Modelling and Analysing Adaptive Networks: From Graph Transformation System to Ordinary Differential Equation

Mudhafar Saber Hussein
Department of Informatics
(previously Computer Science)
University of Leicester

A thesis submitted for the degree of
Doctor of Philosophy
at the University of Leicester
2017
I dedicate this thesis to my loving parents (Saber and Shams) and my beloved brothers and sisters.
Abstract

Online social networks pose particular challenges to designing effective algorithms and protocols. Apart from their distributed nature, their behaviour depends on user behaviour and is difficult to test at a realistic scale. Stochastic graph transformation systems can model the operation of such networks but due to the inherent complexity they are hard to analyse. Techniques such as model checking and simulation, which can be used to verify a range of quantitative properties, do not scale well to systems with large graphs and state spaces.

Aiming for an efficient alternative, we propose to derive a system of differential equations approximating the average evolution of the network. Variables in these equations represent numbers of occurrences of patterns selected to observe structural features. To keep the number of patterns small, rather than aiming for a fully accurate model we approximate complex patterns by the composition of simpler ones. We describe the approximation and its implementation based on critical pair analysis, illustrate and validate the process by examples of a social network protocol for P2P content policing and a voter model. We will also point out limitations in our approach to approximation and the use of differential equations more generally and discuss how they can be overcome.
Acknowledgements

Many Thanks to my supervisor Professor Reiko Heckel, who without his guidance, this thesis would not have been feasible. I sincerely appreciate the time and effort that he put in to ensure this work is standard. I also give gratitude to my family, friends and teaching staff for their encouragement and support while writing this thesis.
## Contents

1 Introduction 11
  1.1 Thesis Statement 13
  1.2 Approach 13
  1.3 Assumptions 16
  1.4 Contributions 17
  1.5 Overview of the Thesis 18

2 Background 20
  2.1 Adaptive Networks 20
  2.2 Graph Transformation 21
    2.2.1 Type and Instance Graph 21
    2.2.2 Graph Transformation Rules 22
    2.2.3 Negative Application Conditions 23
    2.2.4 Attributed Type Graphs 24
    2.2.5 Graph Constraints 26
  2.3 Graph Transformation Systems 27
  2.4 Graph Transition System 27
  2.5 Stochastic Graph Transformation Systems 28
    2.5.1 Continuous-time Markov chain (CTMC) 30
    2.5.2 Stochastic Simulation 32
  2.6 Practical Use of GTSs 34
  2.7 Steady State Analysis 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.8</td>
<td>Ordinary Differential Equations (ODEs)</td>
<td>36</td>
</tr>
<tr>
<td>2.9</td>
<td>Pair Approximation</td>
<td>37</td>
</tr>
<tr>
<td>2.10</td>
<td>Critical Pair Analysis</td>
<td>37</td>
</tr>
<tr>
<td>3</td>
<td>Social Voter Model: A Case Study</td>
<td>39</td>
</tr>
<tr>
<td>3.1</td>
<td>Type Graph</td>
<td>40</td>
</tr>
<tr>
<td>3.2</td>
<td>Graph Constraints</td>
<td>40</td>
</tr>
<tr>
<td>3.3</td>
<td>Start Graph</td>
<td>42</td>
</tr>
<tr>
<td>3.4</td>
<td>Patterns</td>
<td>43</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Pattern [0]</td>
<td>44</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Pattern [1]</td>
<td>44</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Pattern [00]</td>
<td>44</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Pattern [11]</td>
<td>45</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Pattern [01]</td>
<td>45</td>
</tr>
<tr>
<td>3.5</td>
<td>Graph Transformation Rules</td>
<td>45</td>
</tr>
<tr>
<td>4</td>
<td>Stochastic Model Checking</td>
<td>48</td>
</tr>
<tr>
<td>4.1</td>
<td>GROOVE Model Notation</td>
<td>49</td>
</tr>
<tr>
<td>4.2</td>
<td>Generation of the LTS</td>
<td>54</td>
</tr>
<tr>
<td>4.3</td>
<td>Transforming the LTS into a CTMC</td>
<td>55</td>
</tr>
<tr>
<td>4.4</td>
<td>Stochastic Analysis: PEPA</td>
<td>57</td>
</tr>
<tr>
<td>5</td>
<td>Stochastic Simulation</td>
<td>59</td>
</tr>
<tr>
<td>5.1</td>
<td>Modelling Notation</td>
<td>60</td>
</tr>
<tr>
<td>5.2</td>
<td>Metamodel in VPM</td>
<td>63</td>
</tr>
<tr>
<td>5.3</td>
<td>Model in VPM</td>
<td>65</td>
</tr>
<tr>
<td>5.4</td>
<td>Model GT Rules in VIATRA2 Syntax</td>
<td>66</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Patterns</td>
<td>66</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Rules</td>
<td>68</td>
</tr>
<tr>
<td>5.5</td>
<td>Model Probes in VIATRA2 Syntax</td>
<td>71</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Probe [01]</td>
<td>71</td>
</tr>
</tbody>
</table>
5.5.2 Probe [00] ........................................... 72
5.5.3 Probe [11] ........................................... 72
5.5.4 Probe [0] ........................................... 72
5.5.5 Probe [1] ........................................... 72

5.6 Simulation Parameters ........................................ 73
5.6.1 Stochastic Structure ........................................ 73
5.6.2 Execution Parameters ........................................ 75

5.7 Simulation Experiments ........................................ 77
5.7.1 Experiment 1 ........................................... 79
5.7.2 Experiment 2 ........................................... 79

5.8 Results ..................................................... 85
5.8.1 Results of Experiment 1 ...................................... 85
5.8.2 Results of Experiment 2 ...................................... 85

6 From Stochastic GTS to ODEs ................................ 87
6.1 ODEs for Patterns ........................................... 88
6.2 Pattern Approximation ........................................ 88
6.2.1 Approximation of Basic Patterns .............................. 88
6.2.2 Approximating the Effect of Basic Rules ................. 90
6.2.3 Approximating Conditional Patterns and Rules ........... 91

7 Algorithm and Implementation ................................ 93
7.1 Algorithm .................................................. 93
7.2 Implementation ............................................. 100
7.2.1 Architecture ............................................. 100
7.2.2 User Interface ............................................. 109
7.2.3 Validation ................................................ 109

8 ODEs from the Voter Model .................................. 110
8.1 Deriving the ODEs .......................................... 110
8.1.1 Pattern [0] ........................................... 112
List of Tables

4.1 Edit Prefixes .................................................. 50
4.2 Number of Pattern occurrences in certain States ............... 58
4.3 The Probability of each State .................................. 58
4.4 Weighted Average Frequency of Patterns ....................... 58

5.1 The Weighted average Frequencies of Patterns in Steady state . 85
5.2 The Weighted average Frequency of Patterns in Steady state . 86

8.1 Pattern occurrences and their approximation for initial graph . 122
8.2 Pattern occurrences and their approximation for final graphs . 122
8.3 The Steady State in the System .................................. 125

9.1 Initial state ................................................... 134
9.2 Final state ................................................... 135
9.3 Pattern population when they stabilise .......................... 138

10.1 The WFPS in a small model .................................... 146
10.2 Time spent in each method ..................................... 146
10.3 The Weighted Average Frequencies of Patterns in a larger model 148
10.4 Time spent in each method ..................................... 150
10.5 The Weighted Average Frequencies of Patterns in both approaches 151
List of Figures

1.1 The schema of all three analysis methods ................................. 15
2.1 The interplay between network state and topology [24] ........... 20
2.2 Type and Instance Graphs .................................................. 22
2.3 Span of a Graph Transformation Rule in the DPO approach .... 23
2.4 A Graph Transformation Rule ............................................. 23
2.5 Negative Application Condition (NAC) ............................... 24
2.6 NAC Structure ............................................................... 24
2.7 An attributed rule application ............................................. 25
2.8 Attributed Type Graph ....................................................... 26
2.9 Attributed Instance Graph .................................................. 26
2.10 Graphical Constraints on Patterns ...................................... 27
2.11 A Graph Transition System ............................................... 28
2.12 A Q-matrix ................................................................. 31
2.13 P2P content policing GT Rules. ........................................ 35
3.1 Type Graph for Voter Model ............................................... 40
3.2 Graphical Constraints - Node have two attribute values ....... 41
3.3 Graphical Constraints - Parallel Edges ................................. 41
3.4 Graphical Constraints - Loops ............................................. 41
3.5 Graphical Constraints - Symmetric Edges ............................. 42
3.6 Graphical Constraints - Agent holds a Vote ......................... 42
3.7 Start Graph used in GROOVE and VIATRA2 ....................... 43
3.8 Pattern [0] ......................................................... 44
3.9 Pattern [1] ......................................................... 44
3.10 Pattern [00] ....................................................... 44
3.12 Pattern [01] ....................................................... 45
3.13 Rule convertTo0 ................................................. 46
3.14 Rule convertTo1 ................................................. 46
3.15 Rule splitTo0 .................................................... 46
3.16 Rule splitTo1 .................................................... 47
4.1 Type Graph ....................................................... 49
4.2 Start Graph ....................................................... 50
4.3 Rule convertTo0 .................................................. 52
4.4 Rule convertTo1 .................................................. 52
4.5 Rule splitTo0 ..................................................... 53
4.6 Rule splitTo1 ..................................................... 53
4.7 Reverse of conversion rules ................................. 54
4.8 Reverse of split rules ......................................... 54
4.9 Pattern [00] ....................................................... 55
4.10 The generated LTS from the Voter Model ............... 55
4.11 CTMC Generator also shows the input files ............ 56
5.1 VIATRA2 Metamodel Notation ............................. 60
5.2 VIATRA2 Model Notation ..................................... 61
5.3 Voter Metamodel ............................................... 64
5.4 Viatra Visualisation of Metamodel ......................... 65
5.5 StoSimPars Entity of Model Space ......................... 76
5.6 Start Graph ..................................................... 80
5.7 Metamodel including the Counter node .................. 81
5.8 Rule 1 .......................................................... 82
5.9 Creating edges between the nodes ........................ 82
5.10 Delete the NoLink edges .................................. 82
5.11 Defining the Parameters .................................. 83
5.12 Viatra Visualisation of Model .............................. 84
5.13 The evolution of the system over time .................... 86
5.14 Result of 20 runs ......................................... 86

6.1 Overlapping $p_1, p_2$ at $p_0$ ................................. 89
6.2 The critical gluing between rule convertTo1 and pattern $[0]$ . 91
6.3 The gluing critical between rule convertTo0 and pattern $[0]$ . 92

7.1 Overall tool architecture ................................ 100
7.2 Screenshot of AGG graph transformation tool ............ 101
7.3 Class diagram - API in Java .............................. 104
7.4 Pattern independence validator - VIATRA2/GraTS ........ 107
7.5 ODE solver - Octave .................................... 108
7.6 The graphical user interface of the ODE generator ....... 109

8.1 Import the GTS/AGG file into the ODE generator ....... 111
8.2 Generate the equations .................................. 112
8.3 The effect of rule convertTo0 on pattern$[0]/X_4$ .......... 113
8.4 The effect of rule convertTo1 on pattern$[0]/X_4$ .......... 114
8.5 The effect of rule convertTo0 on pattern 01 ............... 116
8.6 The effect of rule convertTo1 on pattern 01 ............... 118
8.7 The effect of rule splitTo0 on pattern 01 ................. 118
8.8 The effect of rule splitTo1 on pattern 01 ................. 119
8.9 The Result of the ODEs .................................. 125

9.1 A P2P content policing Type Graph ....................... 128
9.2 Constraints - Loops ...................................... 128
9.3 Constraints - like and blame at the same time ............. 128
9.4 P2P content policing Patterns ........................... 129
9.5 P2P content policing GT Rules ........................... 130
9.6 P2P content policing model in AGG. ................. 131
9.7 The effect of rule blame_bad_content on Pattern TP ........ 132
9.8 Mapping of $X$ variables ........................................ 134
9.9 Pattern behaviours over time in Simulation .................. 137
9.10 Pattern behaviours over time in ODE ......................... 138
9.11 Pattern evolution: original model .......................... 139
9.12 Second version of P2P content policing GT Rules. .......... 142
9.13 Pattern evolution: with restricted blaming ................ 143

10.1 The Weighted Average Frequency Plot ..................... 147
10.2 The evolution of the system over time in simulation method . 149
10.3 The evolution of the system over time in ODE method .......... 149
10.4 The Weighted Average Frequency Plot ..................... 149
10.5 A larger graph to check the scalability problem in model checking 152
10.6 Simulation run resulting in extinction of minority .......... 153
Chapter 1

Introduction

Online social networks are distributed applications with large numbers of users whose behaviour is hard to predict, but crucial for decisions about the software platform and policies. Understanding how global effects can arise from individual actions is an aim of social network analysis, where individuals and their relationships are represented as nodes and edges in a graph. In order to capture the dynamic nature of such networks, the notion of adaptive networks [24] allows changes to the network topology as well as the data state of individual nodes.

Such networks have been modelled by stochastic graph transformation systems [30] (SGTS) whose rules change to data and topology while their rates represent timing. Other types of adaptive networks modelled by stochastic graph transformations include peer-to-peer VoIP applications and distributed business processes, but other networked applications share the same features: Their properties depend on the collective behaviour of users, which is hard to predict but important for design and policy decisions.

Stochastic graph transformation systems can be analysed through stochastic model checking [26] or simulation [47] to study their evolution over time, discover long-term trends or predict the likelihood or frequency of certain events. However, scalability to large networks is a major challenge, especially because
real-life social networks worth studying can have many millions of nodes.

In computational biology, the stochastic graph rewriting approach Kappa allows to derive ordinary differential equations (ODEs) as effective tool for analysing biochemical reaction networks. Where applicable, fragmentation [20] allows analyses to scale independently of the size of the population (e.g., the number of nodes in the network) or the size of the state space.

The idea of fragmentation is to define a set of structural patterns \( p \in P \) to be observed so that variables \([p]\) represent the number of occurrences of these patterns in a graph \( G \). Differential equations over these variables describes the evolution of the mean of \([p]\) over time, in all graphs encountered at time \( t \) in any of the possible runs of the network, weighted by the probability of being in \( G \) at \( t \). Under suitable assumptions, the latter probability converges towards the probability of \( G \) in the steady state solution of the Continuous Time Markov Chain (CTMC) derived from the SGTS. Then, \([p]\) represents the probability-weighted average frequency of occurrences of \( p \) in the states of the CTMC. While fragmentation works well for the restricted kinds of graphs representing biochemical compounds, for general graphs an accurate analysis of SGTS may require an infinite set of ODEs [13] based on patterns of increasing size generated in the process.

The main idea of our approach is to avoid generating large patterns, approximating them by smaller ones. This approximation works under the condition of pattern independence assumption, i.e., it is accurate if pattern occurrences are independent, as can be expected in large random graphs. In such a case we can express the number of occurrences of a pattern \( p = p_1 \cup p_2 \) with \( p_1 \cap p_2 = p_0 \) by those of \( p_1 \) and \( p_2 \). Independence here means that finding the occurrence of \( p_1 \) does not increase or decrease the probability of finding an overlapping occurrence of \( p_2 \). Based on this principle of approximation we formalise and justify the translation of an SGTS into a system of ODEs, show how to validate the assumption of pattern independence by simulation and provide case studies to assess the approach and its limitations.
1.1 Thesis Statement

Analysis of graph transformation systems via stochastic model checking and simulation can be challenging and resource intensive, particularly for large and complex models. Deriving a system of ODEs from a stochastic graph transformation system increases the scalability of this analysis. The ODE system approximates the average evolution of the network. The approximation and its implementation have been described, and the process is illustrated and validated by examples of social networks. The results of existing approaches, model checking and simulation, are compared to the proposed one in order to address these research questions:

- What are the differences, limitations and trade-offs between different analysis approaches?
- How can they work together and support each other in a coherent methodology?

1.2 Approach

We propose and validate an approach to approximate the dynamic behaviour of a stochastic graph transformation system by a set of differential equations. This approach is scalable and appears to capture most of the long-term behaviour of the origin model. In order to state the dynamics of an adaptive network in terms of ODEs, for each equation describing the evolution of a pattern we need to know the number of occurrences of the rules left-hand sides and the rules’ effects on the pattern’s occurrences, such as how many of them are created and destroyed by one rule application, and the rate for the rule. To calculate these, pair approximation techniques are used to approximate the complex patterns by combinations of simpler ones. The generation of ODEs from SGTS is fully automated using AGG’s critical pair and minimal dependency analysis.
We explain three different methods, namely model checking, simulation and ordinary differential equations in the thesis. The three different approaches can work together to support each other. To address the state space explosion problem, which prevents us from the derivation of a CTMC for larger systems, stochastic simulation can be an alternative method. However, the simulation eventually faces the same problem when the network grows more complex, which can be very costly. That is where the ODEs as an abstract mathematical description become a good alternative method which provides a better trade-off for analysis efforts.

Running different methods on the same version of the model allows to cross-validate results. To achieve that, an example of a dynamic social network is presented that explore the interdependency between voting behaviour and relationships of individuals for model checking graph transformation. The analysis results are then compared with the simulation results of the same original system. Subsequently, we scale up the system to stochastic simulation and the results of simulating are compared with those obtained from solving the system of equations. The simulation experiments link the model checking and differential equation methods. A more complex example Peer-to-Peer Content Policing social network is presented to test the applicability of our approach to other systems. Fig. 1.1, shows the schema of all three methods.

- Stochastic model checking: Following the translation of the type graph, start graph and GT rules into GROOVE (GRaphbased Object-Oriented VERification) [39] syntax in Section 4.1, we will discuss generating Labelled Transition Systems (LTS)s from graph transformation systems using GROOVE in Section 4.2. Then transform them into CTMCs in Section 4.3 in order to assign every transition in LTS a stochastic rate and perform stochastic analysis using Performance Evaluation Process Algebra (PEPA) [21]. This is explained in Section 4.4, concluding by presenting
and analysing the results.

• Stochastic simulation: We present the Voter model, Sections 5.2, 5.3, 5.4 and 5.5, which is translated into VIATRA2 [49] syntax in Section 5.1. Then the simulation parameters are defined in Section 5.6 in order to carry out the simulation experiments in Section 5.7. We conclude the method in Section 5.8 with presenting and analysing the results.

• Ordinary Differential Equations: We present an approach to derive ODEs from stochastic graph transformation systems including an algorithm and its implementation. The approach is presented in Chapter 6 and the algorithm and its implementation in Chapter 7. The case studies and the analysis are all detailed in Chapters 8 and 9. The overall requirements of the approach, as it is shown in the above Fig. 1.1, are:

  - AGG [1] tool: to model the graph transformation system.
  - ODE Generator, where the equations are automatically produced.
— VIATRA2 tool: for the simulation to validate pattern independence.

— Octave tool [23]. to solve the ODEs.

1.3 Assumptions

Our approach works under the following assumptions:

1. Pattern independence: The approximation in our approach is accurate when pattern occurrences are independent from each other. This was discussed in the Introduction of Chapter 1 and further details will be in Chapter 6.

2. AGG: AGG imposes restrictions on attributed graphs and attributed rule applications. Thus, deriving the attribute algebra to compute the critical pairs works up to the limitations imposed by the AGG critical pair analysis. The restrictions are

   • In the left-hand side of a rule, only constants or variables can be used.

   • In the right-hand side of a rule, constants, variables, or expressions can be used.

The detail of AGG and its restrictions will be discussed in Chapter 7.

3. Exponential distribution: The evolution of an adaptive network is a random process and we use a Stochastic Graph Transformation System (SGTS) to model and analysis such networks. Dominating the application of graph transformation rules by exponential distribution is making the SGTS stochastic. Therefore, such assumption is essential in our system. Refer to sections 2.5 and 2.5.2 for further details.
1.4 Contributions

The thesis consists of the following contributions:

1. Develop and validate a generic approach to approximating the dynamic behaviour of a stochastic graph transformation system by a set of differential equations.

2. The approach is automated to generate ODEs from SGTS using the critical pair analysis.

3. To calculate the rules’ left-hand sides and their effects on the patterns, the concept of pair approximations is introduced to replace complex patterns by combinations of simpler ones.

4. Validating the assumptions about the patterns independences using simulation.

5. We model a simple but representative example of adaptive networks, the Voter model, characterised by mutual dependencies between topology evolution and local state changes in network nodes.

6. Contribute in modelling Peer-to-Peer Content Policing which describes a policy of content policing in social networks, to consider a more complex system than the Voter to test our approach on.

7. We investigate and evaluate three analytical techniques in order to address the scalability issues of stochastic graph transformation analysis. The results of all three methods are compared, and it demonstrates that the results of model checking and simulation of the original model is similar to those obtained from solving the system of equations.

8. A small program is developed to extract the number of pattern occurrences in each state and the steady state probability of the state from the PEPA
model. The details and the technical terms will be explained in Chapter 4.

Parts of this thesis submission appeared in the following publication.


In the paper we have used the same case study, the Voter model, but with some adaptations. The rules, patterns and some constraints are similar. Also similar experiments are used as the ones in Chapter 4. Parts of this paper have contributed to Chapters 3 and 4.

### 1.5 Overview of the Thesis

The remainder of this thesis is organised as follows:

**Chapter 2**
Discusses the background of the relevant concepts and approaches we have used throughout the research.

**Chapter 3**
Presents a case study based on a social voter model.

**Chapter 4**
Presents the first approach, model checking analysis technique, to analyse non-functional properties of the Voter system.

**Chapter 5**
Discusses the translation and encoding the Voter model into VIATRA2 syntax in order to perform the stochastic simulation, second approach, using the Graph-based Stochastic Simulation (GraSS) tool. This chapter
also compares the result of the simulation analysis with the model checking
and produce the input for the comparison analysis will be done with the
ODE approach.

Chapter 6
Describes our approach to derive the differential equations from stochastic
graph transformation system. The concepts of pattern approximation is
also introduced.

Chapter 7
An algorithm to systematically generate ODEs is presented as well as its
implementation.

Chapter 8
Applies our third approach on the Voter model to derive ODEs and com-
pare the result analysis to the ones of the previous approaches.

Chapter 9
Present another case study which describes a policy of content policing
by social networks. The chapter includes the description of the case study
and apply the approaches stochastic simulation and ODE on it to compare
the results analysis and evaluate them.

Chapter 10
Evaluates the results of the three different analysis approaches based on
the Voter case study and discuss their limitations and trade-offs. This
chapter also discusses how the approaches work together and support
each other. It also discusses the related works.

Chapter 11
Gives the conclusion and future work.
Chapter 2

Background

In this chapter we discuss the background of the relevant concepts and approaches we have used throughout the research. We have given the basic notions of necessary concepts in the following sections.

2.1 Adaptive Networks

Adaptive networks are characterised by mutual dependencies between nodes’ local state changes and evolving topology. For example, a state change may depend on a node’s connection while in turn enabling the creation of new connections that could lead to changes in nodes, etc [32]. See Fig. 2.1.

![Figure 2.1: The interplay between network state and topology [24]](image)

This interdependency prevents a layered view, where state and topology
changes occur at different speeds and are handled at different levels, such as in traditional approaches to dynamic reconfiguration of component-based systems where the components communication and local computation is halted in order for reconfiguration to take place. Adaptive networks do not allow this two-level approach and are therefore intrinsically more complex [24][32]. Adaptive networks are hardly new, almost every real world networks are adaptive to some extent [25]. Examples range from biological systems and social networks, peer-to-peer or other distributed systems with autonomous components, to technical networks such as power grids, the Internet and mobile communication infrastructure [24][32].

2.2 Graph Transformation

Graphs are a very natural way of explaining complex situations on an intuitive level. Graph can be used to model the states of any kinds of system, which enables one to use graph transformation to model the state changes in these systems. This allows the user to apply graph transformation system to a wide range of fields in computer science. The main idea of using graph transformation is the rule-based modification of graphs, where each application of a graph rule leads to a graph transformation step [18]. Graphs we use throughout the thesis are directed.

