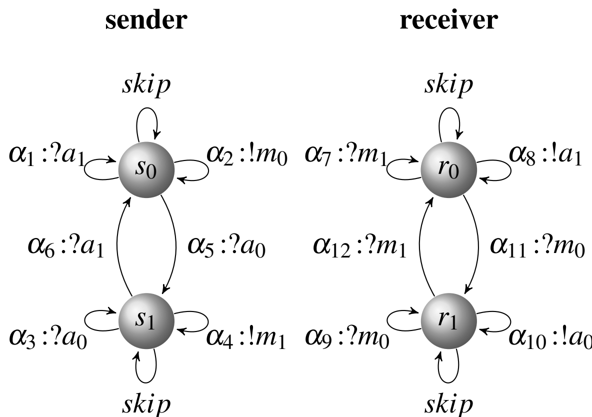


Figure 16.1: The alternating bit protocol involves a *sender* and a *receiver* that communicate over two unbounded, FIFO and lossy channels. The purpose of the protocol is to transmit messages from the *sender* to the *receiver* in correct order, in spite of the fact that the channels can lose messages. Each channel is "fair" in the sense that if infinitely many messages are input, then infinitely many messages will be delivered.

The *sender* and the *receiver* are modeled by the finite-state processes depicted in figure 16.1. The states of the *sender* are in $\{s_0, s_1\}$, while those of the *receiver* are in $\{r_0, r_1\}$. A state of the system is of form $s\,r\,(w_M, w_A)$ where s is a sender state, r is a receiver state, w_M is the content of channel c_M, and w_A is the content of channel c_A. The initial state is $s_0r_0(\langle\rangle, \langle\rangle)$ with both channels empty. From a state $s\,r\,(w_M, w_A)$, the effects of the actions $!m_b$ and $!a_b$, with b in $\{0,1\}$, are respectively $s\,r\,(m_b \cdot w_M, w_A)$ and $s\,r\,(w_M, a_b \cdot w_A)$ ($\cdot$ is the concatenation operator for channel content). The state $s\,r\,(w_M, w_A)$ results from applying the action $?m_b$ to the state $s\,r\,(w_M \cdot m_b, w_A)$, or from applying the action $?a_b$ to the state $s\,r\,(w_M, w_A \cdot a_b)$. Here, c_M and c_A are perfect FIFO buffers, and message losses are modeled as a non-deterministic choice between a send and a *skip* action.

There are techniques to automatically calculate the set of states reachable from the initial state $s_0r_0(\langle\rangle, \langle\rangle)$. An example is to start from the initial state and to apply the technique of *loop-first search* [BG99]. This technique generates the set of reachable states by taking all possible transitions, and evaluating (whenever possible) the effect of performing an arbitrary number of the same transition. Examples of such *accelerations* are α_1^* or α_2^*, resulting in the addition to the tail of c_M, respectively the subtraction from the head of c_A, of an arbitrary number of m_0, respectively a_1. Sets of states can be captured by *Queue-content Decision Diagram (QDD)* of the form $s\,r\,(w_M, w_A)$ where w_M and w_A are regular languages. The search stops once the set of generated states stabilizes, i.e. no new states are generated when applying transitions. For the protocol at hand, this technique returns the set of reachable states as union of the four sets $s_0r_0(m_0^*m_1^*, a_1^*)$, $s_0r_1(m_0^*, a_0^*a_1^*)$, $s_1r_0(m_1^*, a_1^*a_0^*)$, and $s_1r_1(m_1^*m_0^*, a_0^*)$.

We describe the program $(C, \longrightarrow, \mathcal{A})$ corresponding to the *alternating bit protocol*. The set S is here chosen to be the set of reachable configurations computed above. The transition relation $\longrightarrow$ is the union of the actions *skip* and $\alpha_1, \ldots, \alpha_{12}$. All these actions, except *skip*, are in $\mathcal{A}$. This corresponds to the assumption that if a message is continuously retransmitted, then eventually one of the messages is not lost.