2.2.1 Type and Instance Graph

A type graph $TG$ is a graph whose nodes represent nodes types and whose edges represent edges and consists of $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$ where $V_{TG}$ is a vertex type alphabet, $E_{TG}$ is an edge type alphabet and $s_{TG}, t_{TG}$ associate, respectively, a source and target node for each edge in $E_{TG}$. Type graphs are used to restrict the set of admissible graphs, it means restricting the shape of an object in order to prescribe a type for the object. Instance graph, graph $G$, is
a graph that is typed over the type graph $TG$. It is provided with a graph morphism $\text{type}_G : G \to TG$ in order to assign a type to every node and edge in $G$. A typed graph morphism is a morphism that preserves the typing of graphs [18].

$$G \xrightarrow{f} H$$

$$\text{type}_G \quad \text{type}_H$$

$$TG$$

Figure 2.2: Type and Instance Graphs

In a type graph a node type can be compared to a class and an edge type can be compared to an association in a UML class diagram. Also instance graphs can be compared to UML object diagrams.

### 2.2.2 Graph Transformation Rules

Graph transformation rules are used to define observable events that lead from one state to another (action rules), as well as quantitative observations over states (probe rules). We use graph transformation rules in our work which model the evolution of network populations. A graph transformation rule $p = (L \leftarrow K \rightarrow R)$ consists of three instance graphs $L$ the left-hand side (LHS), $K$ the gluing graph and $R$ the right-hand side and two injective graph morphisms $l$ and $r$. The left-hand side of the rule $L$ is a graph specifying the preconditions for the application of the rule. $K = L \cap R$ is the gluing graph, which specifies which nodes and edges in $L$ are unchangeable by the rule. The right-hand side of the rule $R$ represents the postconditions of the rule. $L, K$ and $R$ are all typed over type graph $TG$.

In order to generate a transformation on a given graph $G$, there must be an occurrence of the left-hand side $L$ in the instance graph $G$. This occurrence is also called match, i.e., finding an injective morphism $m$ between $L$ and $G$. 

22
Once a match is found, all vertices and edges matched by $L \setminus K$ are deleted from $G$. At the end, the new nodes and edges in $R \setminus K$ are copied into the result to yield the derived graph $H$. The application of a rule by the double pushout (DPO) approach ensures the construction in Fig. 2.3 and the dangling condition ensures that $D$ is graph [26] [34] [8] [9].

\[ \begin{array}{ccc}
L & \xrightarrow{l} & K & \xrightarrow{r} & R \\
\downarrow m & & \downarrow k & & \downarrow n \\
G & \xrightarrow{f} & D & \xrightarrow{g} & H
\end{array} \]

Figure 2.3: Span of a Graph Transformation Rule in the DPO approach

Fig. 2.4 is an example rule, showing the transformation from one state to another. In the LHS there are three voter nodes. Two of them have different opinions are connected to each other. The rule deletes the edge between them and reconnects the node to third one that has the same opinion.

2.2.3 Negative Application Conditions

Our approach includes Negative application condition (NAC) used in both case studies in Chapters 3 and 9. NACs are defined within graph transformation rules as a preconditions to prevent their application. They specify that certain nodes or edges must not be present in the selected match. For example, Fig. 2.5
illustrates a NAC for a rule and states that the rule will only apply if there is no connection between Agent1 and Agent3. A NAC is denoted with a diagonal line across an edge (or node) within the left-hand side of the rule. In the next presentations of the rules, we do not show $K = L \cap R$.

![Figure 2.5: Negative Application Condition (NAC)](image)

Fig. 2.6 shows the NAC structure to the double pushout DPO construction. A typed graph morphism $m : L \rightarrow G$ satisfies $NAC(x)$, where $x : L \rightarrow X$, if there is no injective typed graph morphism $p : X \rightarrow G$ where $p \circ x = m$ [18].

![Figure 2.6: NAC Structure](image)

### 2.2.4 Attributed Type Graphs

In the thesis, the attributes play an important role. Nodes and edges can be enriched with attributes to stores additional information. In object-oriented languages, an attribute has a name and data value. However, in the context of typed graphs, the attributes that belong to certain node type are declared by their name and data type in the graph. After declaration, each node in the instance graph can carry different values in the attribute, having the same
data type. In graph transformation, attributes can be useful to restrict the applicability of transformation rules. A rule will only be applied if a certain node meets the attribute constraints of the rule. Fig. 2.7 illustrates an attributed rule application where attribute \textit{true} changes to \textit{false} after the rule applies. We have also used attributes to model the local states of nodes in the network which plays a major role in the dynamics of the system. The attributes are generic and attributed algebra is allowed to be used up to the AGG tool limitations, see Section 1.3.

![Figure 2.7: An attributed rule application](image)

Attributed graphs are formally defined by so-called \textit{E-graph} which extend the formalism of graph to allow nodes and edges to store additional information through attributes. In \textit{E-graphs}, there are two distinct kinds of nodes: traditional graph nodes and data nodes. Data nodes hold values for attributes. There are also three kinds of edges as well, the usual graph nodes, node attribute edges and edge attribute edges.

An \textit{E-graph} $G = (V_G, V_D, E_G, E_{NA}, E_{EA}, (source_j, target_j)_{j \in G, NA, EA})$, where $V_G$ are graph nodes, $V_D$ are data nodes, $E_G$ are graph edges, $E_{NA}$ are node attribute edges, and $E_{EA}$ are edge attribute edges. For $j = G$ the source function for graph edges is $source_G : E_G \rightarrow V_G$, and the target function for graph edges is $target_G : E_G \rightarrow V_G$, whereas for $j = NA$ the source function for node attribute edges is $source_{NA} : E_{NA} \rightarrow V_G$ and the target function is...
targetNA : $E_{NA} \to V_D$. When $j = EA$ the source function for edge attribute edges is $source_{EA} : E_{EA} \to E_G$, and the target is $target_{EA} : E_{EA} \to V_D$. In the type graph, the attributes of each node and edge can be set out the same way as graph nodes and edges. The main difference between graphs and E-graphs is that we allow edge attribute edges, where the source of these edges is not a graph node but a graph edge [18].

Fig. 2.8 shows the type graph of a model and Fig. 2.9 shows a possible instance graph. The Agent1 and Agent2 nodes in the Fig. 2.9 are typed over type node Agent of Fig. 2.8; vote is the attribute of the Agent node with the data type Boolean and have true or false values in the instance nodes. These figures also present a mapping of E-graph elements.

2.2.5 Graph Constraints

Graph constraints are additional conditions on graphs that are not expressed by the type graph [16]. They have been created for the graph transformation systems in Chapters 3 and 9. A graphical constraint can be used to define permitted or forbidden subgraphs. They are used to construct the condition that a graph $G$ must (or must not) contain a certain subgraph $G'$, without being
connected to a GT rule as it is shown in the Fig. 2.10. PAR-C2C means that there are no parallel edges between the Agents. V-vote means no Agent has two vote attributes. In UML notation, attributes are forbidden to have more than one value, but according to the E-graph definition nodes are allowed to have more than one data node. In order to satisfy the constraints, a graph $G$ must not contain subgraphs isomorphic to them.

Figure 2.10: Graphical Constraints on Patterns

2.3 Graph Transformation Systems

Graph transformation systems are a natural model for networks with evolving topology, representing network nodes and edges by graph vertices, attributed to model local states, and graph edges, respectively [32].

An attributed graph transformation system $GTS = (TG, R, \pi, G_0)$ consists of a type graph $TG$, a set of rule names $R$ and a function $\pi$ mapping each rule name $r \in R$ to a $TG$-typed rule $r = (L \leftarrow K \rightarrow R, NAC)$, and a $TG$-typed start graph $G_0$.

2.4 Graph Transition System

A transition system describes the behaviour of a system. It consists of states and transitions between states [33]. States represent isomorphism classes of graphs and transitions represent rule applications as it is shown in the Fig. 2.11. Its state space explore all reachable graphs of the transformation system.
from the start graph. Such transition systems can be generated by applying recursively all enabled graph transformation rules of \( G \) at each state and by matching the resulting graphs with already generated isomorphic graphs [8].

Graph transition system is used in Chapter 4 to explore the labelled transition systems (LTS)s from a graph modelled in GROOVE (GRaphbased Object-Oriented VErification) [39].

Figure 2.11: A Graph Transition System

### 2.5 Stochastic Graph Transformation Systems

Stochastic Graph Transformation Systems SGTSs allow us to represent the probabilistic timing. They are used in Chapter 4 for model checking and Chapter 5 for stochastic simulation. A stochastic graph transformation system extends a GTS by associating each rule name to a positive real number representing the rate \( \lambda \) (average delay \( \lambda^{-1} \)) of the exponentially distributed delay of its application [26]. A stochastic graph transformation system \( SG = (GTS, \rho) \) consists of a graph transformation system \( GTS = (TG, R, \pi, G_0) \) and a function \( \rho : R \rightarrow \mathbb{R}^+ \) that maps every rule to its application rate \( \rho(r) \).

For the rules of a model, rates are given by, for example, \( \rho(convertTo0) = \rho(convertTo1) = 1000 \) and \( \rho(splitTo0) = \rho(splitTo1) = 1 \). That means rule \( convertTo0 \) is 1000 times as fast as rule \( splitTo0 \). In other words, if either rule
is applicable in a graph, the average delay of `convertTo0` is one thousandth that of `splitTo0`. It means in a situation where two different opinionated agents are connected, converting one’s peer or oneself is more likely than splitting up. Therefore the addition of delays for each rule via $\rho$ adds quantitative information about the model, which could lead to the extraction of quantitative model properties such as reliability, performance, and steady state statistics. The set of rules $R$ specifies the qualitative behaviour of a model, for example, having two GTS rules and a match for either rule could disable a match for the other. There is no information, in the qualitative model, about the probability of one being disabled by the other or vice versa. In the quantitative SGTS model, if the first rule has a higher application rule rate than the second one, then the first one has a bigger probability of being applied before second one given that they are enabled at the same time [9].

Since the evolution of adaptive network is a random and uncontrolled process, the exponential distribution is used for modelling the system. For systems that have a well-defined start and end to the processes, normal distribution or a lognormal distribution is suitable to use. For example, the time that a customer takes in a supermarket to complete the checkout process is captured by a normal distribution, since the time between the arrival at the queue and leaving has a definite start and end and is controlled by the speed of the staff. However, a customer joining a queue would have an exponential distribution because there is no control by the supermarket over the time of the customer’s arrival at each queue [9].

If $\rho$ assigns an exponential probability distribution to each rule in $R$, then the model corresponds to a Continuous-Time Markov Chain, which has the memoryless property. This means that given times $0 < t < x < x + t$, the probability that a rule match has awaited application from 0 to $t$ is exactly the same as the probability that the match will wait for another time period $t$ at $x$ given that the match has not been applied yet. Therefore, the rule match has no memory of or dependence on how long it has waited to be applied up to that
The exponential distribution is used to compute the time expected for each rule to elapse (scheduled delay) prior to the application of that particular rule. Dominating the application of transformation rules by continuous probability distributions is making the SGTS stochastic. This is because each pair of active rule-match is associated with an independent random variable (timer) which is randomly set according to a continuous probability function [31]. This is discussed more in the Section 2.5.2.

2.5.1 Continuous-time Markov chain (CTMC)

A continuous-time stochastic process \( \{X(t) : t \geq 0\} \) is called a continuous-time Markov chain if it has the Markov property. The Markov property is a \textit{forgetting} property, suggesting memorylessness in the distribution of the time a continuous-time Markov chain spends in any state [46]. A stochastic process has the Markov property if the conditional probability distribution of future states of the process depends only upon the present state, not on the past. The term \textit{Markov property} has been used in connection with a particular \textit{memoryless} property of the exponential distribution [19].

Analysis techniques for CTMCs allow us to perform stochastic analysis, such as steady-state analysis, on a system. In order to derive the CTMCs from a graph transformation system, we first have to generate its state space from the start graph and then translate the labelled transition systems (LTS) into CTMC, i.e., assign every transition in LTS a rate. Section 4.3 will detail this process.

Below are some basic notions for adopting the Q-matrix, ”incidence matrix” of the Markov Chain and explaining the connection with CTMCs [26, 8] .

Let \( S \) be a countable set. A \textit{Q-matrix} on \( S \) is a real-valued matrix \( Q = Q(s, s')_{s, s' \in S} \) satisfying the following conditions:

(i) \( Q(s, s) \leq 0 \) for all \( s \in S \), diagonal,
(ii) $Q(s, s') \geq 0$ for all $s \neq s'$,

(iii) $\sum_{s' \in S} Q(s, s') = 0$ for all $s \in S$.

The Q-matrix is also called transition rate matrix of the Markov chain, see Fig. 2.12 that shows an example of incidence matrix between some states. A CTMC makes transitions from state to state independent of the past for an exponentially distributed amount of time. It has the memoryless property.

A continuous-time Markov chain is a pair $(S, Q)$ where $S$ is a countable set of states and $Q$ is a Q-matrix on $S$.

$$Q(s) = \sum_{s \neq s'} Q(s, s') < \infty$$

If $s \neq s'$ and $Q(s, s') > 0$, then there is a transition from $s$ to $s'$. The transition delay is exponentially distributed with rate $Q(s, s')$. Consequently, the probability that, being in $s$, the transition $s \rightarrow s'$ can be triggered within a time interval of length $t$ is $1 - e^{-Q(s,s')t}$. The total exit rate $Q(s)$ specifies the rate of leaving a state $s$ to any other state and the diagonal entry $-Q(s)$ makes the total row sum zero. If the set \{s' $|$ $Q(s, s') > 0$\} is not a singleton, then there is a competition between the transitions originating in $s$. The probability that transition $s \rightarrow s'$ wins the 'race' is \[\frac{Q(s, s')}{Q(s)}\].
We say that a state \( s' \) can be reached from \( s \), and write \( s \rightarrow s' \), if there are states \( s = s_0, ..., s_n = s' \), such that \( Q(s_0, s_1) \cdot Q(s_1, s_2) \cdot ... \cdot Q(s_{n-1}, s_n) > 0 \). If \( s \rightarrow s' \) and \( s' \rightarrow s \), then we say that \( s \) and \( s' \) communicate, and write \( s \leftrightarrow s' \).

When a system is modelled as a GTS, we have to take into account that the resulting Markov chain should be irreducible. It means that every state in the LTS is reachable from every other one. A \( Q \)-matrix \( Q \) is irreducible if \( s \leftrightarrow s' \) for all \( s, s' \in S \). This is achieved by adding reverse rules for each of the rules. This ensures that the labelled transition system is strongly connected. The reverse rules are given very low rates so they do not affect the dynamic behaviour.

2.5.2 Stochastic Simulation

In SGTSs, properties of interest can be defined using continuous-time Markov chains to do stochastic analysis via model checking. This approach requires the entire state space in the form of a labelled transition system [26]. This approach is accurate when it is used on a small systems, but in complex systems the state space exploration faces scalability problems and becomes intractable. To tackle this, alternative approaches are required. Stochastic simulation is a good alternate when a system is scaled up. Although, it might not be as accurate as model checking, it could provide a good insight into non-functional properties of a system being modelled via the repeated execution of a system with the aid of a computer, such as simulated time per run, the frequency of a pattern’s occurrences at a step, or the probability of undesired events occurring.

In order to perform stochastic simulation experiments on SGTS, the GraTS (Graph-based Stochastic Simulation) tool is used [48]. GraTS is an extension to the VIsual Automated model TRAnsformations (VIATRA2) [49] model transformation plugin and developed in Java-Eclipse. It uses SSJ (Stochastic Simulation Java) libraries to deal with random number generation, distribution functions, and other statistics [48]. GraTS uses an XML file to load the user-defined rules and probes.
The stochastic engine receives the set of enabled rule matches (the active events) from the transformation engine and turns them into timed events by assigned to each of them a time value, randomly determined on the basis of the probability distribution which is associated with the event type. Then it sends the event that has been scheduled first back to the transformation engine for execution [34].

Formally, GraTS simulation consists of the following steps [47]:

1. Initialisation: the simulation time $T$ is initialised to 0 and the set of the enabled matches (active events) is obtained from the graph transformation engine. For each active event $e = (p; m)$, a scheduling time $t_e$ is computed by a random number generator (RNG) based on the probability distribution assigned to the event. Timed events are collected as a list ordered by time (event list).

2. At each simulation step:
   
   (a) the first element $k = (e; t)$ is removed from the event list $s$
   
   (b) the simulation time is increased to $t$
   
   (c) the event $e$ is executed by the graph transformation engine
   
   (d) the new event list $s'$ is computed, by querying the engine, removing all the elements that have been disabled, adding to the list an event for each newly enabled match $m$ with time $t' = t + d$; where $d$ is provided by the RNG depending on $F(p)$, with $p$ being the rule of which $m$ is a match, and reordering the list with respect to time

3. the simulation run ends either when the system reaches a deadlock or the maximum time or maximum number of steps is exceeded, the choice of which being left to the user.

GT rules with no action (empty postconditions), no modifications to the host graph, are used as probes in order to derive statistical data by counting occurrences of particular graph structure during a simulation run.
2.6 Practical Use of GTSs

This section presents a brief overview of the graph transformation tools we have used in our approach. As was explained in the previous chapter the first step in our approach is to model the system using graph transformation tools such as AGG, GROOVE, or Henshin. Afterwards, the modelled GT system is exported into the ODE generator to derive the equations then solving those equations using the ODE solver Octave for the analysis purpose. We use AGG to model our systems and the general layout is illustrated in Chapter 7 section 7.2.1.1.

However, the notation used in AGG is not as easy as the other tools to present the rules and patterns. Therefore, in order to easily illustrate them in the thesis we have used Henshin notation which uses colours to integrate elements created, preserved, and deleted as well as integrate left-hand side and right-hand side of the rules together and distinguish by colours. Fig. 9.5 is one of the examples used in Chapter 9 to show graph transformation rules of the P2P content policing system.

In the bottom right the green colour represents the create action which occurs in the right-hand side of the rule switch_friend, red colour shows that the existing edge in the left-hand side between the users will be deleted in the right-hand side, grey colours (nodes and edges) mean they are preserved in both left and right-hand sides, and blue colour indicates the negative application condition.

2.7 Steady State Analysis

A system is in a steady state if the number of occurrences of the patterns in the system are unchanging in time. This means for that pattern \( p \in P \), the \( \frac{dp}{dt} \) is zero and remains so.

In Continuous Stochastic Logic (CSL) \( S_{\Delta P}(\Phi) \), \( S \) represents the steady state probability of the state \( \Phi \) by means the probability of being in state \( \Phi \) in the
long run. For example, \( S_{\gamma}(s') \) questions what is the long-term probability of being in a state \( s' \)? [29].

In our thesis, we analysis non-functional properties of systems in the steady states using different approaches. The property we take as a characteristic measurement to compare the approaches against each other is Weighted average Frequency of Patterns in the Steady state (WFPS), i.e., the average number of occurrences of a pattern \( p \in P \) in all states, weighted by the probability of states in the steady state.

Figure 2.13: P2P content policing GT Rules.
\[ WFPS(p) = \sum_{s \in S} Pr(s) \times [p]_s \]

\( Pr(s) \) is the steady-state probability of the state \( s \) and \([p]_s\) the number of occurrences of the pattern \( p \) in state \( s \). While this refers to CTMCs, \( WFPS(p) \) from simulations can be determined using the long-term average or final value of \([p]\), this will be discussed in Chapter 5. For differential equations it is obtained by solving for \( d[p]/dt = 0 \), or reading out the stable value of the numerical solution, more details will be discussed in Chapter 8.

### 2.8 Ordinary Differential Equations (ODEs)

An ordinary differential equation (also called ODE, diff eq, or diffy Q) is an equality involving a function and its derivatives. An ODE of order \( n \) is an equation of the form:

\[ F\left(x, y, y', ..., y^{(n)}\right) = 0 \]

where \( y \) is a function of \( x \), \( y' = dy/dx \) is the first derivative with respect to \( x \), and \( y^{(n)} = d^n y/dx^n \) is the \( n \)th derivative with respect to \( x \) [7].

A differential equation states how a rate of change (a differential) in one variable is related to other variables, i.e., the rate of change of patterns with respect to time. This describes the evolution of a model exploring the interdependency between changes in nodes’ state and relationships between them.

In order to state the dynamics of an adaptive network in terms of ODEs, for each equations describing the evolution of a pattern we must know the number of matches of the rules and the rules’ effects on the pattern’s occurrences, such as how many of them are created and destroyed by one rule application, and the rate for the rule. Chapters 6, 7, 8, and 9 will cover these discussions.
2.9 Pair Approximation

In general, neither the number of matches of the rules nor their effects on the patterns can be determined exactly, so we work with approximations capturing their averages by decomposition of complex patterns by simpler ones.

The decomposition which is used for deriving the differential equations is akin to the concept of pair approximation which is common in statistical physics and the study of complex networks [22]. The quality of this type of approximation depends on the particular model at hand, and will generally, but not always, become asymptotically exact as the underlying graph size goes to infinity, which is when one needs it most. This detail will be discussed in Chapter 6.

2.10 Critical Pair Analysis

Critical pair analysis (CPA) enables the detection of all potential overlaps of a set of transformation rules and rules without action, we call them patterns. The result is a set of critical pairs, each reporting a conflict or dependency between a rule and a pattern [5]. A critical pair is a pair of \((r, p)\), where \(r(m1) : G \Rightarrow H1\) and \(p(m2) : G \Rightarrow G\) which are in conflict, and such that graph \(G\) is minimal, i.e., \(G\) is gluing of the left-hand sides of the rules \(r\) and pattern \(p\). It can be computed by overlapping \(L\) of \(r\) and \(p\) in all possible ways, such that the intersection of \(L\) and \(p\) contains at least one item that is deleted or changed by the application of the rule [2]. The concept of critical pair analysis are well-defined in [28].

Two kinds of relations among transformation rules and patterns can be detected: conflicts and dependencies. Conflicts detect the number of occurrences of patterns that are destroyed, i.e., the application of a rule causes an element of the pattern to be deleted/changed. In the case of a dependency, the application of a rule creates the pattern occurrences on the produced model. There are four
possible types of conflicts [5]

- **Delete-use-conflict**: A rule application deletes a graph element which is in the match of the pattern.

- **Produce-forbid-conflict**: A rule application generates graph objects which is prohibited by a NAC of the pattern.

- **Change-use-conflict**: One rule application changes attributes being in the match of the pattern.

- **Change-forbid-conflict**: The application of a rule changes the value of an attribute in such a way that it becomes equal to a forbidden value specified in the pattern.

The following describes the different kinds of dependencies:

- **Produce-use-dependency**: A rule application creates a graph object which is required by the pattern.

- **Delete-forbid-dependency**: A rule application deletes an element which is forbidden by a NAC of the pattern.

- **Change-use-dependency**: It is similar to the produce-use-dependency with the focus on the attribute value. Here the rule application changes the attribute that results of fulfilling the pattern.

- **Change-forbid-dependency**: The rule application changes the value of an attribute that is prohibited by the pattern.

The critical pair analysis is used in our approach to compute rules’ effects on patterns, and the AGG tool’s CPA is used in implementing our algorithm. These will be detailed in Chapter 7.
Chapter 3

Social Voter Model: A Case Study

In order to validate our approach as well as existing ones, we are going to present a case study of a simple but representative example of adaptive networks. The model is adapted from [44] and some of the notations and the ideas are extracted from the paper [32] which I am a co-author of. The model describes the behaviour of opinionated agents which vote for one of two parties and are less likely to stay friendly in case they disagree, i.e., a node connected to another of different opinion will either convert (state change) or disconnect and establish a new connection with a node of the same opinion (topological evolution).

[44]’s original model is based on simple, symmetric graphs whose degree is bounded by 3 for analysis purposes, and where rules are described informally in English. [32]’s model replaces this system by a Kappa-like rewriting system on site graphs, i.e., graphs over the metamodel, where sites control and identify the attachment of links to Agent nodes. Such an encoding is always possible for a system of bounded degree. Our model does not restrict the structure. This discussion will be held in the related work in Section 10.4.2.

In order to compare results of each analysing method (model checking, stochastic simulation, and ODE), we perform quantitative analysis on the Voter model by asking questions about the following properties for each approach:

- What is the evolution over time of the number of agents holding certain
votes?

- What is the evolution over time of the number of edges connecting agents of the same vs. those of different votes?

- What are the steady state values of these numbers?

The following sections present the model in detail.

### 3.1 Type Graph

The type graph of the social voter model is shown in Fig. 3.1. We represent votes as node attributes 0,1 which indicates the opinion of the agents. The nodes, Agent type, are linked to each other by connection edge type.

![Type Graph for Voter Model](image)

Figure 3.1: Type Graph for Voter Model

### 3.2 Graph Constraints

As it was discussed in the previous chapter that graph constraints are additional conditions on graphs that are not expressed by the type graph [16]. It can be used to define permitted or forbidden subgraphs. Negative constraints require the absence of certain structures and positive constraints, require their presence. The negative constraints in our case study are given in Figs. 3.2 to 3.4. V-vote means no agent has two vote attributes. Unlike UML object diagrams whose notations we adopt, attributed graphs (formally E-graphs, see definition
in Section 2.2.4) can have multiple values for the same node attribute. Therefore, the constraints given in the Fig. 3.2 rule out such cases. \textit{PAR-A2A} in Fig.

\begin{center}
\begin{tabular}{c|c}
AgentX: Agent & \\
\hline
Vote & 1, 0
\end{tabular}
\end{center}

Figure 3.2: Graphical Constraints - Node have two attribute values

3.3 means that there must be no parallel edges between the agents. \textit{LOOP-A2A}

\begin{center}
\begin{tabular}{c|c}
\hline
: Agent & : Agent \\
\hline
connection & connection
\end{tabular}
\end{center}

Figure 3.3: Graphical Constraints - Parallel Edges

in Fig. 3.4 means that there is no loop. [32]

\begin{center}
\begin{tabular}{c|c}
AgentX: Agent & \\
\hline
Connection1: connection & \\
\end{tabular}
\end{center}

Figure 3.4: Graphical Constraints - Loops

Positive constraints are required in our model, such as \textit{SYM} in Fig. 3.5, which means that link edges are symmetric. In the graphs we show them as undirected edges. \textit{A-vote} in Fig. 3.6 means each agent has a vote [32]. The shaded arrows in the Figs. 3.5 and 3.6 are for the purpose of implies.

Transformation rules should maintain these constraints. If a rule is applied to a graph that satisfies the constraint, the resulting graph should also be valid
and satisfy the same constraint. This property of our rules are manually verified by applying the rules to valid graphs and checking the results.

These constraints are constructed in the case study for the specification of implementing the model in the GROOVE, VIATRA2 and AGG tools which will be discussed in Chapters 4, 5 and 8.

### 3.3 Start Graph

A start graph, know as host graph as well, represents the initial state in our model. Many start graphs are used throughout the thesis, though only two main graphs are shown in this section where they are used in the Voter model experiments. We work with stochastic graph transformation systems following the DPO approach [17] based on typed attributed directed multi graphs. Though, in this case study we show undirected edges they are represented as symmetric pairs of directed edges.

Fig. 3.7 illustrates an example of an ad hoc host graph which is used in comparing the model checking, simulation and ODE models. It is the smallest possible graph that shows all behaviours of the system. The graph consists of five nodes and 6 edges. Each node either has an attribute vote=1 or vote=0.
and also has a source relation entity and the target relation entity. The graph is designed in a way that has possibilities for the rules to apply. It also presents all the patterns that we are interested to observe.

The second start graph, shown in Fig. 5.12, is used for the validation of both simulation and ordinary differential equation models results. The graph is created randomly but with the same structure and population in both models. It consists of 48 nodes, 8 nodes of vote=0, 40 nodes of vote=1, and 60 edges. The details will be discussed in Chapter 5.

![Figure 3.7: Start Graph used in GROOVE and VIATRA2](image)

### 3.4 Patterns

A pattern is a graph that is relevant for observing its behaviour in the network evolution and analysing it. For the Voter model we are interested in occurrences of patterns that covering the basic states of individual nodes and edges. Some of them are the left-hand side \( L \) for the rules which they will be presented in Section 3.5. All the patterns are explained in the following sections.
3.4.1 Pattern [0]

The pattern represents an individual agent that has a vote=0, see Fig. 3.8.

![AgentX: Agent]

vote = 0

Figure 3.8: Pattern [0]

3.4.2 Pattern [1]

Pattern [1] represents an individual agent that holds a vote=1, see Fig. 3.9.

![AgentY: Agent]

vote = 1

Figure 3.9: Pattern [1]

3.4.3 Pattern [00]

Pattern [00] represents an edge that connects two agents with similar opinions, i.e., vote=0. See Fig. 3.9.

![AgentXX1: Agent]

vote = 0

![ConnectionX: connection]

![AgentXX2: Agent]

vote = 0

Figure 3.10: Pattern [00]
3.4.4 Pattern [11]

Pattern [11] represents an edge that connects two agents with similar opinions, i.e., vote=1. See Fig. 3.9.

![Figure 3.11: Pattern [11]](image)

3.4.5 Pattern [01]

Pattern [11] represents an edge that connects two agents holding different opinions, i.e., vote=0 and vote=1. See Fig. 3.12.

![Figure 3.12: Pattern [01]](image)

3.5 Graph Transformation Rules

Graph Transformation rules are used to define actions that lead from one graph to another (action rules), as well as quantitative observations over graphs. Rules model the coevolution of votes and connections: If two connected agents hold different votes, either one is converted to the opinion of the other (rules convertTo0, convertTo1), see Figs. 3.13 and 3.14, or the link between them is broken and one makes a new connection to an agent of the same opinion they are not
presently connected to (rules splitTo0,splitTo1), shown in Figs. 3.15 and 3.16.

Figure 3.13: Rule convertTo0

Figure 3.14: Rule convertTo1

Figure 3.15: Rule splitTo0
In order to reflect their relative time, rates are defined by assigning each rule a positive real number. For example to make switching more likely than connecting we can assign rates $\rho(convertTo0) = \rho(convertTo1) = 1$ and $\rho(splitTo1) = \rho(splitTo0) = 10$. In this case, given the same number of matches, $splitTo1, splitTo0$ are equally likely, each 10 times as likely as $convertTo0, convertTo1$. However the probability of applying a rule not only depends on its rate but also on its number of matches. For example, if there are no alternative candidates to connect to, $splitTo1, splitTo0$ cannot be applied despite the higher rate. Equally, if there is a larger number of candidates for splitting, $splitTo1, splitTo0$ will find more matches and applications will increase.
Chapter 4

Stochastic Model Checking

Model checking is an analysis technique that aims to generate the entire labelled transition system LTS for a model with the aid of tools such as GROOVE (GRaphbased Object-Oriented VERification). GROOVE is a model checking tool for graph transformation systems. It uses graphs to represent state snapshots, and transitions arise from the application of graph transformation rules [39]. By transforming the LTS into continuous-time Markov chains CTMCs, we will be able to perform a stochastic analysis for the model. The aim of this chapter is to perform a quantitative analysis for the Voter model such as the number of population, i.e., the number of occurrences of the patterns, in the steady state. The state of the system become steady or stabilise when the occurrences of the patterns do not change anymore and become stable, however that does not mean the system ends. The analysis results will be used for the comparison between the model checking and simulation methods. The discussion of the comparison will be detailed in Chapter 10.

In this chapter, following the translation of the type graph, start graph and graph transformation (GT) rules into GROOVE syntax in Section 4.1, we will discuss generating Labelled Transition System (LTS) from graph transformation systems in Section 4.2. Then transform them into continuous-time Markov chains CTMCs in Section 4.3 in order to assign every transitions in LTS a
stochastic rate and carry out the stochastic analysis, such as steady state analysis, by using Performance Evaluation Process Algebra (PEPA) in Section 4.4. We conclude in the same section with presenting and analysing the results.

4.1 GROOVE Model Notation

In this section, we present the GROOVE syntax that will be used for representing the graphical Voter type graph, host graph, GT rules, patterns and constraints defined in Chapter 3.

GROOVE is based on directed node- and edge-labelled graphs. Graph nodes are depicted as boxes, and edges as arrows between them. The node labels are inscribed in the nodes, the edge labels along or on top of the arrows [42]. There are two views in GROOVE, Edit view and Display view. The type graph in Fig. 4.1 and start graph in Fig. 4.2 illustrate both views.

In the Edit view, node labels and edge labels are distinguished from one another. If a node label were prefixed with \textit{type:}, GROOVE will interpret the label as type. If the prefix is omitted, then the label will be interpreted as an edge label. In the Display view, types are set bold [42]. Dealing with \textit{Boolean} datatype is easier than \textit{Integer}, therefore the node attribute values \{0,1\}, i.e., the agents’ votes, are represented as \textit{false} and \textit{true} respectively. Table 4.1 shows the overview of Edit prefixes which are used in our mapping process, including the GT rules mapping.
Table 4.1: Edit Prefixes

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>type:</td>
<td>Declare the remainder of the text to be a node type</td>
</tr>
<tr>
<td>use:</td>
<td>Declare a node or edge to be a reader (the default false)</td>
</tr>
<tr>
<td>del:</td>
<td>Declare a node or edge to be an eraser</td>
</tr>
<tr>
<td>new:</td>
<td>Declare a node or edge to be a creator</td>
</tr>
<tr>
<td>cnew:</td>
<td>Declare a node or edge to be a conditional creator</td>
</tr>
<tr>
<td>not:</td>
<td>Declare a node or edge to be an embargo</td>
</tr>
<tr>
<td>not:=</td>
<td>Injectivities and Negations</td>
</tr>
<tr>
<td>bool:</td>
<td>On nodes, a boolean value or type; on edges, a boolean operator</td>
</tr>
<tr>
<td>let:</td>
<td>Declare a node attribute to be a creator</td>
</tr>
<tr>
<td>test:</td>
<td>Attribute condition that must be satisfied for a rule to apply</td>
</tr>
<tr>
<td>par:</td>
<td>Anonymous or numbered rule parameter node</td>
</tr>
<tr>
<td>rem:</td>
<td>Remark node or edge; used for documentation purposes</td>
</tr>
</tbody>
</table>
Formally, GT rules consist of left-hand sides, right-hand sides, and negative application conditions (NACs), all of which are different graphs connected by morphisms. In GROOVE, graphs are all merged into one single view, and colour coding is used to distinguish the origin components. The components and their colours are shown as follow [42]:

- **Readers**: These are nodes and edges that are in both the LHS and the RHS and their colour is black. See the convert rules in Figs. 4.3 and 4.4.

- **Erasers**: These are nodes and edges that occur in the LHS but not the RHS, meaning that they must be matched in order for the rule to apply, but by applying the rule they will be deleted. In our case study, no nodes are deleted, therefore Erasers will be only used to delete the edges. Such elements are depicted by a thin, dashed blue outline and blue text. In the Edit view, erasers are distinguished by a prefix ‘del:’. See the rule in Fig. 4.5.

- **Creators**: These are nodes and edges that occur in the RHS but not the LHS, meaning that they will be created when the rule is applied. Creators are depicted by a slightly wider, solid green outline and green text. In the Edit view, creators are distinguished by a prefix ‘new:’. See Fig. 4.6.

- **Embargoes**: These are nodes and edges that are in a NAC, but not in the LHS. This means that they forbidden: their presence in the host graph will prevent the rule from being applied. Such elements are depicted by a wide, dashed red outline and red text. In the Edit view, embargoes are distinguished by a prefix ‘not:’. See Fig. 4.8(a).

- **Conditional creators**: These are nodes and edges that occur both in the RHS and in a NAC but not in the LHS. This means that these elements should not be there before the rule is applied, but will be created by the rule application. In other words, the effect combines that of creators and embargoes. Conditional creators are depicted by an overlay of the creator
and embargo attributes, namely by a spiked green-and-red outline and green text. In the Edit view, conditional creators are distinguished by a special prefix cnew:. See Fig. 4.8(b).

The GT rules used in our case study contain all the above types of elements and are illustrated from Figs 4.3 to 4.6. For each of the rules shown we also assume its reverse rule. These are needed in order to ensure that the labelled transition system is strongly connected, and so leads to an irreducible Markov Chain, see Figs. 4.7 and 4.8. The constraints and patterns are also defined in the tool, Fig. 4.9 shows an example of 00 pattern.

Figure 4.3: Rule convertTo0

Figure 4.4: Rule convertTo1
Figure 4.5: Rule splitTo0

Figure 4.6: Rule splitTo1
4.2 Generation of the LTS

After modelling the Voter model as a graph transformation system in GROOVE, an LTS is generated from the start graph of that graph grammar.
All possible applications of rules on the start graph showed in Fig 4.2 and the resulting graphs produce the systems state space of 100 states and 2336 transitions. This provides all possible evolutions of the system. The result is shown in Fig. 4.10. By default, the transitions bear the names of the rules, that have been applied, as labels. We were only able to explore the state space for a graph scaled up to 10 nodes and 9 edges as the scalability issues were emerging every time the system exceeded beyond that size.

4.3 Transforming the LTS into a CTMC

Analysis techniques for CTMCs allow us to perform stochastic analysis on a system, such as steady-state analysis. This section shows how a CTMC is derived from the generated transition system. Generating CTMCs from LTSs requires additional information not present in the LTS, such as rates. A CTMC assigns every transition in LTS a rate (stochastic delay). For this purpose, we have used a tool, CTMC Generator, that was developed for research purpose in the thesis [8]. Fig. 4.11 shows the interface of the CTMC Generator tool.
The first file (lts.gxl) is the generated LTS of the Voter model and the second (rate.txt) is the rules’ rate file, see Section 2.5 for more information about rates.

Figure 4.11: CTMC Generator also shows the input files

Generated CTMCs could be analysed in any tool supporting CTMCs for stochastic analysis. In our work, we have used Performance Evaluation Process Algebra (PEPA) which is a standard tool for performing such analysis. In the tool, each process is assigned certain activities, i.e, transitions. When these activities are performed a process transforms to the resulting processes [8]. The output of the tool is a PEPA model and some fragments of the result are listed below.

Listing 4.1: VoterModel.pepa file

```
// PEPA model
// Rates for the rules
convertTo0 = 10; convertTo1 = 10; splitTo0 = 1; splitTo1 = 1;
prob0 = 0.000000000001; prob1 = 0.000000000001; prob00 = 0.000000000001;
prob11 = 0.000000000001; prob01 = 0.000000000001;

P0 =
("prob0(n8)",prob0).P0 +
("prob0(n3)",prob0).P0 +
("prob00(n3,n8)",prob00).P0 +
("prob00(n8,n3)",prob00).P0 +
("prob01(n8,n7)",prob01).P0 +
("prob1(n8)",prob1).P0 +
("prob1(n7,n1)",prob11).P0 +
("prob1(n1,n7)",prob11).P0 +
("prob1(n9,n7)",prob11).P0 +
("prob1(n7,n9)",prob11).P0 +
("convertTo1(n7,n1)",convertTo1).P1 +
```

56
\begin{verbatim}
P97 =
    ("convertTo1(n3,n8)",convertTo1).P94+
    ("convertTo1(n7,n8)",convertTo1).P94+
    ("convertTo1(n3,n9)",convertTo1).P94+
    ("convertTo1(n7,n9)",convertTo1).P94+
    ("prob11(n3,n8)*,prob11).P97+
    ("prob11(n8,n9)*,prob11).P97+
    ("prob11(n3,n9)*,prob11).P97+
    ("prob11(n8,n3)*,prob11).P97+
    ("prob11(n9,n3)*,prob11).P97;
\end{verbatim}

Process variables such as P0 correspond to a state such as s0 in the LTS. An equation \( P0 = ("convertTo1(n7,n1)",convertTo1).P1+ \) means that there exists a transition from P0 to P1 with rate \( convertTo1 = 10 \), labelled \( convertTo1(n7,n1) \). Whereas \( ("prob0(n8)",prob0).P0+ \) is for observing the evolution of pattern \([0]\) without any action or change to the system.

### 4.4 Stochastic Analysis: PEPA

PEPA [21] is a standard analysing tool which supports CTMC to analysis non-functional properties of a model. The property we want to analysis is the Weighted average Frequency of Patterns in the Steady state (WFPS) as it was discussed in Section 2.7. For a pattern \( p \in P \), the WFPS is

\[ \text{WFPS}(p) = \sum_{s \in S} Pr(s) \times [p]_s \]

A fragment of \( Pr(s) \) and \( [p]_s \) results are shown in Tables 4.2 and 4.3 respectively. Afterwards, the results are integrated to produce the WFPS for each pattern across all states, see Table 4.4. As it is seen in the table, the average number of occurrences of population have certain votes is \( \approx 2.50 \) across all the states, while pattern \([00]\) and pattern \([11]\) are \( \approx 1.7028 \) and pattern \([01]\) is \( \approx 4.2971 \).
<table>
<thead>
<tr>
<th>State</th>
<th>Rule</th>
<th>Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>P76</td>
<td>prob0</td>
<td>1</td>
</tr>
<tr>
<td>P76</td>
<td>prob1</td>
<td>4</td>
</tr>
<tr>
<td>P76</td>
<td>prob11</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 4.2: Number of Pattern occurrences in certain States

<table>
<thead>
<tr>
<th>State</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0.003163011</td>
</tr>
<tr>
<td>P1</td>
<td>0.014894982</td>
</tr>
<tr>
<td>P10</td>
<td>0.052784279</td>
</tr>
<tr>
<td>P11</td>
<td>0.018021531</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 4.3: The Probability of each State

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Weighted Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\approx 2.50$</td>
</tr>
<tr>
<td>1</td>
<td>$\approx 2.50$</td>
</tr>
<tr>
<td>01</td>
<td>$\approx 4.3$</td>
</tr>
<tr>
<td>00</td>
<td>$\approx 1.7$</td>
</tr>
<tr>
<td>11</td>
<td>$\approx 1.7$</td>
</tr>
</tbody>
</table>

Table 4.4: Weighted Average Frequency of Patterns

These results will be used as input for the comparison with the simulation result, which they are expected to be fairly similar. This will be discussed in Evaluation chapter 10, Section 10.1.1
In this chapter we present the second analysis technique, stochastic stimulation. The chapter describes the translation and encoding of the Voter model into VIATRA2 [49] syntax in order to perform stochastic simulation using the Graph-based Stochastic Simulation (GraTS) tool [47] [48]. GraTS enable the execution of simulation experiments made of multiple batches of bounded runs. It can compute statistics about rule application, timing, and patterns. The purpose of the simulation is to compare its analysis outcomes with the results of model checking and ordinary differential equation ODE methods. The simulation links both methods.

In this chapter, we conduct two simulation experiments. One experiment is on a small scale in order to compare the results with the model checking and ODE ones, the other experiment has a large and random start graph to compare only with the ODE result. The reason of having two experiments is due to the scalability issues of the approaches which is explained throughout the chapter particularly in Section 5.7.

The type graph and start graph which they are called metamodel and model respectively in VIATRA2 are contained within a single VIATRA2 model space using the model space tree editor and Visual and Precise Modelling (VPM) metamodel. Section 5.1 explains how the metamodel, model, rules and patterns
in sections 5.2, 5.3, 5.4 and 5.5 are mapped into VIATRA2 syntax. Then
the simulation parameters are defined in Section 5.6 and Section 5.7 discusses
the simulation experiments. We conclude in Section 5.8 with presenting and
analysing the results.

5.1 Modelling Notation

This section presents a mapping of the graphical Voter model into VIATRA2
textual syntax which will be used in the Sections 5.2, 5.3, and 5.4. Figure 5.1
shows the mapping of the graphical Voter metamodel and Figure 5.2 illustrates
the model mapping.

In Fig. 5.1, *entity* is used to define a new node type (Agent) and *relation* is
used to create an edge type between two nodes. The “.” symbol is used for the
purpose of subentity, edge and attribute of a node, whilst the “,” symbol for the
separating purpose. To define an attribute for a node type in VIATRA2, first
the attribute (e.g. vote) is defined by *entity*, then the node and its attribute are
connected using *relation* and finally a datatype (e.g. Boolean) is given to the
attribute using *supertypeOf*. Almost the same style is used in the Fig. 5.2 to
map the instance graph that is typed over the type graph. `InstanceOf` is used to type instances of nodes, edges, and attributes over their types in the type graph. Section 5.4 presents more details about mapping the graphical GT rules into VIATRA2 textual syntax.

The syntax for the definition of patterns is as follows:

```plaintext
pattern pattern name (parameters) =
{
    Pattern body
}
```

The `Pattern` keyword is used to define Patterns. Patterns may have parameters that are declared after the pattern name. The pattern body contains model element and relationship definitions. To explain a pattern we consider an example pattern below:

```plaintext
pattern splitTo0(Agent1,Agent2,Agent3,Vote1,Vote2,Vote3,R1,R2)=
{
    Agent(Agent1); \ Agent1 is variable/instance of type Agent
    Agent(Agent2);
    Agent(Agent3);
    entity(Agent1);
    instanceOf(Agent1,'Agent');
    entity(Agent1)
    {
        entity(Agent1.'vote1')="true";
    }
    relation(Agent1.'attr1',Agent1,'Agent1.'vote1);
    instanceOf(Agent1,'Agent');
    instanceOf(Agent1.'vote1',Agent.'vote1');
    instanceOf(Agent1.'attr1',Agent.'attr1');

    entity(Agent2)
    {
        entity(Agent2.'vote2')="false";
    }
    relation(Agent2.'attr2',Agent2,'Agent2.'vote2');
    instanceOf(Agent2,'Agent');
    instanceOf(Agent2.'vote2',Agent.'vote');
    instanceOf(Agent2.'attr2',Agent.'attr');

    relation(Agent1.'con1',Agent1,'Agent1');
    relation(Agent2.'con2',Agent2,'Agent1');
    instanceOf(Agent1.'con1',Agent.'Connection');
    instanceOf(Agent2.'con2',Agent.'Connection');
}
```

Figure 5.2: VIATRA2 Model Notation
This example shows the LHS of the splitTo0 rule. The pattern has three agents, two of them of different opinions are connected. The check conditions is used to verify the node’s attribute value. The keyword neg identifies a sub-pattern that is embedded into the current one to represent a NAC. The NAC of the rule is there should not be an edge between the two agents that have the same opinion. Since we already have another pattern that connects two agents of similar opinion, we can call that pattern using the find keyword.

As the graph patterns define the logical conditions on the model, the manipulation of models is achieved using graph transformation rules. The process is based on graph patterns for defining the application criteria for transformation steps. A GT rule consists of a precondition, a postcondition and an optional action part. The application of a GT rule on a given graph replaces an image of its precondition pattern (LHS) with an image of its postcondition pattern (RHS). The syntax for transformation rules in VIATRA2 is given below:

gtrule rule name (parameters) = {
    precondition
    
    pattern definition
The precondition, the LHS of a rule, defines the condition for the application of a rule, its execution means finding a pattern in the start graph. The postcondition pattern, the RHS of a rule, is the result of applying a GT rule. The postcondition allows three different operations on the graph, namely preservation of an input parameter, deletion, and creation. The action part to the GT rule can be used to perform the same task as would be performed by the postcondition, with additional flexibility of user defined functions or methods. A GT rule can create model elements, delete existing ones and update values of attributes [34]. In this thesis, we are using GT rules with preconditions and actions.

5.2 Metamodel in VPM

This section presents the mapping result, as described in Section 5.1, of the type graph, know as metamodel in VIATRA2, of the Voter social network in Section 3.1 to VIATRA2 (VPM). A small part of the source code is shown below:

```
entity('metamodel'.'Agent')=""
{
    entity('metamodel'.'Agent'.'vote')="";
}
relation('metamodel'.'Agent'.'Connection','metamodel'.'Agent','metamodel'.'Agent',"Agent");
```
The metamodel can also be created and edited using the model space tree editor as shown in Fig. 5.3. The overview of the metamodel is shown in Fig. 5.3(a), whilst Fig. 5.3(b) illustrates the relational and sub entity details of the metamodel. Fig. 5.4 shows how the metamodel is visualised in VIATRA2.