We use the technique defined in section 15.3 to prove the following four progress properties of the protocol.

$$
\begin{array}{lcrcl}
P_{r_0r_1} & : & s_0r_0(m_0^*m_1^*, a_1^*) & \subseteq & \Diamond s_0r_1(m_0^*, a_0^*a_1^*) \\
P_{s_0s_1} & : & s_0r_1(m_0^*, a_0^*a_1^*) & \subseteq & \Diamond s_1r_1(m_1^*m_0^*, a_0^*) \\
P_{r_1r_0} & : & s_1r_1(m_1^*m_0^*, a_0^*) & \subseteq & \Diamond s_1r_0(m_1^*, a_1^*a_0^*) \\
P_{s_1s_0} & : & s_1r_0(m_1^*, a_1^*a_0^*) & \subseteq & \Diamond s_0r_0(m_0^*m_1^*, a_1^*)
\end{array}
$$

Observe that property $P_{r_0r_1}$ implies that from any state in $s_0r_0(m_0^*m_1^*, a_1^*)$, the system is guaranteed to reach a state in $s_0r_1(m_0^*, a_0^*a_1^*)$. This means the *receiver* changed state from r_0 to r_1. In other words, the receiver is guaranteed to take action α_{11} and to receive the message m_0. A similar reasoning with $P_{s_0s_1}$, $P_{r_1r_0}$ and $P_{s_1s_0}$ ensures *sender* and *receiver* indefinitely alternate sending m_0, a_0, m_1

α_i	$En(\alpha_i)$	$En(\alpha_i) \cap Left(\alpha_i,F)$	$Helpful(\alpha_i,F)$
α_1	$s_0r_0(m_0^*m_1^*,a_1^+)$	$s_0r_0(m_0^*m_1^*,a_1^+)$	$s_0r_0(m_0^*m_1^*,a_1^+) \cup s_0r_1(m_0^*,a_0^*a_1^*)$
α_2	$s_0r_0(m_0^*m_1^*,a_1^*)$	$s_0r_0(m_0^*m_1^*,a_1^*)$	$s_0r_0(m_0^*m_1^*,a_1^*) \cup s_0r_1(m_0^*,a_0^*a_1^*)$
α_3	$s_1r_0(m_1^*,a_1^*a_0^+)$ $\cup\, s_1r_1(m_1^*m_0^*,a_0^+)$	$s_1r_0(m_1^*,a_1^*a_0^+)$ $\cup\, s_1r_1(m_1^*m_0^*,a_0^+)$	$s_1r_0(m_1^*,a_1^*a_0^+) \cup s_1r_1(m_1^*m_0^*,a_0^+)$ $\cup\, s_0r_1(m_0^*,a_0^*a_1^*)$
α_4	$s_1r_0(m_1^*,a_1^*a_0^*)$ $\cup\, s_1r_1(m_1^*m_0^*,a_0^*)$	$s_1r_0(m_1^*,a_1^*a_0^*)$ $\cup\, s_1r_1(m_1^*m_0^*,a_0^*)$	$s_1r_1(m_1^*m_0^*,a_0^*) \cup s_0r_1(m_0^*,a_0^*a_1^*)$
α_5	$\emptyset$	$\emptyset$	$s_0r_1(m_0^*,a_0^*a_1^*)$
α_6	$s_1r_0(m_1^*,a_1^+)$	$\emptyset$	$s_0r_1(m_0^*,a_0^*a_1^*)$
α_7	$s_0r_0(m_0^*m_1^+,a_1^*)$ $\cup\, s_1r_0(m_1^+,a_1^*a_0^*)$	$s_0r_0(m_0^*m_1^+,a_1^*)$ $\cup\, s_1r_0(m_1^+,a_1^*a_0^*)$	$s_0r_0(m_0^*m_1^+,a_1^*) \cup s_1r_0(m_1^+,a_1^*a_0^*)$ $\cup\, s_0r_1(m_0^*,a_0^*a_1^*)$
α_8	$s_0r_0(m_0^*m_1^*,a_1^*)$ $\cup\, s_1r_0(m_1^*,a_1^*a_0^*)$	$s_0r_0(m_0^*m_1^*,a_1^*)$ $\cup\, s_1r_0(m_1^*,a_1^*a_0^*)$	$s_0r_0(m_0^*m_1^*,a_1^*) \cup s_1r_0(m_1^*,a_1^*a_0^*)$ $\cup\, s_0r_1(m_0^*,a_0^*a_1^*)$
α_9	$s_1r_1(m_1^*m_0^+,a_0^*)$	$s_1r_1(m_1^*m_0^+,a_0^*)$	$s_1r_1(m_1^*m_0^+,a_0^*) \cup s_0r_1(m_0^*,a_0^*a_1^*)$
α_{10}	$s_1r_1(m_1^*m_0^*,a_0^*)$	$s_1r_1(m_1^*m_0^*,a_0^*)$	$s_0r_1(m_0^*,a_0^*a_1^*)$
α_{11}	$s_0r_0(m_0^+,a_1^*)$	$s_0r_0(m_0^+,a_1^*)$	$s_0r_0(m_0^+,a_1^*) \cup s_0r_1(m_0^*,a_0^*a_1^*)$
α_{12}	$s_1r_1(m_1^+,a_0^*)$	$\emptyset$	$s_0r_1(m_0^*,a_0^*a_1^*)$