Figure 5.3: Voter Metamodel
5.3 Model in VPM

This section presents the encoding start graph of the Voter model in the first experiment in Section 5.7.1 into VIATRA2 using the syntax defined in Section 5.1. A fragment of the source code is listed below:

```java
entity('startGraph', 'model')->
{
    entity('startGraph', 'model', 'Agent_FalseuN124_4f')->
    {
        entity('startGraph', 'model', 'Agent_FalseuN124_4f', 'Vote_uN125_4f')—"false";
    }

    entity('startGraph', 'model', 'Agent_TrueuN125_3c')->
    {
        entity('startGraph', 'model', 'Agent_TrueuN125_3c', 'Vote_uN126_3c')—"true";
    }

    ...}

relation('startGraph', 'model', 'Agent_FalseuN124_4f', 'uN126_4f', 'startGraph', 'model', 'Agent_FalseuN124_4f', 'startGraph', 'model', 'Agent_FalseuN124_4f', 'Vote_uN125_4f');

relation('startGraph', 'model', 'Agent_TrueuN98_3c', 'uN100_3c', 'startGraph', 'model', 'Agent_TrueuN98_3c', 'startGraph', 'model', 'Agent_TrueuN98_3c', 'Vote_uN99_3c');

...}

instanceOf('startGraph', 'model', 'Counter1', 'metamodel', 'Counter');
instanceOf('startGraph', 'model', 'metamodel');
5.4 Model GT Rules in VIATRA2 Syntax

In this section, the GT rules that have been presented in the Section 3.5 are encoded into VIATRA2 [49] syntax. In order to perform stochastic simulation using GraTS [48], the VIATRA2 textual syntax representation of the GT rules is required.

The following sections are presenting the encoding of all the patterns and rules into VIATRA2 syntax.

5.4.1 Patterns

A graph model satisfies a pattern, if the pattern can be matched to the graph. The VIATRA2 code for all patterns, which are graphically presented in Chapter 3, are given in the following sections.

5.4.1.1 Pattern [0]

```viatra2
pattern pattern0(Agent1, Vote1) =
{
    Agent(Agent1) in startGraph.model;
    Agent.vote(Vote1) in Agent1;
    Agent.attr(V1Attr, Agent1, Vote1);
    check(toBoolean(value(Vote1)) == false);
}
```

5.4.1.2 Pattern [1]

```viatra2
pattern pattern1(Agent1, Vote1) =
{
    Agent(Agent1) in startGraph.model;
    Agent.vote(Vote1) in Agent1;
    Agent.attr(V1Attr, Agent1, Vote1);
    check(toBoolean(value(Vote1)) == true);
}
```
5.4.1.3 Pattern [00]

\[
\text{pattern pattern0_0(Agent1, Agent2, R1, R2, Vote1, Vote2) =}
\{
\text{Agent(Agent1) in startGraph.model;}
\text{Agent(Agent2) in startGraph.model;}
\text{Agent.vote(Vote1) in Agent1;}
\text{Agent.attr(V1Attr, Agent1, Vote1);}
\text{check (toBoolean(value(Vote1)) == false);}
\text{Agent.vote(Vote2) in Agent2;}
\text{Agent.attr(V2Attr, Agent2, Vote2);}
\text{check (toBoolean(value(Vote2)) == false);}
\text{Agent.Connection(R1, Agent1, Agent2);}
\text{Agent.Connection(R2, Agent2, Agent1);}
\}
\]

5.4.1.4 Pattern [01]

The following pattern0_1 is the encoding of pattern [01] and also a call pattern for the preorder of the convertTo0 and convertTo1 rules. The syntax is explained in the code comments:

```
//Declarations i.e. Type(Instance).
//Pattern [01] used in both GT rules (convertTo0, convertTo1).
\text{pattern pattern0_1(Agent1, Agent2, R1, R2, Vote1, Vote2) =}
\{
\text{Agent(Agent1) in startGraph.model; \hspace{1cm} //define an instance node Agent1}
\text{Agent(Agent2) in startGraph.model; \hspace{1cm} //define an instance Agent2}
\text{Agent.vote(Vote1) in Agent1; \hspace{1cm} //define an instance node attribute}
\hspace{1cm} \hspace{1cm} \hspace{1cm} //Vote1 for Agent1}
\text{Agent.attr(V1Attr, Agent1, Vote1); \hspace{1cm} //relation between graph node Agent1}
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} //and its attribute Vote1}
\text{check (toBoolean(value(Vote1)) == false); \hspace{1cm} //check that the Agent1 has}
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} //false vote}
\text{Agent.vote(Vote2) in Agent2;}
\text{Agent.attr(V2Attr, Agent2, Vote2);}
\text{check (toBoolean(value(Vote2)) == true); \hspace{1cm} //verify that the Agent2 is true}
\text{Agent.Connection(R1, Agent1, Agent2); \hspace{1cm} //the bidirectional relation between}
\text{Agent.Connection(R2, Agent2, Agent1); \hspace{1cm} //the graph nodes Agent1 and Agent2}
\}
```
5.4.1.5 Pattern [11]

\[
\text{pattern pattern1_1(Agent1,Agent2,R1,R2,Vote1,Vote2) =}
\{
\begin{align*}
\text{Agent(Agent1) in startGraph.model;} \\
\text{Agent(Agent2) in startGraph.model;} \\
\text{Agent.vote(Vote1) in Agent1;} \\
\text{Agent.attr(V1Attr, Agent1, Vote1);} \\
\text{check (toBoolean(value(Vote1)) == true);} \\
\text{Agent.vote(Vote2) in Agent2;} \\
\text{Agent.attr(V2Attr, Agent2, Vote2);} \\
\text{check (toBoolean(value(Vote2)) == true);} \\
\text{Agent.Connection(R1, Agent1, Agent2);} \\
\text{Agent.Connection(R2, Agent2, Agent1);} \\
\end{align*}
\}
\]

5.4.2 Rules

5.4.2.1 ConvertTo0

In this section the VIATRA2 representation of the rule for changing the opinions of agents, convertTo0 rule 3.5, is presented.

//Rule convertTo0
gtrule convertTo0() =
{
    //LHS of the rule
    precondition find pattern0_1(Agent1,Agent2,R1,R2,Vote1,Vote2) // call
    pattern0_1

    //RHS of the rule
    action
    {
        seq
        {
            println ("convertTo0: "$Agent2" converted its opinion to the same as $Agent1, from " + value(Vote2) + " to " + value(Vote1));
            setValue(Vote2,"false"); //Agent2 changes opinion.
        }
    }
}
The **precondition** matches the pattern that has two agents with two different votes, true and false. The **action** is the postcondition of the rule that performs the change. The **setValue** keyword updates the value of attribute `Vote2` from true to false.

### 5.4.2.2 ConvertTo1

The rule **convertTo1** is similar to the previous rule except that the attribute change happens at a different node.

```java
gtrule convertTo1() = 
{
    precondition find pattern0_1(Agent1, Agent2, R1, R2, Vote1, Vote2)

    action
    {
        seq{
            println ("convertTo1: Agent \" + name(Agent1) + ") converted its opinion to the same as \" + name(Agent2) + ", from \" + value(Vote1) + ", to \" + value(Vote2) + ");
            setValue(Vote1, "true");
        }
    }
}
```

### 5.4.2.3 splitTo0

In this section, we present the rule **splitTo0**. If two connected agents hold different votes, *false* and *true*, the link between them is broken and the agent that has vote *false* makes a new connection to another agent of the same opinion. The syntax is explained in the code comments.

```java
gtrule splitTo0() =
{
    \call pattern splitTo0.
    precondition find splitTo0(Agent1, Agent2, Agent3, Vote1, Vote2, Vote3, R1, R2)

    action
    {
        let R3=undef, R4=undef in \define new edge instances
    }
}
```
5.4.2.4 splitTo1

This is similar to the previous rule except that the reconnection of the nodes happens between the agents that have a true vote.

grule splitTo1() =
{
  precondition find splitTo1(Agent1, Agent2, Agent3, Vote1, Vote2, Vote3, R1, R2)

  action
  {
    let R3=undef, R4=undef in
    seq
    {
      delete(R1); //delete the bidirectional
      delete(R2); // relation between the two graph nodes Agent1 and Agent2
      new(Agent.Connection(R3, Agent1, Agent3)); //reconnect the Agent1 with Agent3
      new(Agent.Connection(R4, Agent3, Agent1));
      println("splitTo1: Agent " + name(Agent1) + " split up with " + name(Agent2) + " and reconneted with " + name(Agent3));
    }
  }
}
5.5 Model Probes in VIATRA2 Syntax

This section presents the probes in VIATRA2 textual syntax. A probe is a rule as well as a pattern that captures a particular pre-condition during a simulation run. Probes have similar format as GT rules in VIATRA2, except that postconditions are empty, i.e., they have no actions. Therefore, probe rules do not have an effect on the model. Not to be confused with the formal GT rules when the right-hand sides are empty that indicates the deletion. In VIATRA2, an empty right-hand side of a rule means no action is detected therefore that rule is called probe. The probes are required for analysing the simulation results based on pattern matching with probes to help solve the initial questions in Chapter 1.

The following VIATRA2 code outlines a probe rule structure:

```vbnet
Probe_Name() =
{
    precondition pattern LHS() =
    {
        // pattern to match
    }
    Action
    {
        // No performance
    }
}
```

The following are all the probes that are defined in our simulation:

5.5.1 Probe [01]

```vbnet
//probe for pattern[01]
gtule Probe_pattern0_1() =
{
    precondition find pattern0_1(Agent1,Agent2,R1,R2,Vote1,Vote2)
    action
    {
        println("Pattern[01]");
    }
}
```
5.5.2 Probe [00]

//probe for pattern[00]
gtrule Probe_pattern0_0() =
{
    precondition find pattern0_0(Agent1, Agent2, R1, R2, Vote1, Vote2)

    action
    {
        println("Pattern[00]";
    }
}

5.5.3 Probe [11]

gtrule Probe_pattern1_1() =
{
    precondition find pattern1_1(Agent1, Agent2, R1, R2, Vote1, Vote2)

    action
    {
        println("Pattern[11]";
    }
}

5.5.4 Probe [0]

//probe for pattern[0]
gtrule Probe_pattern0() =
{
    precondition find pattern0(Agent1, Vote1)

    action
    {
        println("Pattern[0]";
    }
}

5.5.5 Probe [1]

//probe for pattern[1]
gtrule Probe_pattern1() =
{ } precondition find pattern1(Agent1, Vote1)

action
{
    println("Pattern[1]");
}

5.6 Simulation Parameters

Running a simulation in the GraTS [48] tool requires:

- A GTS: The type graph, the initial model (start graph), the transformation rules as a VTCL file.
- Setting the execution parameters: External parameters and internal parameters.
- Setting the stochastic structure.

The GTS is already presented in the previous sections, the next step is defining the stochastic structure and execution parameters.

5.6.1 Stochastic Structure

The Stochastic structure is defined in an XML file (stochastic input file) and it includes the stochastic control type for the transformation rules (which define how to use the rules with respect to transformation and probing), event sorts associated to action rules, assignments of cumulative distributions functions 'cdf' to the action rules, and optional stochastic variation control values. Transformation rules are assigned types which are alphabetically represented as 'A' for application rule action, 'O' for observable action, 'S' for sensitive action, and 'P' for probe rule [48]. The GT rules defined in the Section 5.4 are referred to as action rules 'A'. In the case of sensitive rules, the cumulative distribution functions cdf can vary through different batches. If a rule is observable, then
the simulator returns statistics about the rule application and delay which they have been applied over each run. If a rule is a probe rule, the simulator provides the statistics of the number of its matches at each steps. Event sorts are represented as integers, 0 is the default. For each action rule, each event sort should be assigned a cdf. If it is an exponential distribution then it is defined as:

```
type: exp (string)
numeric parameter: rate
```

If it is lognormal distribution then it should be like:

```
type: norm (string)
numeric parameter: mean, variance
```

Below is one of the stochastic input XML files that is used in a simulation:

```xml
<allrules>
  <ruleset name="random">
    <rule name="convertTo0" type="A">
      <event name="0" type="exp">
        <rate value="10"/>
      </event>
    </rule>
    <rule name="convertTo1" type="A">
      <event name="0" type="exp">
        <rate value="10"/>
      </event>
    </rule>
    <rule name="splitTo0" type="A">
      <event name="0" type="exp">
        <rate value="1"/>
      </event>
    </rule>
    <rule name="splitTo1" type="A">
      <event name="0" type="exp">
        <rate value="1"/>
      </event>
    </rule>
    <rule name="Probe_pattern0_1" type="P"></rule>
    <rule name="Probe_pattern0_0" type="P"></rule>
    <rule name="Probe_pattern1_1" type="P"></rule>
  </ruleset>
</allrules>
```
The ruleset name corresponds to the parameter defined in parameters.xml file. Also, the probe definition is shown in the probeset such that the op corresponds to the name of a predefined probe in the VTCL file.

5.6.2 Execution Parameters

External parameters are stored in the StoSimPars element of the VIATRA2 model space (VPML file) Figure 5.5 but internal parameters can either be found in the model space or read from an XML file (stochastic input file).

The entities within StoSimPars consist of Machine, ModelPath, extInputOption, ioInputFolder, and ioPath values. The Machine is derived from combining the model space (VPML file) name and the VTCL file name, e.g., TreeGraph.test6 as it can be seen in the Figure 5.5. This means that the rules in file test6.vtcl have been used for simulation. The ModelPath entity is the path to the model space element that is targeted for the simulation i.e. the startGraph. The ioPath is the absolute path to the folder that contains the GTS specification. The ioInputFolder is the subfolder of ioPath that contains the XML input files i.e., Distributions. The extInputOption is used to decide whether the internal parameters can be found in the model space or read from the XML file. If the value is set to true then the parameters will be read in the XML file i.e., parameters.xml.
A sample of the content of the parameters.xml is shown below:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<allparameters>
  <iopars>
    <parameter name="CDF_Input" value="stoInputFile.xml"/>
    <parameter name="Output_Folder" value="ioLog"/>
    <parameter name="Output_Files" value="log"/>
  </iopars>
  <runpars>
    <parameter name="Rule_Set" value="random"/>
    <parameter name="Depth_Limit" value="2000"/>
    <parameter name="Time_Limit" value="50"/>
    <parameter name="Time_Opt" value="false"/>
    <parameter name="Batch_Size" value="1"/>
  </runpars>
  <varpars>
    <parameter name="Variation_Type" value="S"/>
    <parameter name="Variations" value="1"/>
    <parameter name="Factor_Up_Opt" value="True"/>
    <parameter name="Factor_Up_Base" value="1"/>
  </varpars>
  <outpars>
    <parameter name="Confidence_Level" value="0.90"/>
    <parameter name="Hierarchical_Aggregation" value="short"/>
    <parameter name="2D_Linear_Aggregation" value="none"/>
    <parameter name="Debug_Level" value="0"/>
  </outpars>
</allparameters>
```
In this section we explain the stochastic simulation experiments and reasons of conducting them which is to compare the results with the result analyses we obtain from the other analysis techniques, namely model checking and ordinary differential equations.

As it was addressed in earlier chapters that the major bottleneck of using graph transformation to model social networks is the scalability of their analysis such as state space generation in the derivation of the CTMC. This problem emerged while we analysed the Voter model in GROOVE. We were only able to scale the system up to certain size, precisely up to 10 nodes and 9 edges where the time needed to explore the state space become unknown and it eventually ran out of memory. Therefore, in the simulation two different experiments were conducted in order to address this problem. One experiment was in a small size, exactly the same graph size used in previous chapter (5 nodes and 6 edges) such that we were able to compare the results with the model checking analysis results. The second experiment was for a scaled system and the result was to compare to the ODE method.

In each experiment we observe certain properties to analyse. In the first
experiment (small graph), the simulation try to answer the same question was asked in the previous chapter by observing the \textit{WFPS} and we expect the results to be matched with model checking one. \textit{WFPS} can be determined from simulations, using the long-term average or final value of \([p]\). This discussion will be extended in later sections of this chapter.

In the second experiment (larger graph) whose results will be compared to the ODE results, the simulations again address the same question about the WFPS, and also questions about the system’s behaviour which is interesting to show how the system evolves over time. These questions cannot be addressed in the model checking this is due to the limitation of the approach. This will be evaluated in detail in the Section 10.2 of Chapter 10. The questions about the system’s behaviour are:

- What is the evolution over time of the number of agents holding certain votes (pattern [0] and pattern [1])?
- What is The evolution over time of the number of edges connecting agents of the same and those of different votes (pattern [11] or pattern [00] and pattern [01] respectively)?
- What are the resulting long-term ratios, i.e., the resulting steady state of such a system?

The same questions will be asked and the same properties will be observed in ODE approach in Chapter 8. We expect the answers to these questions match in both models. Refer to the Evaluation chapter, Section 10.1.2 for the result analysis comparison.

A simulation experiment that contains a set of runs, where all runs share the same stochastic parameters, is called \textit{batch}. Runs from different batches can vary, either in stochastic assignment or in maximum length. In the next sections we will discuss the two simulation experiments mentioned above.
5.7.1 Experiment 1

The aim of this experiment is to confirm that the simulation obtains the same WFPS as the model checking. For that, in this simulation experiment we use exactly the same graph used for the model checking, see Fig. 5.6. The graph consists of 5 nodes, 2 nodes of \( vote = 0 \) and 3 nodes of \( vote = 1 \), and 6 fixed edges between them. We have represented 0,1 votes as false,true respectively, because dealing with the Boolean datatype is easier than with integers.

The input for this experiment is generated manually the same as in the model checking. One could automate and randomise the input following the procedure provided in the second experiment 5.7.2, but due to the limitation of GROOVE tool, we were unable to produce a random graph in the model checking. For the consistency reason, we have fixed the input for this experiment.

Rates for conversion rules are \( = 10 \), and splits = 1. We have chosen the rates for the rules arbitrarily, because the absolute values are irrelevant since rates do not have a unit. If all rates are multiplied by 10, we obtain exactly the same behaviour of the population in the system. The ratio between the rates of the conversion rules and the split rules matters and is considered for choosing the rates.

The simulation has a batch of 5 runs sharing the same simulation parameters. The parameters were sufficient to produce robust result. The result of this experiment is presented in Section 5.8.1.

5.7.2 Experiment 2

In the previous experiment, the approaches address the question about the Weighted average Frequency of Patterns in the Steady state. A graph of small size is used to compare the result analyses. Whereas in this experiment, besides the Weighted average Frequency of Patterns we also ask about the timing behaviour of the model. The aim of this experiment is to confirm that both methods, the simulation and the ODE, have the same WFPS and the system
evolve similarly in both approach. Also to show that the approaches can address different questions. This discussion will be taken place in the Evaluation Chapter, Section 10.2.

The start graph that is used in this experiment created randomly and it consists of 48 nodes, 8 nodes of $voted=0$, 40 of $voted=1$ and 60 random edges. Rates for conversion rules are 10, and 1 for splits. The rates were chosen arbitrarily as mentioned in the previous section. The simulation has a batch size of 5 runs sharing the same simulation parameters. The same number of nodes and edges will be used as the start state or input model for the ODE approach reflecting the same random graph in the simulation, with the same rates.

Since the social network in our case study is not a real or existing network, thus we do not have real data as an input for our simulation. Also the second experiment uses scaled graphs which cannot be fixed easily, thus it is preferred to automate the generation of the graph and randomly. In order to systematically generate random graphs in VIATRA2, we use a reliable and repeatable method to create a graph where the edges are equally distributed, i.e., uniform distribution between the edges. In the procedure, a random number generator is used to generate a random number between 0 and 1. We use the random
number with the edge probability to decide whether a random pair of nodes are connected by an edge or not. This will be discussed in the phase two of the below procedure.

The parameters are attributes of node Counter in Fig. 5.7. They will be initialised in the beginning of each process.

![Metamodel including the Counter node](image)

Figure 5.7: Metamodel including the Counter node

\[ N: \text{number of nodes of the graph to be created.} \]
\[ T: \text{number of } vote = 1 \text{ nodes, } T \leq N. \]
\[ P: \text{edge probability, the probability that a random pair of nodes is connected by an edge.} \]

The procedure, has three phases and each phase contains a rule which is applied sequentially for as long as possible. Each rule is saved in a separate VTCL file.

**Rule 1**: Creating nodes. (see Fig. 5.8)
Create \( N \) nodes in a loop. Assign \( T \) for \( vote = true \), and the remaining \( N - T \) for \( vote = false \).
Rule 2: Creating edges between the nodes. (see Fig. 5.9)

precondition: two nodes without any edge between them.
postcondition: generate a random number $x$ between 0 and 1. If $x < P$, create NoLink edge. The NoLink edge is used to remember that we considered this pair of nodes already. If $x \geq P$, create Connection edge.

Rule 3: delete the NoLink edge. (see Fig. 5.10)
The VIATRA2 textual syntax code for implementing the above algorithm is available at https://goo.gl/3keiup.

The parameters are defined in the start graph using the model space tree editor as it is shown in Figure 5.11. In this example, the parameter $N = 48$, $T = 40$ and $P = 0.5$.

![Figure 5.11: Defining the Parameters](image)

Bidirectional relations in VIATRA2 are created as two directed edges, see Fig. 5.12. Each node has a source relation entity and target relation entity. But, to ease the illustration, link edges are shown as undirected. Fig. 5.12 shows how the start graph is visualised in the VIATRA2.
Figure 5.12: Viatra Visualisation of Model
5.8 Results

In this section we present the analysis results were obtained from both stochastic simulation experiments.

5.8.1 Results of Experiment 1

This section presents the result of the simulation experiment 5.7.1. The WFPS for all patterns are shown in Table 5.1.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>WFPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>≈ 2.8</td>
</tr>
<tr>
<td>1</td>
<td>≈ 2.3</td>
</tr>
<tr>
<td>01</td>
<td>≈ 2.5</td>
</tr>
<tr>
<td>00</td>
<td>≈ 4</td>
</tr>
<tr>
<td>11</td>
<td>≈ 2.7</td>
</tr>
</tbody>
</table>

Table 5.1: The Weighted average Frequencies of Patterns in Steady state

We expect the results to be similar to the model checking one. The evaluation of the comparison between the WFPS in both approaches will be detailed in Evaluation Chapter, Section 10.1.1.

5.8.2 Results of Experiment 2

In this section we presents the result of the second experiment explained in Section 5.7.2. The result of this section will be used as an input for the comparison purpose with the ODE results analysis.

Fig. 5.13 shows the overall behaviour of the system overtime and Table 5.2 shows the WFPS for each pattern. We expect to obtain similar results from the ODE approach. The result of both methods will be analysed and evaluated in Evaluation chapter, Section 10.1.2.

In order to check that five simulation runs were sufficient to represent reasonable and robust result, another simulation experiment with the same start
Figure 5.13: The evolution of the system over time

<table>
<thead>
<tr>
<th>Pattern</th>
<th>WFPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern [0]</td>
<td>9.8</td>
</tr>
<tr>
<td>Pattern [1]</td>
<td>38.2</td>
</tr>
<tr>
<td>Pattern [00]</td>
<td>27.8</td>
</tr>
<tr>
<td>Pattern [01]</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.2: The Weighted average Frequency of Patterns in Steady state

graph, parameters and a batch size of 20 runs were carried out. The result is illustrate in Fig. 5.14 which indeed shows the similarity to that five runs.
Chapter 6

From Stochastic GTS to ODEs

In this chapter, a technique will be presented to derive ordinary differential equations from stochastic graph transformation systems. The numerical use of differential equations provides another analysis technique for graph transformation systems. The aim is to apply the method on the Voter and P2P content policing case studies in order to be able to answer the analysis questions asked in Chapter 1.

This chapter is organised as follows. Section 6.1 describes the general equation that states the dynamics of adaptive networks. The concept of pattern approximation will be introduced in Section 6.2.
6.1 ODEs for Patterns

In order to state the dynamics of an adaptive network in terms of ODEs, for each equation describing the evolution of a pattern we must know the number of matches of the rules and the rules’ effects on the pattern’s occurrences, such as how many of them are created and destroyed by one rule application, as well as the rate for the rule. For a pattern \( p \in P \) an equation modelling the evolution over time of the number of occurrences \([p]\) of \( p \) is of the form

\[
\frac{d[p]}{dt} = \sum_{r \in R} [L_r] \times \rho(r) \times \Delta(r, p)
\]

where \([L_r]\) is the number of occurrences of \( r \)’s left-hand side in the current graph, \( \rho(r) \) is the rule’s rate and \( \Delta(r, p) \) is the net effect of applying \( r \) on the number of occurrences of \( p \). In general, neither \([L]\) nor \( \Delta(r, p) \) can be determined exactly, so we work with approximations capturing their averages.

6.2 Pattern Approximation

This section presents the approximation of complex patterns by simpler ones which is used for deriving the differential equations. We will first look at the approximation of basic patterns and effects of basic rules, then consider conditional ones.

6.2.1 Approximation of Basic Patterns

Assume patterns \( p = p_1 \cup p_2 \) \(^1\) and let \( p_1 \cap p_2 = p_0 \). We say that \( p_1 \) and \( p_2 \) are independent (in a given graph \( G \)) iff \([p] = [p_1] \times [p_2]/[p_0]\). This is justified as follows: \([p] = [p_1] \times [p_2] \times P(o_1|p_0 = o_2|p_0)\) using the probability \( P(o_1|p_0 = o_2|p_0)\) that an arbitrary pair of occurrences \( o_1 : p_1 \rightarrow G \) and \( o_2 : p_2 \rightarrow G \) overlap in an

\(^1\)Writing \( G \cup H \) or \( G \cap H \) for graphs \( G, H \) we imply that there exists a graph \( S \) such that \( G, H \subseteq S \), so union and intersection are well defined.
occurrence of \( p_0 \), i.e., two different patterns are restricted to overlap at \( p_0 \), see the example in Fig. 6.1 Thus both \( o_1 \) and \( o_2 \) contain an occurrence of \( p_0 \) and if \( p_1, p_2 \) are independent, the probability that both select the same occurrence is \( 1/[p_0] \).

Figure 6.1: Overlapping \( p_1, p_2 \) at \( p_0 \)

As a consequence, for independent \( p_1, p_2 \) we have the following equalities by substituting \([p_1] \times [p_2]/[p_0] \) for \([p] \).

1. \([p]/[p_1] = [p_2]/[p_0] \)
2. \([p]/[p_2] = [p_1]/[p_0] \)

This is useful if, e.g. in the first case, \( p_2, p_0 \) are much smaller than \( p, p_1 \), allowing us to express the same relationship more succinctly. The best possible reduction is obtained if, given \( p_1 \subseteq p \), we define \( p_0 \subseteq p_2 \) as the smallest subgraphs \( p_0 \subseteq p_1 \) and \( p_2 \subseteq p \) such that \( p = p_1 \cup p_2 \) and let \( p_1 \cap p_2 = p_0 \). In categorical terms this construction is known as an initial pushout, so we write \( \text{IPO}(p_1 \subseteq p) = p_0 \subseteq p_2 \).

We approximate complex patterns by decomposition into simpler ones. As a special case, disjoint patterns \( p \uplus q \) are approximated by \([p] * [q] \) because there is exactly one occurrence of their empty intersection in every graph. Since union and disjoint union are associative and distributive, we can chose how to decompose a complex patterns into an expression over smaller subpatterns.
6.2.2 Approximating the Effect of Basic Rules

Given a rule \( r = (L \leftrightarrow K \leftrightarrow R) \) and a pattern \( p \), a gluing is a pair of injective morphisms \( L \rightarrow G \leftarrow p \) that are jointly surjective. We write \( L \cup p \) and call such gluing critical if \( r \) is applicable at \( L \rightarrow G \) such that elements in the image of \( p \) are deleted in the application, that is \( L \cup p \) is critical if \( L \cap p \not\subset R \). Symmetrically \( R \cup p \) is critical if the inverse \( r^{-1} \) of \( r \) is applicable, i.e., the match into \( R \cup p \) satisfies the dangling conditions, so that elements in \( p \) are deleted, i.e., \( R \cap p \not\subset L \).

The effect of applying rule \( r \) on the occurrences of pattern \( p \) is approximated by

\[
\Delta(r,p) = - \sum_{L \cup p \; \text{critical}} \frac{[L \cup p]}{[L]} + \sum_{R \cup p \; \text{critical}} \frac{[r^{-1}(R \cup p)]}{[L]}
\]

The first component of the sum handles the destruction of pattern occurrences. Every critical gluing of a pattern with a rule’s left-hand side is evidence of the potential to destroy an instance of that pattern. How many of these instances are on average destroyed per application depends on the average number of occurrences of the gluing \( L \cup p \) per match for the rule. Fig. 6.2 illustrates an example of a critical gluing rule of convertTo1 with pattern [0], where the rule application destroys one occurrence of the pattern. The left-hand side of the rule is [01] and the pattern \( p \) is [0], their gluing critical \( L \cup p \) is [01] as they overlap at [0]. That results \([01]/[0]\) for the component \([L \cup p]/[L]\).

The second component handles creation of pattern occurrences. Here the critical gluing \( R \cup p \) is between the right-hand side of the rule and the pattern, which is transformed to a gluing over the left-hand side \( r^{-1}(R \cup p) \) using the reverse rule \( r^{-1} \) of \( r \).

The above equation holds without assuming that \( L, p \) or \( R, p \) are independent. However, it creates new, larger patterns \( L \cup p \) and \( R \cup p \). In order to avoid generating new patterns of increasing size, which could lead to an infinite number of variables in ODEs, more succinct versions can be adopted under the
Figure 6.2: The critical gluing between rule \textit{convertTo1} and pattern $[0]$

assumption of independence.

If $L, p$ are independent $[\frac{L \cup p}{L}]$ can be rewritten as $[\frac{p}{L \cup p}]$ using Equation 1 in previous section. For the second component we first have to derive a predecessor of $p$ under the application of $r$. This is achieved by an initial pushout $\text{IPO}(L \subseteq r^{-1}(R \cup p)) = i \subseteq q$ which will produce the smallest pattern $q$ such that applying $r$ to $L \cup q$ yields $R \cup p$. This gives us

$$
\overline{\Delta}(r, p) = - \sum_{L \cup \cup \text{ critical}} \frac{[p]}{[L \cap p]} + \sum_{R \cup \cup \text{ critical}} \frac{[q]}{[L \cap q]}
$$

with $q \subseteq r^{-1}(R \cup p)$ the smallest graph such that $L \cup q \Rightarrow R \cup p$. Fig. 6.3 shows the critical gluing between the right-hand side of the rule \textit{convertTo0} and pattern $[0]$. The rule application creates one occurrence of the pattern without any destroy. The rule’s left-hand side is $[01]$, pattern $p$ is $[0]$, and $q$ (the predecessor of $p$) is $[1]$. Thus, the result for this component $[\frac{q}{L \cup p}]$ is $[\frac{1}{1}]$.

### 6.2.3 Approximating Conditional Patterns and Rules

For conditional patterns and rules we have to restrict the set of occurrences to those satisfying the application conditions. For a pattern $p$ with application condition $C$ the occurrences of the conditional pattern $(L, C)$ are approximated
Figure 6.3: The gluing critical between rule convertTo0 and pattern [0]

by \([L, C] = [L] \times P(C)\), i.e., the number of occurrences of \(r\) multiplied by the probability that condition \(C\) is satisfied. For \(C = \{c_1, \ldots, c_k\}\), assuming conditions are independent, \(P(C) = P(c_1) \times \cdots \times P(c_k)\) with the probabilities of individual conditions defined by

- \(P(\neg c) = 1 - P(c)\)

- \(P(c : L \subseteq N) = [N]/[L]\), which can be simplified to \(P(c : I \subseteq M) = [M]/[I]\) if IPO\((d : L \subseteq N) = I \subseteq M\) such that \(M\) and \(L\) are independent.

The first equality follows from the axioms of probability. The second one is true if extensions of occurrences of \(L\) to \(N\) are unique, i.e., for every injective \(m : L \rightarrow G\), if there are injective \(m_1, m_2 : N \rightarrow G\) with \(m_1|_L = m_2|_L\) then \(m_1 = m_2\). This is the case, for example, if application conditions only check for (absence of) edges between existing nodes, or outgoing edges from existing nodes where it is known that at most one such edge may exist.
Chapter 7

Algorithm and Implementation

As a general approach to derive ODEs, an algorithm has been developed and implemented, which shows step by step how to obtain the ordinary differential equations from a rule-based graph transformation system.

The algorithm is presented and explained in Section 7.1 and its implementation is covered in Section 7.2.

7.1 Algorithm

The algorithm is divided into two parts, the first part is to find the effects of rules on pattern occurrences, and the second describes the derivation of the equations. We provide a description for the definitions and notations used in the algorithm and a brief explanation of the overall solution.

- Patterns are considered as rules without action, i.e., where the left and right-hand sides are equal.

- $\text{Con}(p_i, r_j)$ is the set of critical pairs (conflicts) between patterns $p_i$ and rules $r_j$, i.e., they represent minimal situations where both pattern $p_i$ and a rule $r_j$ have an occurrence in a given graph, such that the application of $r_j$ destroys the occurrence of $p_i$. 
• \( \text{Dep}(p_i, r_j) \) is set of minimal dependencies between patterns \( p_i \) and rules \( r_j \) where an application of rule \( r_j \) creates an occurrence of pattern \( p_i \).

• \( \Delta \) is a two-dimensional array, indexed by patterns \( p_i \) and rules \( r_j \). Its entries are pairs \( (CS, DS : \text{List}(\text{Frac}(\text{num, den} : \text{nat})) \) of lists of fractions of positive integers. \( CS \) and \( DS \) represent the sums obtained from conflicts and dependencies respectively. Both numerator and denominator are pattern indices. E.g., \( \Delta[i][j].CS \) represents the sum resulting from the conflicts between \( p_i \) and \( r_j \).

• Lists are extended using the operator \( : \), e.g. \( DS \) is extended by \( DS : \langle \text{num/den} \rangle \) adding a new fraction \( \langle \text{num/den} \rangle \) at the end. Empty lists are denoted by \( \epsilon \). For a list \( X \), by \( X[a] \) we denote its \( a \)th element.

• Conflicts \( \text{con} \) have a field \( \text{con.I} \) for the intersection of the pattern and the left-hand side and \( \text{con.U} \) for their union.

• Dependencies \( \text{dep} \) have fields, \( \text{dep.q, dep.I} \) for the derived pattern \( q \) and its intersection with the left-hand side as well as \( \text{dep.U} \) for their union. In a dependency between pattern \( p_i \) and rule \( r_j : L \rightarrow R \), \( q \) is the smallest graph such that \( L \cup q \xrightarrow{r_j} R \cup p_i. \)

We implemented the derivation of ODEs using the AGG tool’s Critical Pair (CP) and Minimal Dependency (MD) analysis to compute critical overlaps. The AGG assumptions for critical pair analysis were discussed in Section 1.3.

The algorithm starts with a given list of patterns \( X \) including all the rule’s left-hand sides and all patterns in \( P \). It invokes AGGs CP and MD analysis for all pairs of patterns \( p \in X \) and rules \( r \in R \). For each such critical pair we have two cases. If \( p \cup L \), the relevant gluing of \( r \)’s left-hand side with the pattern graph, is independent, an expression \( -[p]/[p \cap L] \) is added to the effect \( \overline{\Delta}(r, p) \), with \( p \cap L \) the intersection of \( p \) and \( L \) in \( p \cup L \). Otherwise, the effect is \( -[p \cup L]/[L] \). In either case, the list of patterns is extended by \( p \cap L \) or \( p \cup L \) only if they are new patterns. Analogously we add positive contributions \( [q]/[L \cap q] \)
or $[r^{-1}(R \cup p)]/[L]$ for every minimal dependency between $p$ and $r$ and again record in $X$ every new pattern arising. Once all pairs have been considered, we repeat the process, invoking AGGs CP and MD analysis again for all pairs of new patterns in $X$ and rules in $R$. The process terminates when no new patterns are produced.

By default we assume all patterns to be independent, always choosing the more compact alternatives above. If we do not want to avoid making this assumption for a particular pattern, we can add this to an embargo list which prevents its approximation at the cost of increasing the number of patterns generated, and thus the number of ODEs. All approximations that are used are recorded in a list $AP$ in order to be reused should the pattern come up again, and to be evaluated for independence once the generation is complete. In general, there is no guarantee that this process will terminates, but we can decide to curtail further refinement if we judge the ODEs to be sufficiently accurate.
Algorithm 1 Effects $\Delta$ of Rules on Patterns

Given

$GTS = (TG, R, \rho)$ where

- $TG$ type graph;
- $R = \{r_1, \ldots, r_m\}$ rules, with $L_R$ set of set left-hand sides
- $\rho : R \to \mathbb{R}$ rate assignment;

Set of patterns $P = \{p_1, \ldots, p_n\}$ without iso copies

Init

$X \leftarrow L_R : P$ // list of all left-hand sides and patterns

$i \leftarrow size(L_R) + 1$ // index in $X$ of next pattern to be checked

$I \leftarrow \epsilon$ // map of isomorphisms between patterns, by index in $X$

$PA$ // map of pattern approximations, fully defined on $L_R$

WHILE ($i < size(X)$)

IF ($I[i]$ undefined AND $PA[X[i]]$ undefined) THEN

FOR $j \rightarrow 1 \ldots m$

$\langle con_1, \ldots, con_k \rangle \leftarrow Con(p_i, r_j)$

FOR $c \rightarrow 1 \ldots k$

$X \leftarrow X : con_c.I$

$\Delta[i][j].CS \leftarrow \Delta[i][j].CS : \langle i/size(X) \rangle$

$PA(con_c.U) = [p_i] \times [L_j]/[size(X)]$

IF ($\exists 1 \leq a \leq size(X) - 1 : con_c.I \cong X[a]$)

THEN $I[size(X)] \leftarrow a$ END IF

END FOR

$\langle dep_1, \ldots, dep_l \rangle \leftarrow Dep(p_i, r_j)$

FOR $m \rightarrow 1 \ldots l$

$X \leftarrow X : dep_l.q : dep_t.I$

$\Delta[i][j].DS \leftarrow \Delta[i][j].DS : \langle size(X) - 1/size(X) \rangle$

$PA(dep_c.U) = [X[size(X) - 1]] \times [L_j]/[size(X)]$

IF ($\exists 1 \leq a \leq size(X) - 2 : dep_l.q \cong X[a]$)

THEN $I[size(X) - 1] \leftarrow a$ END IF

IF ($\exists 1 \leq a \leq size(X) - 1 : dep_t.I \cong X[a]$)

THEN $I[size(X)] \leftarrow a$ END IF

END FOR

END FOR

END IF

$i \leftarrow i + 1$

END WHILE
Algorithm 2 Print ODEs

FOR $i \rightarrow \text{size}(L_R) + 1 \ldots \text{size}(X)$
    FOR $j \rightarrow 1 \ldots m$
        IF $(\Delta[i][j].CS \neq \epsilon \text{ OR } \Delta[i][j].DS \neq \epsilon)$ THEN
            PRINT("d X" + $i$ + "/ dt = " + $PA(L_j) + " \times " + \rho(r_j) + " \times ")$
        END IF
    CS $\leftarrow \Delta[i][j].CS$
    IF (CS $\neq \epsilon$) THEN PRINT(""")
        WHILE (CS = CS': $\langle num/den \rangle$)
            PRINT(" - X" + num + "/ X" + den)
            CS $\leftarrow$ CS'
        END WHILE
    PRINT("") END IF
    DS $\leftarrow \Delta[i][j].DS$
    IF (DS $\neq \epsilon$) THEN PRINT(""")
        WHILE (DS = DS': $\langle num/den \rangle$)
            PRINT(" + X" + num + "/ X" + den)
            DS $\leftarrow$ DS'
        END WHILE
    PRINT("") END IF
END IF
END FOR
END FOR
We assume to be given a GTS including a type graph, set of rules and their rates, and a set of basic patterns. In this thesis, the AGG tool is used to model the graph transformation system.

The algorithm starts with a given list of patterns $X$ including all the rules’ left-hand sides and all patterns in $P$. We have introduced $i$ as index for list $X$. Its initialisation is $size(L_R) + 1$, pointing the index to the field following the left-hand sides of rules. $I$ is a list that will contain all the isomorphic mappings between patterns in list $X$ in order to avoid repeating the critical pair analysis process. $PA$ will contain the approximations for the gluing between the patterns and rules’ left-hand sides. Keeping track of approximations is important in order to avoid taking complex, approximated patterns for basic ones that need to be checked against the rules. Besides, the list can be used for validating pattern dependencies in the network.

$WHILE(i < size(X))$ creates a loop to go through all patterns in list $X$, and it continues by incrementing the index $i$ by 1 each circle until the index reaches the end, i.e., the last pattern in the list. The conditional statement $IF (I[i] \text{ undefined AND } PA(X[i]) \text{ undefined})$ ensures that there is no isomorphic copy for the pattern $X[i]$ in $I$, i.e., the pattern does not already exist in $X$ and also has not been approximated before.

For each pattern that satisfies the conditional statement, we run all the rules in order to find their $\Delta$ on the pattern. To do that we will perform the critical pair analysis in order to find the critical overlaps, i.e., when the application of a rule will change the occurrences of a pattern. If the application of a rule destroys a pattern occurrences this gives rise to a conflict recorded by $Con(p_i,r_j)$. It is a dependency when the application of a rule creates pattern occurrences as recorded in $Dep(p_i,r_j)$. We use the AGG tool’s Critical Pair (CP) and Minimal Dependency (MD) analysis to compute critical overlaps.

All the critical pairs for every rule that destroys the pattern occurrences are collected in a list and for every critical pair there is exactly one intersection $con_c.I$ which will be added to the list $X$. Now we know $con_c.I$, we can derive (–
\[ \sum_{L \cup p \text{ critical}} \frac{|p|}{|L \cap p|} \] by adding the \( \langle i/\text{size}(X) \rangle \) to the list \( \Delta[i][j].CS \). We deal with the indices instead of actual patterns to ease creating Octave/Matlab equations in the second part of the algorithm. Here the index \( i \) identifies the pattern \( X[i] \) and \( \text{size}(X) \) represents the index of the last added element of list \( X \) which is \( \text{con}_c.I \). This procedure will be repeated for all critical overlaps found. The next step is to add into the list \( PA \) the approximations, \( PA(\text{dep}_c.U) \), for the gluing of the pattern with each \( LHS \) of the rules in each critical overlaps. We use the pair approximation assumption \( [p] = [p_1] \times [p_2]/[p_0] \), where \( [p_1] \) is \( X[i] \), \( [p_2] \) is \( [L_j] \), \( [p] \) is their union, and \( [p_0] \) is their intersection. Following that, there is an isomorphism check \( \text{con}_c.I \cong X[a] \) to see whether the new pattern already exists in \( X \), in which case it is mapped as \( I[\text{size}(X)] \).

The same process will be repeated for calculating the dependencies by finding all minimal dependencies for each rule that produces patterns occurrences. For each critical pair we find the predecessor of pattern \( X[i] \) which is \( \text{dep}_q \) and its intersection with the left-hand side of the rule \( \text{dep}_I \) and calculate their \( \Delta \) that represents this part \( + \sum_{R \cup p \text{ critical}} \frac{|q|}{|L \cap q|} \) in the equations. The index of \( \text{dep}_q \) in the list \( X \) is \( X[\text{size}(X) - 1] \) since the last index is allocated for \( \text{dep}_I \) which is \( X[\text{size}(X)] \). The pair approximation here is for the gluing pair of the left-hand side of the rule and the pattern \( q \). In the dependency analysis, both \( q \) and \( q \cap L \) will be checked as to whether there are already isomorphic copies in \( X \).

Now that we have the pair approximations for the left-hand sides of rules, their rates, and their effects on patterns, no ingenuity is required to derive ordinary differential equations following our first part of the algorithm. In the second part of the algorithm we use a vector of approximations of left-hand sides of rules using the patterns in \( X \), a vector of rates, and a vector of effects \( \Delta \). We represent the basic patterns, which we create equations for, as \( X \) and identify them in the list \( X \) by their indices. For each pattern we run through all the rules such that their effects, \( \Delta[i][j].CS \) and \( \Delta[i][j].DS \), are not \( \epsilon \). We subtract the number of conflicts, i.e., pattern occurrences are being destroyed, while adding the dependencies.
7.2 **Implementation**

The motivation of implementing the algorithm was to validate our approach by applying it to additional and more complex examples. Complex and larger case studies need an automated process to produce the results. This section briefly discusses the implementation of the algorithm, components, design and the overall tool architecture.

7.2.1 **Architecture**

The overall architecture of the approach, see Fig. 7.1, combines the following components:

- **GT tool (AGG):** to model the graph transformation system.
- **ODE Generator.**
- **Simulation tool (VIATRA2-GraTS):** to validate pattern independence.
- **ODE Solver (Octave).**

![Figure 7.1: Overall tool architecture](image)

But in order to implement the algorithm, several tools and languages were involved to carry out the task. The details of each component are briefly discussed in the following sections.
7.2.1.1 AGG GUI

AGG\textsuperscript{1} [1] is a graph transformation tool that was needed during the implementation to model the systems and test them. Fig. 7.2 shows the general layout of the tool and features utilised in the project.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{agg_gui.png}
\caption{Screenshot of AGG graph transformation tool}
\end{figure}

\textsuperscript{1}http://www.user.tu-berlin.de/o.runge/agg/index.html
7.2.1.2 ODE Generator

AGG API:
The AGG Application Programming Interface (API) is used to integrate the
critical pair analysis methods into our tool as well as other functionalities such
as loading the AGG modelled system, applying rules, finding matches, and
reverse rules.

For example, the methods `conflictCPA.getCriticalPair()` and `dependency-
CPA.getCriticalPair()` are invoked from the AGG API to find the critical over-
laps and they are adapted into the `getConflict()` and `getDependency()` to com-
pute the $\Delta$, i.e., the rules’ effects on the patterns. The class diagram 7.3 illus-
trates more details about what methods are used in the implementation of the
tool and some fragments of the source code are listed below to show how they
are used. The complete source code can be found at https://goo.gl/3keiup.

```
Listing 7.1: Generate Equations

private void generatingEquations(){
    int iIndex=0;
    while (iIndex <this.A.size()){  
        AItem p = this.A.get(iIndex);
        if (!p.isPattern()){
            iIndex++;
            continue;
        }
        String str_P_Equation="\t" + p.getXDotVariable() + "=";
        boolean addPlus=false;
        for (int iRuleIndex=0; iRuleIndex<this.A.size(); iRuleIndex++){  
            AItem r = this.A.get(iRuleIndex);
            if (!r.isRule()){
                break;
            }
            String strDelta = this.getDelta(r, p);
            if (strDelta==null){
                continue;
            }
            if (addPlus){
                str_P_Equation += "..." + System.lineSeparator() + "\t\t\t" ;
            }
        }
    }
}
```
Listing 7.2: Calculate $\Delta$

```java
private String getDelta(AItem r, AItem p){

    String strDelta =
        this.getConflict((ARule)r, (APattern)p) +
        this.getDependency((ARule)r, (APattern)p);
    if (strDelta.length()==0){
        return null;
    }
    return strDelta;
}
```

The while loop in the first list is to iterate the critical pair analysis between patterns and rules and eventually produce the equations. In the second list, the `getConflict()` returns the number of patterns occurrences that are destroyed by each rule application, while `getDependency()` calculates the pattern occurrences that are generated.

The design and organisation of classes that carry out the functionalities are given in Fig. 7.3. It illustrates the high level structure of our tool and the results API using a UML class diagram.
Figure 7.3: Class diagram - API in Java
ARule and APattern are classes which fetch the rules and basic patterns respectively as inputs in the algorithm. AItem represents the list $X$ where we store the rules’ left-hand sides and the patterns. IsoPattern records the mappings for the iso-copies of the existing patterns and GeneratedPattern records the new generated patterns. Whilst class GenerateEquation describes the declaration of AGG object (gragra: GraGra) as well as the main methods that implement our algorithm and deriving the equations. The last class GUIApp produces the graph user interface of the tool.

Previously, it was explained that the critical pair analysis methods were used to find critical overlaps. However in order to find the $\Delta$ and derive the ODEs according to the algorithm we have constructed many structures such as $p \cap L$, $G$, $q$, and $q \cap L$ each time a critical overlap is found, depends on whether there is a critical conflict or a minimal dependency, and this process is iterated.

For example, a code fragment to construct graph $q$ is presented in the below list. This will be illustrated in more detail by examples in the next two chapters.