Figure 16.2: Alternating bit protocol. Calculation of $\Diamond s_0r_1(m_0^*,a_0^*a_1^*)$

and a_1. We show in the following how to prove the property $P_{r_0r_1}$; the other properties can be proven similarly.

Let $F \triangleq s_0r_1(m_0^*,a_0^*a_1^*)$. We use the technique defined in section 15.3 to calculate a set of states included in $\Diamond F$. To ensure the stability of F, we first modify all actions α to $\neg F \wedge \alpha$. The sets where the actions are enabled are shown in Figure 16.2. Observe that $\neg F \wedge \alpha_5$ is empty. The results of the computations (according to Proposition 15.1) of the helpful set of states for each fair action α_i in $\mathcal{A}$ appear in the same figure. Let us give an intuition of why $Helpful(\alpha_7,F)$ equals the union of the sets $s_0r_0(m_0^*m_1^+,a_1^*)$, $s_1r_0(m_1^+,a_1^*a_0^*)$ and $s_0r_1(m_0^*,a_0^*a_1^*)$. For every state c in this union, it is the case that either (1) c is in $F = s_0r_1(m_0^*,a_0^*a_1^*)$; or (2) c is in the union of $s_0r_0(m_0^*m_1^+,a_1^*)$ and $s_1r_0(m_1^+,a_1^*a_0^*)$. In the second case, observe that α_7 is enabled and that the only action that does not commute with α_7 is action α_{11} (which is not enabled). We have that (1) α_7 is enabled from c and commutes with any other enabled action; and (2) the execution of any other action from c leads to the same union of $s_0r_0(m_0^*m_1^+,a_1^*)$ and $s_1r_0(m_1^+,a_1^*a_0^*)$. Observe that α_7 is not enabled outside the union of $s_0r_0(m_0^*m_1^+,a_1^*)$, $s_1r_0(m_1^+,a_1^*a_0^*)$ and $s_0r_1(m_0^*,a_0^*a_1^*)$.

By Corollary 15.1, we have $G \triangleq Pre^* \left(\left\{ \overset{\alpha_i}{\hookrightarrow}_F \mid i = 1..12 \right\}, F \right) \subseteq \Diamond F$. Observe that $s_0 r_0 (m_0^* m_1^*, a_1^*) = Pre^* \left(\left\{ \overset{\alpha_i}{\hookrightarrow}_F \mid i = 2, 7, 11 \right\}, s_0 r_0 (m_0^*, a_1^*) \right) \subseteq G$. We therefore conclude that $s_0 r_0 (m_0^* m_1^*, a_1^*) \subseteq \Diamond F$. $\square$

16.3 Conclusions

We have applied our methods on a number of infinite-state systems, in order to compute a set of terminating configurations. While the complementary method is tailored for parameterized systems, the main method may be applied to general fair transition systems. The method shows an interesting a potential, and we need to try more examples in order to evaluate more accurately its applicability, and the possibilities for including it in an automatic scheme for checking liveness in general and termination in particular. The existence of exact methods for performing efficient reachability computations would be the first criteria to investigate new applications.