### Listing 7.3: Construct q

```java
private Graph update_g_toGet_q(final Graph g){
    ArrayList<GraphObject> qList = Collections.list(g.getElements());
...

    for (GraphObject gObject: qList){
        if (gObject.getObjectName().equals("Critical_Original")||
            gObject.getObjectName().equals("Critical_Created")||
            gObject.getObjectName().equals("PatternOnly")){
            continue;
        }

        if (this.canGraphObjectBeDestroiedSafely(gObject)
            && g.isElement(gObject)){
            try {
                g.destroyObject(gObject);
            }
        }
    }
}
```

105
```java
} catch (TypeException e) {
    e.printStackTrace();
}
}
return g;
}

A part of the code to obtain the intersection between the graph $q$ and $L$ is shown below

**Listing 7.4: Construct $q \cap L$**

```java
public Graph update_q_toGetIntersectionWith_L(final Graph q){
    ArrayList<GraphObject> qList = Collections.list(q.getElements());
    for (GraphObject gObject: qList){
        if (gObject.isCritical()){
            continue;
        }
        else if (q.isElement(gObject)){
            try {
                q.destroyObject(gObject);
            } catch (TypeException e) {
                e.printStackTrace();
            }
        }
    }
    return q;
}
```

The complete source code are available at https://goo.gl/3keiup.

**Java and Eclipse:**
The high-level programming language Java [6] was used to implement the tool with the 1.8 (JDK 8u112) version. The apparent reason of using Java was due to the AGG API which is developed with the same programming language.

Eclipse Kepler [3] is used as an integrated development environment for Java, as well as for the simulation.
7.2.1.3 VIATRA2 - GraTS

VIATRA2 is a model transformation framework that has been implemented as an Eclipse plug-in. Stochastic simulation of graph transformation systems is enabled via GraTS extending the VIATRA2 plugin [47].

We use VIATRA and GraTS for the simulation as a validator for the assumption of patterns independence. These details are discussed in Chapters 8 and 9. The basic layout is shown in Fig. 7.4.

Figure 7.4: Pattern independence validator - VIATRA2/GraTS
7.2.1.4 Octave

Octave is a scientific and high-level programming language, primarily intended for numerical computations. It has a powerful mathematics-oriented syntax with built-in plotting and visualisation tools. We use Octave to solve the ordinary differential equations, as well as plotting the evolution of the system and obtaining the number of occurrences of the patterns. Refer to Chapters 8 and 9 for further details on how the task is performed. The Fig. 7.5 shows the layout of the tool.

Figure 7.5: ODE solver - Octave
7.2.2 User Interface

To make our approach user friendly, we have also implemented a GUI for the tool. The Fig. 7.6 illustrates the layout of the tool and all features utilised in the project. Comments are given on the layout to explain each section of the interface.

![Graphical User Interface of the ODE generator](image)

Figure 7.6: The graphical user interface of the ODE generator

7.2.3 Validation

In order to do validate the implementation, we manually checked the hand generated ordinary differential equations for the systems against the ones were automatically generated from the tool. The results were equivalent.
Chapter 8

ODEs from the Voter Model

In this chapter, we apply and test our approach on the Voter model case study Chapter 3. The aim is to answer the questions asked in the model checking and simulation chapters 4 and 5 respectively. The comparison between the result analysis of the approaches are discussed in Evaluation chapter, Section 10.1.

This chapter is organised as follows. Section 8.1 presents the derivation of the ODEs from the rule-based GTS system. Validating the assumptions about the patterns independences will follow in section 8.2. The chapter concludes with the results of the approach in Section 8.3.

8.1 Deriving the ODEs

For the Voter model we are interested in occurrences of patterns [0], [1], [00], [01] and [11] covering the basic states of individual nodes and connections. We present the result of the derivation of ODEs according to the algorithm for all the patterns listed above and discuss in more detail the pattern [0] and pattern [01].

We work on the GTS that was modelled in chapter 3, using the AGG tool, which includes the type graph, cconvertTo0, convertTo1, splitTo0, splitTo1 rules, and the patterns mentioned above. The AGG file is imported into the ODE generator, see Fig. 8.1, then the rates of the rules are provided in order
to perform the critical pair analysis and derive the equations automatically, see Fig. 8.2.

![Image: Importing GTS/AGG file into the ODE generator]

Figure 8.1: Import the GTS/AGG file into the ODE generator

The list $X$ initially contains all the left-hand sides of the rules (without repeating isomorphic copies) and the basic patterns which we create equations for.

List $X$:

$X_1 = L_1 : [01]$

$X_2 = L_2 : [01_0]$

$X_3 = L_3 : [01_1]$

$X_4 = p_1 : [0]$

$X_5 = p_2 : [1]$

$X_6 = p_3 : [01]$

$X_7 = p_4 : [00]$

$X_8 = p_5 : [11]$
$X_1$ is the left-hand side of the converting rules, $X_2$ is $splitTo0$’s LHS, and $X_3$ is $splitTo1$’s. Now we perform the algorithm on each pattern and show the results.

### 8.1.1 Pattern [0]

Pattern [0] is listed in $X$ as $X_4$ with no other copies. Thus, the critical pair analysis (CPA) can be performed. We only need to consider conversion rules $convertTo0$, $convertTo1$. Switching rules $splitTo0$, $splitTo1$ have no impact on [0] because they do not affect node states, in other words, there is no critical overlap between the rules and the pattern.

The critical overlap between the first rule $convertTo0$ and the pattern as well as the rule’s effect on the pattern are illustrated in Fig. 8.3. The application of the rule does not destroy any occurrences of the pattern since the critical overlap only occurs between the right-hand side of the rule and the pattern.
That means the pattern is produced each time the rule applies. So, \( q \), the predecessor of \( p = [0] \) created, is pattern [1] and it will be added to \( X \) as \( X_9 \). Since an isomorphic copy already exists in the list we record their mappings. As a consequent, \( X \) will extend by adding \( X_9 \) which is mapped by \( I(9) = 5 \). In addition, the \( L \cap q \) is also added to the list as \( X_{10} \) and its mapping is recorded as \( I(10) = 5 \). As a result the effect of \( convertTo0 \) rule on pattern[0] is

\[
\Delta(4, 1).CS = \emptyset
\]

\[
\Delta(4, 1).DS = X_9/X_{10}
\]

4 is the index of pattern [0] in \( X \), \( i \leftarrow \text{size}(L_R) + 1 \), and 1 is the index, \( j \), of the rule in \( R \). Both \( X_9 \) and \( X_{10} \) are isomorphic with exactly one occurrence produced each time the rule applies. That means, the effect of the rules can be tracked exactly, so that no approximation is required.

We approximate the gluing of left-hand side of the rule and \( q \), \( L \cup q \), as \( PA(dep_1.U) = (X_9 \ast X_1)/X_{10} \). Since \( X_9 \) and \( X_{10} \) are isomorphic the result becomes \( PA(dep_1.U) = X_1 \).

With regards to the \( convertTo1 \) rule, see Fig. 8.4, the overlap which is critical only happens between the left-hand side of the rule and the pattern.
This means the rule application only destroys the pattern occurrences without producing any. $L \cap p = [0]$, added to $X$ as $X_{11}$, and its map is $I(11) = 4$ since it is isomorphic to the pattern $[0]$. The rule’s effect is

$$\Delta(4, 2).CS = -X_4/X_{11}$$

Since $X_4$ and $X_{11}$ are isomorphic, the rule destroys exactly one occurrence per application, hence the effect factor is $-1$, i.e., no approximation is required.

The approximation for the gluing of the left-hand side of the rule and pattern $[0]$, $L \cup p$, is $PA(\text{con}_1.U) = (X_4 \ast X_1)/X_{11}$. Since $X_4$ and $X_{11}$ are isomorphic the result becomes $PA(\text{con}_1.U) = X_1$.

To simplify the explanation, we describe the second part of the algorithm on the pattern $[0]$ in order to construct an ODE.

One of the vectors of the general equation holds the number of matches for the rules. We use patterns in $X$ to approximate the occurrences of the left-hand sides of the rules. The number of matches for the $\text{convertTo}0$ rule, is the number $[01]$ of 01 edges. This also applies to the $\text{convertTo}1$ rule, and indeed to all occurrences of these two rules in any ODE derived here. That means, the number of matches can be tracked exactly, so that no approximation is
required. \( \rho(convertTo0) \) and \( \rho(convertTo1) \) are the rates of the rules. The following ODE for pattern \([0]\) is produced according to the algorithm.

\[
\frac{d[0]}{dt} = [01] \times \rho(convertTo0) \times (X_9/X_{10})
+ [01] \times \rho(convertTo1) \times (-X_4/X_{11})
\]

due to the isomorphisms, it can be presented as follows

\[
\frac{d[0]}{dt} = [01] \times \rho(convertTo0) \times (1)
+ [01] \times \rho(convertTo1) \times (-1)
\]

The result of the updated lists of \(X, I, PA\) are:

List \(X\) :
\[
X_1 = L_1 : [01] \\
X_2 = L_2 : [01,0] \\
X_3 = L_3 : [01,1] \\
X_4 = p_1 : [0] \\
X_5 = p_2 : [1] \\
X_6 = p_3 : [01] \\
X_7 = p_4 : [00] \\
X_8 = p_5 : [11] \\
X_9 = dep.l.q : [1] \cong X_5 \\
X_{10} = dep.c.I : [1] \cong X_5 \\
X_{11} = conc.I : [0] \cong X_4
\]

List \(I\) :
\[
I(9) = 5 \\
I(10) = 5 \\
I(11) = 4
\]

List \(PA\) :
\[ PA_1 : PA(dep_{1,U}) = X_1 \]
\[ PA_2 : PA(con_{1,U}) = X_1 \]
\[ PA_3 : PA(L_{convertTo0}) = PA(L_1) = X_1 \]
\[ PA_4 : PA(L_{convertTo1}) = PA(L_1) = X_1 \]

8.1.2 Pattern [01]

The evolution of this pattern has contributions from all four rules and its index in \( X \) is 6. There are critical overlaps between the convertTo0 rule and the pattern, Fig 8.5 shows how it effects the pattern’s occurrences.

![Figure 8.5: The effect of rule convertTo0 on pattern 01](image)

The rule destroys and creates the pattern’s occurrences at the same time. Performing the conflict analysis function, we obtain an overlap between the \( L_1 \) and \( p_3 = [01] \) and it is critical because the rule is applicable and the element of the pattern, \([1]\), will be deleted, i.e., \( L_1 \cap p_3 \) is not a subset of the right-hand side, see the left rule transformation in Fig 8.5. The \( L_1 \cap p_3 \) is added to \( X \) and \( I(12) = 5 \) is recorded since it has already a copy in the list. The conflict result is

\[ \Delta(6,1).CS = -X_6/X_{12} - X_1/X_{12} \]

116
where $-X_1/X_{12}$ is another overlap of the match as whole, $L_1$, with the pattern and it destroys one occurrence. The pair approximation for the gluing pair is calculated as

$$(X_1 \times X_6)/X_{12}$$

Performing the dependency analysis, we obtain an overlap between the $p_4 = [00]$ and $p_3$ and it is critical because the inverse of the rule is applicable and the element of the pattern, $[0]$, will be deleted, i.e., $p_4 \cap p_3$ is not a subset of the left-hand side, see the right rule transformation in Fig 8.5. Patterns $q = X_{13}$ and $q \cap L = X_{14}$ are added to $X$ and as well as their mappings $I(13) = 5$ and $I(14) = 5$. The dependency result is

$$\Delta(6, 1).DS = X_{13}/X_{14}$$

Rule convertTo1 is handled in the same way of rule convertTo0. It has both conflict and dependency, see Fig 8.6. The result of the conflict is

$$\Delta(6, 2).CS = -X_6/X_{15} - X_1/X_{15}$$

and for the dependency is

$$\Delta(6, 2).DS = X_{16}/X_{17}$$

With regards to the splitting rules, splitTo0, splitTo1, they only destroy the pattern’s occurrences. Figures 8.7 and 8.8 show the overlap between their left-hand sides and pattern [01]. Since their intersection is isomorphic to pattern [01], we use the index of the pattern instead of the new added patterns in the equation to ease the process. Hence the conflict sum for splitTo0 will be

$$\Delta.CS(6, 3) = -X_6/X_6$$
and for $\text{splitTo}_1$

$$\Delta(6, 4).CS = -\frac{X_6}{X_6}$$

Figure 8.6: The effect of rule $\text{convertTo}_1$ on pattern 01

Figure 8.7: The effect of rule $\text{splitTo}_0$ on pattern 01

To construct an ODE for pattern [01], we use patterns in $X$ to approximate the left-hand sides of the rules especially the splitting rules. The number of matches for $\text{convertTo}_0, \text{convertTo}_1$ is [01] but we approximated the effects of these two rules. In turn, for $\text{splitTo}_0, \text{splitTo}_1$ the numbers of matches have to be approximated, but their effects are fixed at $-1$ because they replace exactly
one 01 edge with one labelled 00 or 11, respectively.

The basic left-hand side patterns of the splitting rules $\text{splitTo}_1, \text{splitTo}_0$, consisting of a 01 edge and either a 1 or a 0 node can be mapped accurately as $[01,1] = [01][1]$ and $[01,0] = [01][0]$. However, there are negative application conditions preventing the prior existence of edges the rules are about to create, e.g., between the two 1 nodes in $\text{splitTo}_1$’s and the two 0 nodes in $\text{splitTo}_0$’s left-hand side. Using the mapping for conditional patterns and rules, the matches for $\text{splitTo}_0$ are approximated by $[L_r] = [01][0] \times P(c) = [01][0] \times (1 - P(c)) = [01][0] \times (1 - [00]/[0_0])$ where $[00]/[0_0]$ is the quotient $[M]/[I]$, and $[0_0] = [0][0]$ is the graph with two disconnected 0 nodes, i.e., the set of order pairs of 0 nodes in the graph. The approximation of the number of matches for $\text{splitTo}_1$ is derived analogously, $[01][1] \times P(c) = [01][1] \times (1 - [11]/([1][1]))$.

Hence the ODE will be formed as
\[
\frac{dX_6}{dt} = X_1 \times \rho(\text{convertTo0}) \times \left( \frac{X_{13}}{X_{14}} - \frac{X_6}{X_{12}} - \frac{X_1}{X_{12}} \right) \\
+ X_1 \times \rho(\text{convertTo1}) \times \left( \frac{X_{16}}{X_{17}} - \frac{X_6}{X_{15}} - \frac{X_6}{X_{15}} \right) \\
+ (X_1 \times X_4 \times (1 - \frac{X_7}{X_4(X_4 - 1)})) \times \rho(\text{splitTo0}) \times \left( -\frac{X_6}{X_6} \right) \\
+ (X_1 \times X_5 \times (1 - \frac{X_8}{X_5(X_5 - 1)})) \times \rho(\text{splitTo1}) \times \left( -\frac{X_6}{X_6} \right)
\]

8.1.3 Pattern [00]

\[
\frac{dX_7}{dt} = X_1 \times \rho(\text{convertTo0}) \times \left( \frac{X_6}{X_5} + \frac{X_6}{X_5} \right) \\
+ X_1 \times \rho(\text{convertTo1}) \times \left( -\frac{X_7}{X_4} - \frac{X_7}{X_4} \right) \\
+ (X_1 \times X_4 \times (1 - \frac{X_7}{X_4(X_4 - 1)})) \times \rho(\text{splitTo0}) \times \left( \frac{X_6}{X_6} + \frac{X_6}{X_6} \right)
\]

8.1.4 Pattern [11]

\[
\frac{dX_8}{dt} = X_1 \times \rho(\text{convertTo0}) \times \left( -\frac{X_8}{X_5} - \frac{X_8}{X_5} \right) \\
+ X_1 \times \rho(\text{convertTo1}) \times \left( \frac{X_4}{X_4} + \frac{X_4}{X_4} + \frac{X_6}{X_4} + \frac{X_6}{X_4} \right) \\
+ (X_1 \times X_5 \times (1 - \frac{X_8}{X_5(X_5 - 1)})) \times \rho(\text{splitTo1}) \times \left( \frac{X_6}{X_6} + \frac{X_6}{X_6} \right)
\]

8.1.5 Pattern [1]

\[
\frac{dX_5}{dt} = X_1 \times \rho(\text{convertTo0}) \times (-X_5/X_5) \\
+ X_1 \times \rho(\text{convertTo1}) \times (X_4/X_4)
\]

In the execution of the algorithm using the implemented tool, 12 critical pairs and 16 minimal dependencies were generated, leading to 44 patterns derived all of which were isomorphic to patterns already in the list. The details are provided in https://goo.gl/3keiup. The mapping of variables \(X\) to the patterns
and rules is displayed in list X in the GUI tool while generating the equations, see Figure 7.6.

### 8.2 Pattern Independence

In this section, we address the question whether the assumptions about independence of patterns are satisfied in the case study.

Simulation is used to validate the assumptions. We use a graph created randomly, consisting of 100 nodes equally shared between patterns 0 and 1, linked by 166 00 edges, 182 11 edges, and 161 01 edges. Split and conversion rules have the same rate of 1. The simulation has a batch size of 5 runs sharing the same simulation parameters.

The independence of patterns can be validated in a given graph by counting the occurrences of patterns \( p, p_1, p_2, p_0 \) and comparing \([p]\) to \([p_1][p_2]/[p_0]\). In our case study this means to check if, e.g., \([001] = [00][01]/[0]\) and analogously for approximations of \([011], [111], [000], [010], [101]\). After evaluating these approximations in given graph, we have to assess if they remain accurate over the course of the simulation, i.e., if they are preserved by the dynamics of the model. Therefore we validate the six approximations again at the end of the simulation. Since our given graph is generated randomly, we expect all patterns to be independent initially, but the application of rules can create higher-level structures where certain combinations of patterns occur more frequently than others.

Table 8.1 shows the results for the initial state. Table 8.2 is the statistics for the final states of the 5 runs, all starting in the same given graph, in terms of the mean and 95% confidence intervals for all patterns and their approximations.

In all cases the confidence intervals overlap or are close. This suggests that patterns are indeed independent and that this independence is preserved.
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Exact</th>
<th>Approx.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[001]v.[00][01]/0</td>
<td>537</td>
<td>534.52</td>
</tr>
<tr>
<td>[011]v.[01][11]/[1]</td>
<td>585</td>
<td>586.04</td>
</tr>
<tr>
<td>[111]v.[11][11]/[1]</td>
<td>592</td>
<td>662.48</td>
</tr>
<tr>
<td>[000]v.[00][00]/0</td>
<td>534</td>
<td>551.12</td>
</tr>
<tr>
<td>[010]v.[01][01]/[1]</td>
<td>480</td>
<td>518.42</td>
</tr>
<tr>
<td>[101]v.[01][01]/[0]</td>
<td>496</td>
<td>518.42</td>
</tr>
</tbody>
</table>

Table 8.1: Pattern occurrences and their approximation for initial graph

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Mean</th>
<th>Lower</th>
<th>Upper</th>
</tr>
</thead>
<tbody>
<tr>
<td>[001]</td>
<td>4.2</td>
<td>3.8</td>
<td>4.6</td>
</tr>
<tr>
<td>[00][01]/[0]</td>
<td>6.1</td>
<td>5.9</td>
<td>6.3</td>
</tr>
<tr>
<td>[011]</td>
<td>5.4</td>
<td>2.8</td>
<td>7.9</td>
</tr>
<tr>
<td>[01][11]/[1]</td>
<td>7.1</td>
<td>6.9</td>
<td>7.2</td>
</tr>
<tr>
<td>[111]</td>
<td>2692.4</td>
<td>2468.5</td>
<td>2916.2</td>
</tr>
<tr>
<td>[11][11]/[1]</td>
<td>2790.9</td>
<td>2538.6</td>
<td>3043.2</td>
</tr>
<tr>
<td>[000]</td>
<td>1631.2</td>
<td>1358.6</td>
<td>1903.7</td>
</tr>
<tr>
<td>[00][00]/[0]</td>
<td>1697.6</td>
<td>1456.3</td>
<td>1938.9</td>
</tr>
<tr>
<td>[010]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[01][01]/[1]</td>
<td>0.018</td>
<td>0.016</td>
<td>0.019</td>
</tr>
<tr>
<td>[101]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[01][01]/[0]</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Table 8.2: Pattern occurrences and their approximation for final graphs
8.3 Octave and Results

This section presents the analysis results we obtain from solving the differential equations using the ODE solver Octave [23].

For the ODE we are interested in the same questions asked of the simulations experiment 2: How does the evolution affect the system over time? Also, how do the patterns’ numbers, or the edges connecting them change over time. Moreover, when are the Weighted average Frequency of Patterns in the Steady state? The results obtained from the equations provide the prerequisite of such analyses.

The initial model for ODEs is provided with the same input as the start graph of the simulation experiment 2. It consists of 48 nodes, 8 nodes holding vote=0, 40 nodes of vote=1, and 60 edges between them. Rates for each rule are convertTo0=10, convertTo1=10, splitTo0=1, splitTo1=1.

The source code for encoding the ODEs in Octave is given in the listing below:

Listing 8.1: Octave Code

```octave
function xdot = voter(X, t)
    Conv0 = 1; #rule convertTo0
    Conv1 = 1; #rule convertTo1
    Split1 = 1; #rule splitTo1
    Split0 = 1; #rule splitTo0

    # [Pattern1]   -> X(5)
    # [Pattern0]   -> X(6)
    # [Pattern00]  -> X(7)
    # [Pattern11]  -> X(8)
    # [Pattern01]  -> X(9)

    xdot(1) = 0; xdot(2) = 0; xdot(3) = 0; xdot(4) = 0;