17. Conclusions

We have presented a method for proving liveness and termination properties of fair concurrent programs using backwards reachability analysis. The method uses neither computation of transitive closure nor explicit construction of ranking functions and helpful directions, and relies instead on showing certain commutativity properties between different actions of the program. The advantage of our method is that reachability analysis can typically be expected to be simpler to perform than computation of transitive closures or ranking functions. We expect that it should be possible to use and develop powerful techniques for backwards reachability analysis for many classes of parameterized and infinite-state programs. The technique is in general incomplete, but its power can be increased by performing repeated applications and by applying complementary techniques afterwards. The examples in the paper indicate that the method should be applicable to several classes of infinite-state systems.

Part IV: Conclusions

In this thesis we have focused on developing general methods allowing automatic verification for a particular family of infinite-state systems, namely the one of parameterized systems. We have approached this goal from three different angles.

In Part I, we have extended the framework of regular model checking to the case of parameterized systems with tree-like structures. Here, we have presented a technique for computing the transitive closure of a tree transducer. The technique is based on the definition and the computation of a good equivalence relation which is used to collapse the states of the transitive closure of the tree transducer. Our technique has been implemented and successfully tested on a number of protocols.

In Part II, we have presented a simple, light-weight, and efficient method permitting automatic analysis of safety properties for parameterized systems. We have considered three different classes of linear parameterized systems. These classes differ by the type of manipulated variables: finite, numerical, or arrays of finite or numerical variables. A key notion in our approach is to enforce monotonicity by over-approximating the original transition system. We perform a backwards reachability analysis that starts from a set of bad configurations representing the violation of the safety property to be checked. The analysis is continued until a fixpoint is reached. The analysis on the approximate transition system is guaranteed to terminate in the case where only finite variables are allowed. Based on the method, we have automatically verified several non-trivial parameterized systems such as Lamport Bakery algorithm [Lam74], Ricart and Agrawala distributed algorithm [RA81], and German cache coherence protocol [PRZ01].

In Part III, we have presented a method for proving termination properties of fair concurrent programs using backwards reachability analysis. The method relies on showing certain commutativity properties between different actions

of the program. The advantage of our method is that reachability analysis is typically simpler to perform than computating transitive closures or ranking functions.

Bibliography

[ABH+97] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification*, pages 340–351, 1997.

[ABJN99] Parosh Aziz Abdulla, Ahmed Bouajjani, Bengt Jonsson, and Marcus Nilsson. Handling global conditions in parameterized system verification. In *Computer Aided Verification*, pages 134–145, 1999.

[ACD90] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for real-time systems. In *LICS*, pages 414–425. IEEE Computer Society, 1990.

[ACJT96] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *Logic in Computer Science*, pages 313–321, 1996.

[ACJT00] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Inf. Comput.*, 160(1-2):109–127, 2000.

[AJ96a] Parosh Aziz Abdulla and Bengt Jonsson. Undecidable verification problems for programs with unreliable channels. *Inf. Comput.*, 130(1):71–90, 1996.

[AJ96b] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Inf. Comput.*, 127(2):91–101, 1996.

[AJMd02] Parosh Aziz Abdulla, Bengt Jonsson, Pritha Mahata, and Julien d'Orso. Regular tree model checking. In Brinksma and Larsen [BL02], pages 555–568.

[AJN+04] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, Julien d'Orso, and Mayank Saksena. Regular model checking for ltl(mso). In Alur and Peled [AP04], pages 348–360.

[AJNd03] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. Algorithmic improvements in regular model checking. In Jr. and Somenzi [JS03], pages 236–248.

[AJNS04] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Mayank Saksena. A survey of regular model checking. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 35–48. Springer, 2004.

[AK86] K R Apt and D C Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

[ALdR05] P. A. Abdulla, A. Legay, J. d'Orso, and A. Rezine. Tree regular model checking: A simulation-based approach. Technical Report 42, Centre Federe en Verification, 2005. available at http://www.montefiore.ulg.ac.be/~legay/papers/paper-tree.ps.

[And99] Greg R Andrews. *Foundations of Parallel and Distributed Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[AP04] Rajeev Alur and Doron Peled, editors. *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *"Lecture Notes in Computer Science"*. Springer, 2004.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[BG99] Bernard Boigelot and Patrice Godefroid. Symbolic verification of communication protocols with infinite state spaces using qdds. *Formal Methods in System Design*, 14(3):237–255, 1999.

[BHRV06] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomás Vojnar. Abstract regular tree model checking. *Electr. Notes Theor. Comput. Sci.*, 149(1):37–48, 2006.

[BHV04] Ahmed Bouajjani, Peter Habermehl, and Tomás Vojnar. Abstract regular model checking. In Alur and Peled [AP04], pages 372–386.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In Emerson and Sistla, editors, *cav00*, volume 1855 of *LECTURE NOTES IN COMPUTER SCIENCE*, pages 403–418. Springer-Verlag, 2000.

[BL02] Ed Brinksma and Kim Guldstrand Larsen, editors. *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *"Lecture Notes in Computer Science"*. Springer, 2002.

[BLP06] Sébastien Bardin, Jérôme Leroux, and Gérald Point. Fast extended release. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 63–66. Springer, 2006.

[BLS01] Kai Baukus, Yassine Lakhnech, and Karsten Stahl. Verification of parameterized protocols. *J. UCS*, 7(2):141–158, 2001.

[BLS02] Kai Baukus, Yassine Lakhnech, and Karsten Stahl. Parameterized verification of a cache coherence protocol: Safety and liveness. In Agostino Cortesi, editor, *VMCAI*, volume 2294 of *"Lecture Notes in Computer Science"*, pages 317–330. Springer, 2002.

[BLW03] Bernard Boigelot, Axel Legay, and Pierre Wolper. Iterating transducers in the large (extended abstract). In Jr. and Somenzi [JS03], pages 223–235.

[BLW04] Bernard Boigelot, Axel Legay, and Pierre Wolper. Omega-regular model checking. In Jensen and Podelski [JP04], pages 561–575.

[BMS05a] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *"Lecture Notes in Computer Science"*, pages 491–504. Springer, 2005.

[BMS05b] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination analysis of integer linear loops. In Martín Abadi and Luca de Alfaro, editors, *CONCUR*, volume 3653 of *"Lecture Notes in Computer Science"*, pages 488–502. Springer, 2005.

[BMS05c] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination of polynomial programs. In Radhia Cousot, editor, *VMCAI*, volume 3385 of *"Lecture Notes in Computer Science"*, pages 113–129. Springer, 2005.

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

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[BT02] Ahmed Bouajjani and Tayssir Touili. Extrapolating tree transformations. In Brinksma and Larsen [BL02], pages 539–554.

[CC77] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. In *Language Design for Reliable Software*, pages 77–94, 1977.

[CDG+98] H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 1998.

[CDG+99] H. Common, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. not yet published, October 1999.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[CGP+07] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. Proving that programs eventually do something good. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 265–276. ACM, 2007.

[CH88] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In Michael I. Schwartzbach and Thomas Ball, editors, *PLDI*, pages 415–426. ACM, 2006.

[CPR07] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving thread termination. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 320–330. ACM, 2007.

[CS01] Michael Colón and Henny Sipma. Synthesis of linear ranking functions. In Tiziana Margaria and Wang Yi, editors, *TACAS*, volume 2031 of *"Lecture Notes in Computer Science"*, pages 67–81. Springer, 2001.

[CS02] Michael Colón and Henny Sipma. Practical methods for proving program termination. In Brinksma and Larsen [BL02], pages 442–454.

[CTV06] Edmund M. Clarke, Muralidhar Talupur, and Helmut Veith. Environment abstraction for parameterized verification. In E. Allen Emerson and Kedar S. Namjoshi, editors, *VMCAI*, volume 3855 of *"Lecture Notes in Computer Science"*, pages 126–141. Springer, 2006.