    #Pattern1   -> X(5)
    xdot(5) = X(9) * Conv0 * ( - X(5)/X(5)) ...
        + X(9) * Conv1 * ( + X(6)/X(6));
```

Operating lsode solves the system of ODEs. The function voter passed as an argument operates on \( X \), the vector of variables representing the number of pattern occurrences, i.e., pattern [1] \( \rightarrow X(5) \), pattern [0] \( \rightarrow X(6) \), pattern [00] \( \rightarrow x(7) \), pattern [11] \( \rightarrow x(8) \), pattern [01] \( \rightarrow x(9) \). Scalar \( t \) represents the time passed, with range and resolution defined by linspace(0,0.1,100) by creating a vector of 100 evenly spaced points in the time interval [0,0.1] [36].

The result in Fig. 8.9 illustrates the evolution of the system over time and Table 8.3 shows the WFPS. The evaluation and analysis of the comparison
between the result of ODE and Simulation will be taken place in Chapter 10, Section 10.1.2.

Figure 8.9: The Result of the ODEs

<table>
<thead>
<tr>
<th>Pattern</th>
<th>WFPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>8</td>
</tr>
<tr>
<td>[1]</td>
<td>40</td>
</tr>
<tr>
<td>[00]</td>
<td>15</td>
</tr>
<tr>
<td>[01]</td>
<td>0.16</td>
</tr>
</tbody>
</table>

Table 8.3: The Steady State in the System
Chapter 9

ODEs from the Peer-to-Peer Content Policing Model

The Voter model is simple, and does not feature complex left-hand sides and patterns. That means, it does not say much about the scalability of the approach. Therefore, in this chapter, we consider a more complex model describing a policy of content policing as employed by some social networks.

This chapter is organised as follows. Section 9.1 presents the case study including the type graph, rules and patterns. Deriving the ordinary differential equations from the graph transformation system is discussed in Section 9.2. Validating the assumptions about the patterns’ independence will follow in Section 9.3. The chapter concludes with the analysis and discussion of the comparison between the results of the ODE and simulation model in Section 9.4.

9.1 P2P Content Policing Graph Transformation Model

Extreme or illegal content in online social networks poses risks to individuals and society. Policing such content is often used as an argument for centralised surveillance. However, such a centralised approach to policing suffers from two
drawbacks.

1. The amount of data to be collected if content is to be policed comprehensively is prohibitive.

2. If centralised content policing is effective, it risks sacrificing privacy.

An alternative, distributed policing models perform their function without centralised management. In such a model, peers can blame inappropriate content they encounter and, beyond a certain threshold, aggregated blame from a number of peers leads to content being blocked or reported for evaluation. Such a protocol is potentially scalable, less intrusive than centralised surveillance, but it carries its own risks.

1. It is potentially sensitive to abuse of blame to suppress opposing views, harming freedom of expression.

2. It may not be effective enough in filtering out genuinely extreme content reliably.

3. Users may not be trusted to "blame correctly".

The sensitivity to abuse and reliability of a distributed policing protocol depends on a number of factors, including the amount of blame available in the network, the thresholds for blocking or reporting content and the behaviour of the majority of individuals in the network. We are interested in exploring under which circumstances and parameters a protocol like this can deliver satisfactory results, to understand its potentials and risks.

We approach this question by modelling the networks operation as a stochastic graph transformation system describing the interaction of users and content in the network and the adaptation connections between users based on the content they like. Adding rules for blaming content with varying rates we can explore how different behaviours influence the overall outcomes.
The type graph of the model is shown in Fig. 9.1, where Users can be friends and can like or blame Content which has a Boolean attribute `appr` to indicate if it is appropriate. To limit the amount of blame available, each user can only appoint blame once (e.g., during a certain interval represented by one run of the model). Constraints are also defined for the purpose of implementing the model in AGG. They are used to prevent loop connections and like or blame a content at the same time by a user, see figures 9.2 and 9.3.

The rules are given in the integrated notation for rules of Henshin [4] instead of AGG for the better presentation, see Fig. 9.5, as $L \cup R$ using colours and labels to indicate which elements are created, preserved and deleted. E.g., in the case of rule `like_content`, a User creates a `like` edge to a Content that is
liked by one of their friends if the User does not have a like or blame edge for that Content yet. Rule blame_bad_content correctly blames inappropriate content while blame_good_content incorrectly blames appropriate content. Rules switch_like and switch_friend play similar roles as convertTo0, convertTo1 and splitTo0, splitTo1 in the voter model, respectively, i.e., they are used for adopting a friend’s opinion or switching the connection.

Fig. 9.4 shows the patterns we want to observe in order to evaluate the operation of the model. They are also given as Henshin rules, but without creation or deletion of elements. Patterns TP, FP and TN, FN represent True and False Positives and True and False Negatives, respectively. We deem a Content to be flagged as inappropriate if it receives blame from a least two different Users. Then, if appr=false, i.e., it has been flagged correctly, this represents a true positive, otherwise a false positive. Content without two blames is not flagged, which gives rise to a true negative if appr=true, a false negative otherwise.

Figure 9.4: P2P content policing Patterns.
Figure 9.5: P2P content policing GT Rules.
9.2 Deriving the ODEs

In this section we discuss the ordinary differential equations generated from the P2P content policing graph transformation system using our implemented tool.

The system is modelled in AGG tool, see Fig. 9.6. We are interested in the evolution of patterns TP, FP, and TN, FN. Therefore there are only four equations to generate and we will discuss pattern TP in detail.

**Pattern TP**: True positive pattern is one of the basic patterns in the list $X$ with no other copies. Performing the critical pair analysis captures the critical overlap only with the rule `blame_bad_content` since the other rules have no effects
on the pattern’s evolution. Fig. 9.7 shows how the rule changes the occurrences of the pattern.

Figure 9.7: The effect of rule blame_bad_content on Pattern TP

The overlap occurs between the rule’s right-hand side and the pattern. Each time the application of the rule applies the patterns is produced without destroying any occurrences. The figure also illustrates how are the \( q \), the predecessor of the pattern, and its intersection with the rule’s left-hand side constructed. The newly produced patterns are added to the list \( X \) in order to perform the critical pair analysis on at the end and also use them to approximate the left-hand side of the rules while generating the equations as it was discussed in details in the previous two chapters. We follow the same procedure that was conducted to translate the Voter graph transformation system to ordinary differential equations. Due to the symmetry reason the overlap is calculated twice, that means two occurrences of pattern TP is created each time the rule applies. As a result
the effect of blame_bad_content rule on pattern TP is

$$\Delta = \frac{[u^b c_F]}{c_F} + \frac{[u^b c_F]}{c_F}$$

$[u^b c_F]$ is the $q$, and $[c_F]$ is the $q \cap L$. The ODE for the pattern is presented as follows

$$\frac{d[TP]}{dt} = ([u][c_F] - ([u^l c_F] + \frac{u^b}{[u][c_F]} \times \rho(b_{bl,c} \times ([u^b c_F] + \frac{u^b}{[u][c_F]}))$$

where $[u][c_F] - ([u^l c_F] + \frac{u^b}{[u][c_F]}$ is the approximated number of occurrences of the rule’s left-hand side using the patterns in $X$, and $\rho(b_{bl,c}$ is the rule’s rate (blame_bad_content). The simpler notation is used here to ease understanding the formula, while for the rest of the equations there will be mappings for each variable in the GUI of the tool, see Figure 9.8, for instance. The complete generated equations for all four patterns are presented in Section 9.4, including the approximations for the rules’ left-hand sides.

In the execution of the algorithm 21 critical pairs and 22 minimal dependencies were generated, leading to 65 patterns derived, where 53 of them were isomorphic to patterns already in the list. The complete source code of the generated patterns can be found at https://goo.gl/3keiup.

### 9.3 Pattern Independence

The independence of patterns is validated in the P2P case study, using simulation, by counting the occurrences of patterns $p, p1, p2, p0$ and comparing $[p]$ to $[p1][p2]/[p0]$. For example, the occurrences of the gluing $[q \cup L]$ should stay the same to their approximation $[q][L]/[q \cap L]$ at the initial and final states. The tables below show the results obtained from the simulation experiments.

Table 9.1 shows the results for the initial state. Table 9.2 shows the statistics
for the final states of the 5 runs, all starting in the same given graph, in terms of the mean and 95% confidence intervals for all patterns and their approximations. The experimental setup is detailed in the next section 9.4.

\( X_{12} \) is the generated \( q \) from the gluing pattern \( TP \) and \( rhs \) of rule \( \text{blame\_bad\_content} \), \( X_3 \) is \( lhs \) of the same rule, \((X_{12} \cup X_3).TP\) is their overlap, and \( X_9 \) is their intersection. Whilst \( X_2 \) is the \( lhs \) of rule \( \text{blame\_good\_content} \) and \( X_8 \) is the intersection between the new created pattern \( q \) of pattern \( FP \) with \( lhs \).

In all cases the confidence intervals overlap or are close. This suggests that patterns are indeed independent and that this independence is preserved.

<table>
<thead>
<tr>
<th>( (X_{12} \cup X_3).TP )</th>
<th>( X_{12}.TP )</th>
<th>( X_3 )</th>
<th>( X_9 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact</td>
<td>Approx.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>70</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9.1: Initial state
<table>
<thead>
<tr>
<th>Expression</th>
<th>Mean</th>
<th>Lower</th>
<th>Upper</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(X_{12} \cup X_3)_{TP}$</td>
<td>10.6</td>
<td>3.68</td>
<td>17.51</td>
</tr>
<tr>
<td>$[X_{12}\cdot TP][X_3]/[X_9]$</td>
<td>10.24</td>
<td>4.23</td>
<td>16.24</td>
</tr>
<tr>
<td>$(X_{12} \cup X_2)_{FP}$</td>
<td>17.4</td>
<td>10.39</td>
<td>24.4</td>
</tr>
<tr>
<td>$[X_{12}\cdot FP][X_2]/[X_8]$</td>
<td>12.28</td>
<td>8.85</td>
<td>15.7</td>
</tr>
<tr>
<td>$(X_8 \cup X_2)_{TN}$</td>
<td>1.2</td>
<td>0.05</td>
<td>2.34</td>
</tr>
<tr>
<td>$[X_8]\cdot [X_2]/[X_8]$</td>
<td>5.44</td>
<td>2.08</td>
<td>8.79</td>
</tr>
<tr>
<td>$(X_9 \cup X_3)_{FN}$</td>
<td>1.4</td>
<td>0.4</td>
<td>3.21</td>
</tr>
<tr>
<td>$[X_9]\cdot [X_3]/[X_9]$</td>
<td>3.1</td>
<td>1.52</td>
<td>6.07</td>
</tr>
</tbody>
</table>

Table 9.2: Final state

### 9.4 Simulation vs. ODE model

This section presents the result of the simulation and ODE models to find out how closely the ODE model captures the behaviour of the explicit, rule-based representation.

**Experimental Setup:** For the simulation experiments we use a graph created randomly using the method presented in Section 5.7.2. The graph consists of 20 nodes of users, linked by 50 edges, 10 nodes of contents equally shared between $appr=false$ and $appr=true$, and 55 like edges from users to contents. Rates for each rule are $like\_content\_u = 1$, $blame\_good\_content = 1$, $blame\_bad\_content = 10$, $switch\_like = 10$, $switch\_friend = 1$. The simulation has a batch size of 5 runs sharing the same simulation parameters.

The source code for encoding the different equations in Octave is given in the listing below.

```octave
tic;
function xdot = voter(X, t)

    like_content_u = 1;
    blame_good_content = 1;
    blame_bad_content = 10;
    switch_like = 10;
    switch_friend = 1;
```

135
\[
\begin{align*}
&x_{dot}(1) = 0; \\
&x_{dot}(2) = 0; \\
&x_{dot}(3) = 0; \\
&x_{dot}(4) = 0; \\
&x_{dot}(5) = 0; \\
&x_{dot}(12) = 0; \\
&x_{dot}(13) = 0; \\
&x_{dot}(14) = 0; \\
&x_{dot}(15) = 0; \\
&x_{dot}(16) = 0; \\
&x_{dot}(17) = 0; \\
&x_{dot}(18) = 0; \\
&x_{dot}(19) = 0; \\
&x_{dot}(20) = 0; \\
&x_{dot}(21) = 0; \\
&x_{dot}(22) = 0; \\
&x_{dot}(23) = 0; \\
&x_{dot}(24) = 0;
\end{align*}
\]

#PatternTP => X(6) --> Cf = light green
\[
\begin{align*}
&x_{dot}(6) = \left( X(12) \cdot X(9) - \frac{X(17)}{X(12) \cdot X(9)} + \frac{X(14)}{X(12) \cdot X(9)} \right) \cdot \text{blame\_bad\_content} \cdot \left( + \frac{X(12)}{X(13)} + \frac{X(14)}{X(9)} \right);
\end{align*}
\]

#PatternFP => X(7) --> Ct = black
\[
\begin{align*}
&x_{dot}(7) = \left( X(12) \cdot X(8) - \frac{X(17)}{X(12) \cdot X(8)} + \frac{X(15)}{X(12) \cdot X(8)} \right) \cdot \text{blame\_good\_content} \cdot \left( + \frac{X(12)}{X(13)} + \frac{X(15)}{X(8)} \right);
\end{align*}
\]

#PatternTN => X(8) --> Ct = dark blue
\[
\begin{align*}
&x_{dot}(8) = \left( X(12) \cdot X(8) - \frac{X(17)}{X(12) \cdot X(8)} + \frac{X(15)}{X(12) \cdot X(8)} \right) \cdot \text{blame\_good\_content} \cdot \left( - \frac{X(8)}{X(15)} - \frac{X(8)}{X(15)} \right);
\end{align*}
\]

#PatternFN => X(9) --> Cf = Dark green
\[
\begin{align*}
&x_{dot}(9) = \left( X(12) \cdot X(9) - \frac{X(17)}{X(12) \cdot X(9)} + \frac{X(12)}{X(12) \cdot X(9)} \right) \cdot \text{blame\_bad\_content} \cdot \left( - \frac{X(9)}{X(14)} - \frac{X(9)}{X(14)} \right);
\end{align*}
\]

\[
\begin{align*}
&x_{dot}(10) = \left( (X(16) \cdot X(17) \cdot X(24)) / (X(12) \cdot X(12) \cdot X(9)) \right) \cdot \text{like\_content\_u} \cdot \left( + \frac{X(13)}{X(13)} + \frac{X(13)}{X(13)} + \frac{X(16)}{X(12)} + \frac{X(10)}{X(17)} \right) \ldots \\
&+ \left( (X(17) \cdot X(16) \cdot X(17) \cdot X(24)) / (X(12) \cdot X(9)) \right) \cdot \text{switch\_like} \cdot \left( - \frac{X(10)}{X(11)} - \frac{X(10)}{X(11)} - \frac{X(10)}{X(17)} - \frac{X(10)}{X(17)} + \frac{X(17)}{X(17)} \right)
\end{align*}
\]
(17) + X(17)/X(17) + X(11)/X(18) + X(19)/X(20)) ... 
+ ( (X(16) * X(17) * X(17))/(X(12) * X(9)) - (X(16)/X(12)^2) ) * 
switch_friend * ( - X(10)/X(11) - X(10)/X(11) - X(10)/X(16) - X(10) 
/X(16) + X(16)/X(16) + X(16)/X(16) + X(21)/X(22) + X(21)/X(22));

xdot(11) = ((X(16) * X(17) * X(24))/(X(12) *X(12) *X(9))) * 
like_content_u * ( - X(11)/X(17) - X(11)/X(10) + X(16)/X(12)) ... 
+ ((X(17) * X(16) * X(17) * X(24))/(X(12)^3 * X(9))) * 
switch_like * ( - X(11)/X(11) - X(11)/X(17) - X(11)/X(17) - X(11)/X(10) + X(23)/X(18) + X(11)/X(18) + X(17)/X(17)) ... 
+ ( (X(16) * X(17) * X(17))/(X(12) * X(9)) - (X(16)/X(12)^2) ) 
* switch_friend * ( - X(11)/X(11) - X(11)/X(16) - X(11)/X(16) + X(23)/X(22) + X(11)/X(16));

endfunction

x = lsode("voter", [0, 0, 0, 0, 0, 0, 5, 5, 10, 10, 20, 1, 1, 1, 50, 55, 
1, 1, 1, 1, 1, 1, 1], (t = linspace (0, 0.009, 100)')); 
plot(t,x);

Fig. 9.10 shows the overall behaviour of the observed patterns over time in 
ODE, which is similar to the ones captured in the simulation, see Fig. 9.9.
Table 9.3 shows the results for both, simulations and ODEs when the population stays steady. They stabilise in a similar way, but to be valid statistical evidence we would expect the values for the ODEs to be within the confidence intervals of the corresponding simulation results, which is the case for all the patterns.

The runtime required for computing solutions in both models differs considerably. The ODE model is significantly faster at 4 seconds than the simulation at 376.65 seconds. More importantly, the effort for solving the ODEs is independent of the size of the population whereas in general simulations on larger graphs require both more steps to cover the same interval of simulated time and more time and memory for graph matching.
Plotting the number of occurrences of these observables over time we can judge the effectiveness of different versions of the model. A high rate of false positives indicates that, as suggested in question 1 above, the system might be open to abuse by suppressing inconvenient views. In classification terms this would result in low precision $\frac{TP}{TP + FP}$. A high rate of false negatives would raise doubts about the effectiveness of the system identifying inappropriate content, resulting in a low recall $\frac{TP}{TP + FN}$.

The result in the Fig 9.11 shows that the system is sensitive to the correct behaviour of users. Even if the rate of blame_good_content is only a 10th of blame_bad_content, FP (black) is close to half of TP (yellow).

![Figure 9.11: Pattern evolution: original model](image)

We therefore consider a variant where the rules for appointing blame are restricted such that content to be blamed is liked by a friend, see Fig. 9.12 and the below List for ODEs source code in Octave.