[Del00a] Giorgio Delzanno. Automatic verification of parameterized cache coherence protocols. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *"Lecture Notes in Computer Science"*, pages 53–68. Springer, 2000.

[Del00b] Giorgio Delzanno. Verification of consistency protocols via infinite-stae symbolic model checking. In Tommaso Bolognesi and Diego Latella, editors, *FORTE*, volume 183 of *IFIP Conference Proceedings*, pages 171–186. Kluwer, 2000.

[DLS02] Dennis Dams, Yassine Lakhnech, and Martin Steffen. Iterating transducers. *J. Log. Algebr. Program.*, 52-53:109–127, 2002.

[EFM99] Javier Esparza, Alain Finkel, and Richard Mayr. On the verification of broadcast protocols. In *LICS '99: Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, page 352, Washington, DC, USA, 1999. IEEE Computer Society.

[EK03a] E. Allen Emerson and Vineet Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In Daniel Geist and Enrico Tronci, editors, *CHARME*, volume 2860 of *"Lecture Notes in Computer Science"*, pages 247–262. Springer, 2003.

[EK03b] E. Allen Emerson and Vineet Kahlon. Model checking guarded protocols. In *LICS*, pages 361–370. IEEE Computer Society, 2003.

[EN95] E. Allen Emerson and Kedar S. Namjoshi. Reasoning about rings. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 85–94, New York, NY, USA, 1995. ACM Press.

[EN96] E. Allen Emerson and Kedar S. Namjoshi. Automatic verification of parameterized synchronous systems (extended abstract). In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 87–98, London, UK, 1996. Springer-Verlag.

[EN98] E. Allen Emerson and Kedar S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *Logic in Computer Science*, pages 70–80, 1998.

[FP01] Dana Fisman and Amir Pnueli. Beyond regular model checking. In *Proc. 21th Conference on the Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, December 2001.

[FPPZ04] Yi Fang, Nir Piterman, Amir Pnueli, and Lenore D. Zuck. Liveness with incomprehensible ranking. In Jensen and Podelski [JP04], pages 482–496.

[FPPZ06] Yi Fang, Nir Piterman, Amir Pnueli, and Lenore D. Zuck. Liveness with invisible ranking. *STTT*, 8(3):261–279, 2006.

[FS98] Alain Finkel and Philippe Schnoebelen. Fundamental structures in well-structured infinite transition systems. *Lecture Notes in Computer Science*, 1380:102–??, 1998.

[GL93] Susanne Graf and Claire Loiseaux. A tool for symbolic program verification and abstration. In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 71–84, London, UK, 1993. Springer-Verlag.

[God96] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer-Verlag Inc., New York, NY, USA, 1996.

[GP93] Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 438–449, London, UK, 1993. Springer-Verlag.

[GS92] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.

[GW93] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, 1993.

[GZ98] E. Pascal Gribomont and Guy Zenner. Automated verification of szymanski's algorithm. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 424–438, London, UK, 1998. Springer-Verlag.

[Han93] Jim Handy. *The cache memory book.* Academic Press Professional, Inc., San Diego, CA, USA, 1993.

[HHK95] Monika Rauch Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, pages 453–462, 1995.

[Hig52] G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc. (3)*, 2(7):326–336, 1952.

[HKPM04] G. Huet, G. Kahn, and Ch. Paulin-Mohring. *The* Coq *Proof Assistant - A tutorial - Version 8.0*, April 2004.

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Inf.*, 29(6-7):523–543, 1992.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[HZ05] Nicolas Halbwachs and Lenore D. Zuck, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *"Lecture Notes in Computer Science"*. Springer, 2005.

[JN00] Bengt Jonsson and Marcus Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 220–234, 2000.

[JP04] Kurt Jensen and Andreas Podelski, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *"Lecture Notes in Computer Science"*. Springer, 2004.

[JS03] Warren A. Hunt Jr. and Fabio Somenzi, editors. *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *"Lecture Notes in Computer Science"*. Springer, 2003.

[KM89] R. P. Kurshan and K. McMillan. A structural induction theorem for processes. In *PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 239–247, New York, NY, USA, 1989. ACM Press.

[KMM+01] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, 256(1–2):93–112, 2001.

[KP99] Yonit Kesten and Amir Pnueli. Verifying liveness by augmented abstraction. In *CSL*, pages 141–156, 1999.

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[LB04] S. Lahiri and R. Bryant. Indexed predicate discovery for unbounded system verification, 2004.

[LHR97] David Lesens, Nicolas Halbwachs, and Pascal Raymond. Automatic verification of parameterized linear networks of processes. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 346–357, New York, NY, USA, 1997. ACM Press.

[LPS93] N. Lynch and B. Patt-Shamir. Distributed algorithms , lecture notes for 6.852 fall 1992. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1993.

[Mai01] Monika Maidl. A unifying model checking approach for safety properties of parameterized systems. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV*, volume 2102 of *"Lecture Notes in Computer Science"*, pages 311–323. Springer, 2001.

[McM92] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[MP84] Zohar Manna and Amir Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Sci. Comput. Program.*, 4(3):257–289, 1984.

[MP90] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties. In *PODC*, pages 377–410, 1990.

[MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[Nil02] Marcus Nilsson. Regular model checking, 2002. Website.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *"LECTURE NOTES IN COMPUTER SCIENCE"*. Springer, 2002.

[ORS92] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *"Lecture Notes in Artificial Intelligence"*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[Pel94] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, London, UK, 1994. Springer-Verlag.

[PPR05] Amir Pnueli, Andreas Podelski, and Andrey Rybalchenko. Separating fairness and well-foundedness for the analysis of fair discrete systems. In Halbwachs and Zuck [HZ05], pages 124–139.

[PR04] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *LICS*, pages 32–41. IEEE Computer Society, 2004.

[PR05] Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 132–144. ACM, 2005.

[PR07] Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.

[PRZ01] Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 82–97, London, UK, 2001. Springer-Verlag.

[PS00] Amir Pnueli and Elad Shahar. Liveness and acceleration in parameterized verification. In *Computer Aided Verification*, pages 328–343, 2000.

[PXZ02] Amir Pnueli, Jessie Xu, and Lenore D. Zuck. Liveness with (0, 1, infty)-counter abstraction. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 107–122, London, UK, 2002. Springer-Verlag.

[PZ03] Amir Pnueli and Lenore D. Zuck. Model-checking and abstraction to the aid of parameterized systems. In *VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, page 4, London, UK, 2003. Springer-Verlag.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[RA81] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, 1981.

[Rev93] Peter Z. Revesz. A closed-form evaluation for datalog queries with integer (gap)-order constraints. *Theor. Comput. Sci.*, 116(1&2):117–149, 1993.

[Rez08] Ahmed Rezine. Unbounded distributed parameterized systems, 2008. Website.

[RR04] Abhik Roychoudhury and I. V. Ramakrishnan. Inductively verifying invariant properties of parameterized systems. *Automated Software Engg.*, 11(2):101–139, 2004.

[SPBA00] Ekaterina Sedletsky, Amir Pnueli, and Mordechai Ben-Ari. Formal verification of the ricart-agrawala algorithm. In Sanjiv Kapoor and Sanjiva Prasad, editors, *FSTTCS*, volume 1974 of *"Lecture Notes in Computer Science"*, pages 325–335. Springer, 2000.

[Szy90] B. Szymanski. Mutual exclusion revisited. In *Proc. Fifth Jerusalem Conference on Information Technology*, pages 110–117. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 133–192. 1990.

[Val89] Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Applications and Theory of Petri Nets*, volume 483 of *"Lecture Notes in Computer Science"*, pages 491–515. Springer, 1989.

[Val92] Antti Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297–322, 1992.

[Var91] Moshe Y. Vardi. Verification of concurrent programs: The automata-theoretic framework. *Annals of Pure and Applied Logic*, 51(1-2):79–98, 1991.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344. IEEE Computer Society, 1986.

[WL90] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 68–80, New York, NY, USA, 1990. Springer-Verlag New York, Inc.