```
tic;
function xdot = voter(X, t)
    like_content_u = 1;
```

139
blame_good_content_u = 1;
blame_bad_content_u = 10;
switch_like = 10;
switch_friend = 1;

xdot(1) = 0;
xdot(2) = 0;
xdot(3) = 0;
xdot(4) = 0;
xdot(5) = 0;

xdot(12) = 0;
xdot(13) = 0;
xdot(14) = 0;
xdot(15) = 0;
xdot(16) = 0;
xdot(17) = 0;
xdot(18) = 0;
xdot(19) = 0;
xdot(20) = 0;
xdot(21) = 0;
xdot(22) = 0;
xdot(23) = 0;
xdot(24) = 0;
xdot(25) = 0;
xdot(26) = 0;
xdot(27) = 0;

#PatternTP => X(6) = cf
xdot(6) = ( (X(20) * X(15))/X(23) - (X(15)/(X(23) * X(9)) + X(12)/(X
(23) * X(9))) ) * blame_bad_content_u + ( + X(12)/X(9) + X(12)/X(9))
;

#PatternFP => X(7) = ct
xdot(7) = ( (X(20) * X(15))/X(23) - (X(15)/(X(23) * X(8)) + X(13)/(X
(23) * X(8))) ) * blame_good_content_u + ( + X(13)/X(8) + X(13)/X
(8));

#PatternTN => X(8) = ct
xdot(8) = ( (X(20) * X(15))/X(23) - (X(15)/(X(23) * X(8)) + X(13)/(X
(23) * X(8))) ) * blame_good_content_u + ( - X(8)/X(13) - X(8)/X
(13));
#PatternFN => X(9) = cf

\[ xdot(9) = \left( \frac{X(20) \cdot X(15)}{X(23)} - \frac{X(15)}{(X(23) \cdot X(9))} + \frac{X(12)}{X(15) \cdot X(9)} \right) \cdot \text{blame_bad_content}_u \cdot \left( - \frac{X(9)}{X(12)} - \frac{X(9)}{X(12)} \right) \]

#PatternSameOp => X(10)

\[ xdot(10) = \left( \frac{X(20) \cdot X(15) \cdot X(27)}{(X(23) \cdot X(23) \cdot X(9))} \cdot \text{like_content}_u \cdot \left( + \frac{X(14)}{X(14)} + \frac{X(14)}{X(14)} + \frac{X(10)}{X(15)} + \frac{X(11)}{X(13)} \right) \ldots \]

\[ + \left( \frac{X(15) \cdot X(20) \cdot X(15) \cdot X(27)}{(X(23)^3 \cdot X(9))} \cdot \text{switch_like} \cdot \left( - \frac{X(10)}{X(11)} - \frac{X(10)}{X(15)} - \frac{X(11)}{X(15)} - \frac{X(10)}{X(11)} + \frac{X(14)}{X(25)} + \frac{X(15)}{X(15)} \right) \ldots \]

\[ + \left( \frac{X(20) \cdot X(15) \cdot X(15)}{(X(23) \cdot X(9))} - \left( \frac{X(20)}{X(23)^2} \right) \cdot \text{switch_friend} \cdot \left( - \frac{X(10)}{X(11)} - \frac{X(11)}{X(15)} + \frac{X(20)}{X(23)^2} \right) \ldots \]

#PatternDiffOp => X(11)

\[ xdot(11) = \left( \frac{X(20) \cdot X(15) \cdot X(27)}{(X(23) \cdot X(23) \cdot X(9))} \cdot \text{like_content}_u \cdot \left( - \frac{X(11)}{X(15)} - \frac{X(10)}{X(15)} + \frac{X(20)}{X(23)} \right) \ldots \]

\[ + \left( \frac{X(15) \cdot X(20) \cdot X(15) \cdot X(27)}{(X(23)^3 \cdot X(9))} \cdot \text{switch_like} \cdot \left( - \frac{X(11)}{X(11)} - \frac{X(11)}{X(15)} - \frac{X(11)}{X(15)} - \frac{X(11)}{X(15)} + \frac{X(14)}{X(25)} + \frac{X(15)}{X(15)} \right) \ldots \]

\[ + \left( \frac{X(20) \cdot X(15) \cdot X(15)}{(X(23) \cdot X(9))} - \left( \frac{X(20)}{X(23)^2} \right) \cdot \text{switch_friend} \cdot \left( - \frac{X(11)}{X(11)} - \frac{X(11)}{X(20)} + \frac{X(26)}{X(22)} \right) \ldots \]

endfunction

x = lsode('voter', [0, 0, 0, 0, 0, 0, 0, 5, 5, 35, 35, 1, 1, 1, 50, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 80, 1, 1, 20, 1, 1, 1, 1, 1], (t = linspace(0, 0.01, 100)'));
plot(t, x);
If, as is expected in the longer term, friends share opinions, a user will not fundamentally disagree with the content liked by a friend, but may still perceive as inappropriate the way it is expressed. As seen in the Fig 9.13, the behaviour changes significantly. In our interval, FP rises to less than 15% of TP. TP also rises much more quickly. Longer term FP catches because even once blame_bad_content is no longer enabled because there is no more bad content
to blame, \textit{blame_good_content} remains applicable until all blame is used up. However, if we look at $t = 0.003$ TP is already at maximum while FP and FN are negligible. This suggests that timing information can be used as part of the process to identify content to be blocked, i.e. if content is blamed earlier this is more significant than later blame.

Figure 9.13: Pattern evolution: with restricted blaming
Chapter 10

Evaluation

In this chapter we evaluate the results of all three analysis methods and discuss their limitations and trade-offs. The evaluation result and comparison of the approaches will be based on the Voter case study, since the result of Peer-to-Peer Content Policing case study is already evaluated in Chapter 9. Moreover, we discuss how the approaches work together and support each other. Sections 10.1 10.2 hold these discussions. The chapter will end with related works in Section 10.4.

10.1 Discussion of Analysis Results

This section briefly discusses alternative analysis techniques applicable to graph transformation systems, namely model checking, stochastic simulation, and ordinary differential equations (ODEs), comparing and contrasting their advantages, disadvantages and applicability to our case study.

As it was addressed earlier the major bottleneck of using graph transformation to model adaptive networks is the scalability of their analysis such as state space generation in the derivation of the CTMC. In order to address this problem, we have presented three analysis methods, attempting to understand how the specific model works by performing the quantitative analysis in each approach based on the two case studies Voter Chapter 3 and Peer-to-Peer Con-
ent Policing Chapter 9. The results of the three approaches are compared in order to answer the analysis questions asked in Chapter 1. Here we evaluate the comparison experiments we have conducted for the Voter model. The results of the approaches are compared in a small as well as a scaled system in order to validate the scalability challenge in our approach.

To compare the approaches we use the Weighted average Frequency of Patterns in the Steady state (WFPS) as a characteristic measures. For a pattern $p \in P$ this is the average number of occurrences of $p$ across all states, weighted by the probability of states in the steady state solution.

$$WFPS(p) = \sum_{s \in S} Pr(s) \times [p]_s$$

$Pr(s)$ is the steady-state probability of the state $s$ and $[p]_s$ the number of occurrences of the pattern $p$ in state $s$. While the formula refers to the state space of the model, it is easy to determine $WFPS(p)$ from simulations using the long-term average or final value of $[p]$. For differential equations it is obtained by solving for $d[p]/dt = 0$, or reading out the stable value of the numerical solution.

We take the confidence intervals for the results we obtain from the simulation experiments, to check whether the result of the other approaches fall into the range of that intervals. The equations is

$$x \pm 1.96 \frac{\sigma}{\sqrt{n}}$$

The confidence interval is a range of values. Where $x$, the weighted average frequency, is at the centre of this range and the range is $x \pm Margin of Error$. $\sigma$ is the standard deviation and $\sqrt{n}$ is the sample size. We calculate the area under the standard normal curve that equals 95 percent. This value is $\pm 1.96 ^{[37]}$. 

145
10.1.1 Experiment 1

Table 10.1 and the plot in Fig. 10.1 show the analysis results of the approaches model checking in Chapter 4 and simulation experiment 1 in Section 5.7.1. We have included the result of ODE approach as well to check how it results from the small system and it is interesting to compare it with the other two approach. The results of this experiment show the Weighted average Frequency of Patterns in the Steady state.

The differences are not significant, all the results are in the range of confidence intervals apart from the WFPS for 01 edge, which is higher in the model checking approach. It was expected to decrease over time because agents are less likely to stay friendly in case they disagree. The WFPS is also higher for 00 edge in the simulation and lower for 11 edge in the model checking. This is investigated more by performing further experiments in order to obtain more accurate and reasonable results which is achieved in Section 10.1.3. By applying the approaches on larger graphs.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Model Checking</th>
<th>Stochastic Simulation</th>
<th>ODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\approx 2.50$</td>
<td>$3 \pm 1.24 [1.76 \text{ to } 4.24]$</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>$\approx 2.50$</td>
<td>$1.8 \pm 1.69 [0.114 \text{ to } 3.48]$</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>$\approx 4.3$</td>
<td>$0.6 \pm 0.48 [0.12 \text{ to } 1.08]$</td>
<td>$\approx 1.4$</td>
</tr>
<tr>
<td>00</td>
<td>$\approx 1.7$</td>
<td>$5.4 \pm 3.26 [2.15 \text{ to } 8.66]$</td>
<td>$\approx 2.2$</td>
</tr>
<tr>
<td>11</td>
<td>$\approx 1.7$</td>
<td>$4.6 \pm 3.8 [0.799 \text{ to } 8.4]$</td>
<td>$\approx 3.9$</td>
</tr>
</tbody>
</table>

Table 10.1: The WFPS in a small model

The time for analysing each model is shown in seconds in Table 10.2. In model checking, the time needed for the state pace exploration (100 states and 2336 transitions) was 1.013 seconds, and the steady state solution was $\approx 2$ seconds. The simulation needed 5.7 seconds for 5 runs over 368 step numbers.

<table>
<thead>
<tr>
<th></th>
<th>Model Checking</th>
<th>Stochastic Simulation</th>
<th>ODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\approx 3.013$</td>
<td>5.7</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Table 10.2: Time spent in each method
Based on this experiment, the results are consistent with the claim that
the simulation average over the entire set of possible runs matches the model
checking and ODE result. However, they do not prove it positively. The results
merely do not contradict each other.

10.1.2 Experiment 2

This section evaluates and addresses the question of how closely does the ODE
approach capture the behaviour of the explicit, rule-based representation. We
compare the results of the second simulation experiment in Section 5.7.2 with
the ODE results we obtained in Chapter 8. In the experiments, different param-
eters are used. In particular, in order to validate the results obtained from the
ODEs against the stochastic simulation results, one set of parameters matches
those of the simulation. The aim is producing the same results we obtained in
the simulation model.

Since the ODEs are approximating the averages over all the system (not
reflecting the randomness observable in individual simulation runs), their solu-
tion is fixed once the parameters are given. It is one of the advantages of ODEs
that gives a quicker description of the dynamics of a system.

Table 10.3 and Figures 10.2 to 10.4 illustrate the WFPS of all the patterns
and their timing behaviour in the system. The results show that over time the
individuals holding certain votes are behaving the same way in both analysis
techniques, which retains the similar or same number of occurrences as initially. This happens because of giving the same rate to both conversion rules provides equal chance for their application, since they are the only rules that create and destroy pattern \([0]\) and pattern \([1]\).

Results also show that the relation between the individuals evolve similarly in both approaches, i.e., agents tend to stay together when they have similar opinion, or break the relation when they have different opinion to reconnect to an agent with similar opinion. Likewise in the simulation experiment, in ODE the number of 01 edges decrease gradually. This happens due to the effect of the rules on the pattern \([01]\). Each rule will destroy at least one occurrence, however rules convertTo0, convertTo1 might create and destroy more than one pattern \([01]\) in each application. The effect of rules also increase the number of the relations between agents having the same vote (11, 00 patterns) over time. The split rules create at least one 11-edge or 00-edge in each application, whereas the conversion rules might create and destroy more than one 11-edge or 00-edge in one application. Based on the rates we have given to the rules, the conversion rule matches are overwhelmed by the matches of the split rules and therefore there are more chances that conversion rules’ application are chosen over split rules.

The plots also show that \([11]\) rises more than \([00]\) because of the majority population of \([1]\). This means, even though the rates are equal for the split rules, matches for splitTo1 will be more frequent than for splitTo0.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Stochastic Simulation</th>
<th>ODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9.8 ± 2.59 [7.21 to 12.39]</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>38.2 ± 2.59 [35.61 to 40.79]</td>
<td>40</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0.16</td>
</tr>
<tr>
<td>00</td>
<td>27.8 ± 9.4 [18.3 to 37.2]</td>
<td>15</td>
</tr>
<tr>
<td>11</td>
<td>99 ± 9.2 [89.8 to 108.2]</td>
<td>105</td>
</tr>
</tbody>
</table>

Table 10.3: The Weighted Average Frequencies of Patterns in a larger model

Even though in ODE the WFPS for pattern \([00]\) is not in the range of
Figure 10.2: The evolution of the system over time in simulation method

Figure 10.3: The evolution of the system over time in ODE method

Figure 10.4: The Weighted Average Frequency Plot
confidence interval of simulation as it is seen in Table 10.3, the WFPS of the patterns are similar and their behaviour trends and evolves similarly.

Times that were needed for obtaining the solutions for both models are shown in Table 10.4. The ODE model was significantly faster than the simulation. It only took approximately 2 seconds, whereas 33.384 seconds were taken for the simulation model.

<table>
<thead>
<tr>
<th>Stochastic Simulation</th>
<th>ODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.284</td>
<td>$\approx 2$</td>
</tr>
</tbody>
</table>

Table 10.4: Time spent in each method

Thus, the comparison suggests that our approach is an alternative method for analysing GTSs modelling adaptive networks.

10.1.3 Further experiments

We conducted more experiments on the Voter system using different start graphs to understand the further differences between the approaches.

**Experiment 3**: The test was carried out on the system which had the same input for both, simulations and ODEs. Due to the scale of the graph, the model checking approach was not tested. The system was populated with 100 nodes equally shared between patterns 0 and 1, linked by 166 00 edges, 182 11 edges, and 161 01 edges. Split and conversion rules have the same rate of 1.

Table 10.5 shows the results for both models. It shows the system behaves in a similar way and the values for the ODEs are within the confidence intervals of the corresponding simulation results. However, the values for the patterns 00 and 11 are close to the boundary of the interval range. These variation could be due to the limited number of the simulation runs and the average values that were taken over all the runs. The results are consistent and do not contradict the hypothesis that the models are equivalent.

The time required for computing solutions in both models differs considerably. The ODE model is significantly faster at 3 seconds than the simulation
at 259.95 seconds. More importantly, the effort for solving the ODEs is independent of the size of the population whereas in general simulations on larger graphs require both more steps to cover the same interval of simulated time and more time and memory for graph matching.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Stochastic Simulation</th>
<th>ODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>44.8 ± 4.77 [40.02 to 50]</td>
<td>50</td>
</tr>
<tr>
<td>1</td>
<td>55.2 ± 4.77 [50.4 to 59.9]</td>
<td>50</td>
</tr>
<tr>
<td>00</td>
<td>295.6 ± 33.6 [262 to 329.2]</td>
<td>327</td>
</tr>
<tr>
<td>11</td>
<td>376 ± 33.6 [342.4 to 409.6]</td>
<td>343</td>
</tr>
<tr>
<td>01</td>
<td>1 ± 0 [1 to 1]</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 10.5: The Weighted Average Frequencies of Patterns in both approaches

**Experiment 4:** This experiment was to investigate further the scalability of each approach and explore their boundaries. In this experiment we have scaled up the systems and have used different inputs for each approach in order to examine and be able to show how large each model can be coped with.

The input graph used for model checking consisted of 10 nodes and 9 edges, it is illustrated in Figure 10.5. The time needed to generate the state space was unknown, it eventually ran out of memory.

With regards to the simulation, a graph of 100 nodes used, 50 each of agents of either opinion, and 1000 edges. In a batch, each simulation run needed 115.76585 minutes to complete. We could not go beyond this boundary due to the memory use up.

The same input graph of the simulation used for the ODEs, it took only 3-4 seconds to solve the equations. There was actually no limitation on how large is the system. With the same speed we obtained a result for a 15 million populated system.

**Experiment 5:** In our simulation we also investigated if changing the ratio of rates between splitting and conversion rules causes the system to behave differently. An interesting phenomenon occurred at a convert-to-split ratio of above 400 where the size of the patterns of 0 and 1 nodes fluctuated more
strongly and in many cases the minority ones declined until reaching extinction. A plot is shown in Fig 10.6, obtained on a graph with 48 nodes, 40 voting 1 and 8 voting 0, as well as 60 edges of type 01. Using the same rates in the ODE method, this phenomenon did not occur.

The phenomenon, known as Gambler’s Ruin [50], shows a limitation of differential equations in capturing random behaviour due to the fact that only the mean behaviour is reflected. The observation is that minority populations are more vulnerable to fluctuations and therefore more likely to go extinct, even if the odds are fairly distribute. This is visible in simulations while the ODE produces a flatline graph as average mean over all possible behaviours.

10.2 Limitations, Trade-offs, and Supports

The analyses show that each method can address different questions. All methods can address questions about the Weighted average Frequency of Patterns in the Steady state (WFPS). Stochastic simulation and ODE approaches can also answer the questions about the system’s behaviour, and the evolution over time
of agents hold certain votes and their connections. Whereas the model checking graph transformation limitation does not allow for this analysis. Further limitations to this approach, model checking with explicit states does not scale well to models with large state space. Since non-functional properties often depend on the behaviour of large populations (network nodes, connections, services, etc.), this limitation is significant.

To tackle that, the stochastic simulation could replace this approach. Although like the model checking graph transformation, based on our experiments, simulation can suffer the same problem when the system scales and become more complex, also the time efforts each experiment demand, in term of result integration and performing the statistic analysis, is significant.

Therefore, stochastic model checking and simulation can be expensive and less scalable. They might consume a significant amount of time to produce the results. Despite these disadvantages, we challenge the Gambler’s ruin phenomenon in the simulation model when in the graph the global population is split up into small populations, i.e., when the graph is not homogeneously connected. This causes the minority individuals decline over time and become 0 when the simulation terminates.

Unlike the two previous approaches, ODEs provide a more flexible trade-off between analysis effort and confidence in the result and so allow to verify soft
performance targets in large-scale systems.

Based on the analyses we propose that our approach efficiently is an alternative method to model and analyse graph transformation systems and more scalable and flexible than the other two methods. Also another advantages of ODEs is that they give a quicker description of the dynamics of a system. However, the pair approximation and deriving the differential equations are a demanding task, which require automation to scale to models of more interesting size and patterns. Our approach includes the automation that tackles this challenge. Another potential problem is the limitation of the ODEs, which they do not reflect the randomness of rule application and structure of the graph, i.e., the topology that separating a graph into subgraphs.

The trade off between all three different approaches are various. The model checking and simulation methods are more accurate for small graphs but they less scalable for complex systems. Whilst ODEs are very scalable and flexible but less accurate than the other models since it approximates the averages over all the system, i.e., approximating the effects of the rules on the patterns. Having said that, as it was mentioned previously that the approximation used in our approach will generally, but not always, become asymptotically exact as the underlying graph size goes to infinity, which is where it is needed most.

The three different approaches can work together in a way that they support each other. The models work as levels of stairs that come after each other. One can be used to validate the result of the other. For example, for small or simple systems it is more effective to use the model checking analysis since it is more accurate. For the scaled systems, simulation can be used as alternative method. In order to do that, we need to check whether both models have the same results in a small system. If the results were similar, this would confirm that both models are bisimilar and stochastic simulation can be used for more complex systems. The same process is used to validate the ODE model against the simulation model in order to use ODE for more complex systems. Thus the simulation approach links both model checking and differential equation
To test different examples and further check the scalability we have conducted extra experiments using more complex case study in order to see how our approach scales when patterns and rules become larger and more complicated. The results again show that our approach is flexible and scales well to these complexity. The details of this analysis are well-presented in Chapter 9.

The complexity of computing critical pairs and proving their joinability arises the need for tool support. We have implemented that tool and it supports the computation of critical pairs for attributed graphs. That means the approach scales to the increasing of the number of parameters/attributes nodes have, however this may produce more critical pairs and computing them may become more complicated and take longer.

10.3 Threats to Validity

The voter model is simple, and does not feature complex left-hand sides nor patterns. That means, it does not say much about the scalability of the approach and particular types of critical pairs or dependencies will not have been encountered. However critical pair analysis, which represent the main computational step in the derivation of ODEs, is an established technology which has been used in larger and more complex case studies. The fact that the graph structure in the voter model is unrestricted suggests that an accurate set of ODEs may require an infinite set of equations.

Simulations do not scale well to larger graphs and more runs, so variances observed are high and confidence intervals quite broad and all runs start from the same random graph. This weakens claims as to the consistency of the simulation and ODE analyses. Indeed it has been observed that the results are not a perfect match in some circumstances such as Gambler’s Ruin phenomenon.

In the evaluation we work on random graphs with the assumption of pattern independence. But, data from real social networks is likely to have more depen-
encies between patterns. That means the model as it is may not be applicable to such data, but in this case a model using weaker assumptions could be constructed. For this to be possible automatically, our implementation allows to specify embargoes on patterns that must not be approximated because their component patterns are not deemed independent, leading to more and larger patterns and more equations.

10.4 Related Work

10.4.1 Model Checking and Simulation of Graph Transformations

Previous works [8] aimed to introduce modularity in model checking graph transformation systems to reduce complexity and handle scalability issues of the state space generation. They have used three approaches, monolithic, top-down and bottom-up. First a system is modelled as a global or whole-world system using the monolithic approach. In the top-down approach a global system is decomposed into local views which reduces the rules and start graph of graph transformation systems to their local perspectives based on their type graphs. Correspondingly smaller LTSs are then generated from these views independently and transformed into CTMCs. These CTMCs are further translated into PEPA. In PEPA subsystems are synchronized over shared interfaces to compose a global system using the bottom-up approach. The authors demonstrate that the composed model is bisimilar to its original global model. This way, the approach provides more scalable techniques to perform stochastic analysis.

Similar to the previous work, in [40, 41] a notion of composition of graph transformation systems is defined, where rules are decomposed into subrules and then original rules are constructed by merging corresponding subrules. Unlike the previous work, they do the decomposition at instance level not at type level.
Work on abstraction in graph transformation has followed a variety of approaches and motivations. Much of the existing literature on abstraction of graph transformation systems focuses on abstraction as a means to improve comprehensibility of complex GTS by altering the visibility of substructures. In [38], it is a means to improve comprehensibility of complex GTS by hiding and retrieving substructures as required. [9] uses stochastic simulation to model and analyses graph transformation systems. They have proposed abstractions of such systems to increase the scalability of this analysis. They have formalised a methodology that hides details in the lowest level of the hierarchy, but retains any important information as attributes. Bayesian networks, are used to train the parameters of the abstract system so that it shows behaviour consistent with the original model.

Existing work on the use of abstraction in aiding analysis of GTS models focuses mainly on model checking, using temporal logic on abstract graphs to check properties of the original concrete system. [11] uses neighbourhood abstraction to group graph elements via some equivalence relation up to some radius defining a node’s neighbourhood. This allows the level of precision of abstraction to be adjusted if the current abstraction does not allow the verification of properties. The resulting graph can be expanded to concrete equivalents through edge multiplicities in the abstract graph. The abstraction yields abstract graphs of bounded, finite size even if the concrete system can consist of unbounded graphs. This obvious reduction in state space simplifies analysis by model checking. However, the formalism does not allow for negative application conditions, which might be important in social network applications. [51] uses a similar approach, but rather than using equivalence relations to group nodes, abstracted nodes are characterized by satisfaction of temporal logic formulae that represent some behavioural property of the concrete system. In [43], abstraction is used to reduce the state space required by the labelled transition system that outlines the behaviour of a GTS. Based on shape graphs introduced in [45], nodes are grouped by structural similarity with multiplicities to capture
concrete representations of an abstract shape. Several states are therefore combined into a single structure. In this work, while the abstraction creates a bounded labelled transition system from a potentially unbounded one, certain properties that can be verified at the concrete level cannot at the abstract one.

10.4.2 Pattern Approximation and Differential Equations

All the previous approaches keep within the same formalism, replacing one graph transformation system by a more abstract one. But, in [44] pair approximations are used to derive a system of ordinary differential equations from a rule-based graph transformation system. In this thesis, we proceed in the same spirit, but differently. The basic notion of pair approximation only works with individual edges, we extends that to include bigger/general patterns and negative conditions, both essential features in rule-based modelling. Similar to the first case study in this thesis, [44] has used the Voter model to test their approach but based on simple, symmetric graphs and with rules described informally in English. Whereas, our model is a formal graph transformation system whose rules model the coevolution of votes and connections. Further, they consider both rewire-to-same and rewire-to-random while we only are interested in the rewire-to-same version. Rewire-to-same is when two connected individuals hold different opinions, the link between them is broken and one of the individuals reconnecting to another chosen from those with the same opinion. Whereas rewire-to-random reconnects one of the individuals to a random one from the network as a whole.

The idea of analysing rule-based systems via ODEs appears in [20, 14, 13, 10]. They use patterns as variables in a set of ODEs that capture the dynamics of the observables of interest in the original system, either based on a notion of refinement, which is only possible with certain graph structures, or accepting to generate potentially infinitely many ODEs. Fragmentation [20, 14, 10], discussed in Chapter 1, works well for the restricted kinds of graphs representing
biochemical and chemistry compounds. For general graphs an accurate analysis
of SGTS by ODEs is not possible because of the following problem: In order
for the equations to accurately reflect the operation of the system they need to
capture

1. The number of occurrences of each relevant rule.

2. The effect of the rule’s application on occurrences of all patterns \( p \in P \).

The first requirement means that, for a rule’s left-hand side \( L \), the number
of \( L \)’s occurrences in any reachable graph \( G \) has to be expressible in terms
of patterns in \( P \). The second requirement enforces a refinement of rules \( r \) by
rule instances whose effect on pattern occurrences can be predicted statically,
without reference to the specific application context. That means to consider
embeddings of \( r \) into all contexts resulting from overlapping its left or right hand
side with a pattern, resulting in a cycle of pattern-based rule refinements that
leads to larger rules with larger left-hand sides creating new patterns, leading
to further rule refinements, etc. Therefore fragmentation delivers an accurate
analysis in a finite number of steps only if at some stage no further relevant
refinements can be produced and/or suitable invariants can be found such that
all new patterns can be expressed by existing ones. Our approach follows the
idea of using patterns, but approximates them to avoid infinite derivations.

Moreover, the approach in [10] is a domain-specific analysis. The paper de-
scribes a methodology that extracts a system of differential equations describing
the evolution of the system from a discrete, structural one. They have used the
accountability assumption (without using the notion of rigidity [32]) that the
number of occurrences of each pattern \( p \in P \) created and destroyed by each
rule \( r \in R \) are fixed. The rules are refined and expressed by a place-transition
net and then a set of ODEs is extracted from the incidence matrix of that
PT net. Their equation is \( \frac{d[X]}{dt} = S.R \), where \( \frac{d[X]}{dt} \) is the differential
with respect to time, \( t \), of a chemical species, \( X \), in the system, \( S \) is the stoi-
chiometrix matrix, and \( R \) is the rate law vector. In additional, their approach
is tested by means of a simple unimolecular nucleophilic substitution (SN$_1$) re-
action which was chosen for its simplicity as a fundamental step in testing and
demonstrating the basics of their methodology. Naturally models in chemistry,
biology, and biochemistry are structured so that they can be accountable and
rigid. For example, a particular carbon atom can have exactly two connections
to other carbon atoms. This is rigidity, as it has named or limited connections,
implies that the exact connections are known for particular nodes. This makes
the approach less applicable to adaptive networks, which are less structured
than, e.g., models in biochemistry and biology. Applying this method on an
adaptive network would generate infinitely many equations. In the thesis we
do not use rigidity but pair approximation based on pattern independence to
avoid infinite derivation. Furthermore, their method is not evaluated by carry-
ring out a comparison against the stochastic model checking and simulation to
determine how the results compare and complement each other. In contrast,
our approach is evaluated and compared to the existing approaches and the
results are promising. Currently their approach require manual observation to
derive intermediate reaction results. For unbounded reactions the automation
is necessary.

In counterexample-guided abstraction refinement based on unfoldings [35],
the behaviour of a GTS is represented by a Petri graph representing an approx-
imated unfolding. The approach [35] derives a Petri net-like representation of
the unfolding, i.e., at the semantic level, while [32] translates the specification
into a Petri net.

The proposed approach in [32] uses a refinement technique developed for the
Kappa graph rewriting approach to derive a stochastic (place-transition) Petri
net from a stochastic graph transformation model, replacing graphs as states
by markings representing the frequency of occurrences of certain patterns and
rules as net transitions. In order to represent a rule as a net transition, the rule
has to have a deterministic effect on the patterns represented by the nets places,
creating and destroying a fixed number of occurrences of all patterns. This is not
the case in general, where the effect of the rule on certain patterns may depend on the context of the application. A system whose rules are deterministic in this sense is called *balanced* and, assuming limits on the degree of the graphs which are also preserved by the rules, a graph transformation system can be transformed into an equivalent balanced one. However, this technique does not guarantee that a rule’s applicability can be captured exactly unless its left-hand side is a pattern itself. Balanced refinements are inspired by Kappa where they play a crucial role in enabling analytical techniques such as the thermodynamic approach [12] and the derivation of differential equations [20, 14].

In our work [32], the idea of the balanced refinements is to replace each rule by a set of extensions that are jointly equivalent (in the sense of a stochastic bisimulation) to the original rule, such that the extended rules describe all the cases in which the original rule could affect any of the chosen patterns. This is possible due to a property of graphs, called *rigidity*, similar to the absence of V-structures in graphs [15] where no node is allowed to carry two or more edges unless they are distinguishable by their types or attributes, or those of their target nodes. To control embeddings of rules into context and preserve rigidity of graphs under rule application, they, [32], employ reflection constraints based on the concept of open maps [27].

With refinement techniques, [32] provides an exact number of the rules’ effects on patterns but with more efforts and only with certain graph structures. Whereas in my approach, instead of rule refinements, we use pair approximation techniques to calculate the average of the rules’ effects on patterns. To restrict the structure of the model, [32] uses *rigidity*. Whilst in my approach we use pattern independence assumption. The latter is a better assumption than rigidity. The reasons, first, our approach is more scalable and theirs become less scalable with unbounded system. Second, adaptive networks are not rigid and making them rigid would potentially limit or lose the scope of the application. Due to these drawbacks and limitations, it was not desired to pursue and use the approach of my paper [32] in this thesis and instead it was preferred to work
with the approximation techniques in order to avoid infinite ODE derivations by approximating the larger patterns with smaller ones and also avoid limiting the approach to certain or restricted graph structures. Other reasons, constructing a balanced refinement is itself a demanding task, which requires automation to scale to models of more interesting size. Another potential bottleneck in [32] is the state space analysis on the constructed Petri net. The evaluation of the thesis suggests that ODEs are more scalable and flexible, also they give a quicker description of the dynamics of a system.
Chapter 11

Conclusion

11.1 Summary of Contributions

We proposed and validated an approach to approximating the dynamic behaviour of a stochastic graph transformation system by a set of differential equations. This approach is scalable and appears to capture most of the long-term behaviour of the original model. Experience shows that different methods can address different questions, offering different tradeoffs between scalability to large models, complexity of properties or accuracy of the analysis. Model checking and simulations are more accurate for small graphs but less scalable to large states and state spaces. Systems of ODEs are known to scale less well to a large number of variables. This is where our approximation has the capacity to limit the number of patterns and thus of variables and equations to be generated. The three different approaches can work together to support each other, e.g., starting from the most accurate and least scalable model checking approach which allows effective debugging to gain confidence in the model, via simulations to explore the system at a somewhat larger scale, to ODEs as an abstract mathematical description. Running different methods on the same version of the model allows to cross-validate results. The generation of ODEs from SGTS is fully automated using AGGs critical pair analysis.
11.2 Challenges and Future Work

In this section we discuss the open questions and challenges with our approach, and possibilities for extending the work in the future.

Limitation of ODEs
As mentioned earlier, one of the concerns and challenges in our approach is the limitation of the ODE models which they do not reflect the randomness of rule applications and structure of the graph, i.e., the topology separating a graph into subgraphs. They might need a better approximation incorporating topology and randomness.

Model Real Data
The approaches in this thesis are validated using quantitative analysis of data gathered from hypothetical models of an adaptive network, i.e., pilot projects. Data obtained from analysing the Voter social network and the Peet-to-Peer Content Policing confirm the results of the model, but a more thorough comparison is still outstanding. In the future, applying our approach on a real data would be another new area of research. The challenge to the approach is that, unlike with random graphs, some patterns in these networks will not be independent. As well as ensuring scalability and collecting data.

Modelling and analysing real data are common problems among other existing approaches. [10] would generate infinite equations due to the accountability assumption they have used, whereas [32] uses rigidity assumption and it becomes less scalable with unbounded adaptive networks and making the system bounded would potentially limit or lose the scope of the application.

However, since the case studies in our work are only intended to illustrate the use of the three approaches, this thesis placed more emphasis on the actual approaches as opposed to an empirical study of the Voter and P2P content policing systems.
Bibliography


[38] Francesco Parisi-Presicce and Gabriele Piersanti. Multilevel graph grammars. In *Graph-Theoretic Concepts in Computer Science, 20th Interna-


