THIAGO RAFAEL BECKER

VeriGraph: A Tool For Model Checking
Graph Grammars

Monograph presented in partial fulfillment
of the requirements for the degree of
Bachelor of Computer Science

Prof. Dr. Rodrigo Machado
Advisor

Porto Alegre, December 17th, 2014
Thiago Rafael Becker,

VeriGraph: A Tool For Model Checking Graph Grammars / Thiago Rafael Becker. – Porto Alegre: Graduação em Ciência da Computação da UFRGS, 2014.

95 f.: il.


1. Model Checking. 2. Graph Grammars. 3. Temporal Logic. I. Machado, Rodrigo. II. Título.
The end of a melody is not its goal: but nonetheless, had the melody not reached its end it would not have reached its goal either. A parable.
(Friederich Nietzsche)
ACKNOWLEDGMENTS

I would like to acknowledge the great people that helped me during this journey, friends and family. In special, I would like to thank my father, mother and siblings for supporting and helping me during the process of writing this monograph: love you all; my brothers and sister: they made growing up less boring; my friends Éder, Daia, Felipe, Gui, and all the others, for the fun and for understanding when I’m absent for being overloaded with work.

I would also like to thank to all the professors that helped in my education, in special Prof. Rodrigo for providing me with the opportunity to do this work, and guidance to deal with academic matters. To the evaluators, for the patience and understanding during this hard time, thank you.
## CONTENTS

<table>
<thead>
<tr>
<th>CONTENTS</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>11</td>
</tr>
<tr>
<td>ACRONYMS</td>
<td>15</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>15</td>
</tr>
<tr>
<td>RESUMO</td>
<td>17</td>
</tr>
</tbody>
</table>

### 1 INTRODUCTION

1.1 Motivation 21
1.2 Goals 22
1.3 Contribution 22
1.4 Structure of this work 22

### 2 GRAPH GRAMMARS

2.1 Preliminary definitions 23
2.2 The double pushout approach 25
2.3 State space generation 30
2.4 Final remarks 32

### 3 MODEL CHECKING

3.1 Model checking 35
3.2 Computational Tree Logic 36
3.2.1 Important equivalences for CTL 40
3.2.2 CTL satisfaction algorithm 41
3.3 Model checking graph grammars 44
3.4 Final Remarks ..................................................... 46

4 IMPLEMENTATION .................................................... 49
4.1 Haskell .............................................................. 49
  4.1.1 Type classes .................................................... 50
  4.1.2 Monads ......................................................... 51
  4.1.3 Parser Combinators ........................................... 52
4.2 Implementation of Graph Rewriting ............................... 52
  4.2.1 Graphs .......................................................... 52
  4.2.2 Rules ........................................................... 54
  4.2.3 Rewriting ....................................................... 57
  4.2.4 Graph matching ................................................ 58
  4.2.5 State space generation ....................................... 58
  4.2.6 Other modules ................................................ 59
4.3 CTL implementation ............................................... 59
  4.3.1 Model ........................................................... 59
  4.3.2 Parser .......................................................... 60
  4.3.3 Semantics ....................................................... 60
4.4 Integration ......................................................... 60
4.5 Experiments ........................................................ 61
4.6 Final remarks ...................................................... 62

5 CONCLUSIONS AND FUTURE WORK ............................... 65
5.1 Future work ......................................................... 65

REFERENCES .......................................................... 67

APPENDICES .......................................................... 71

APPENDIX A .......................................................... 71
A.1 GraphGrammar ...................................................... 71
A.2 GraphGrammar.Graph ............................................. 71
A.3 GraphGrammar.Rule ............................................... 75
A.4 GraphGrammar.Transformation ................................... 77
A.5 GraphGrammar.Match ............................................. 79
A.6 GraphGrammar.StateSpace ......................................... 84
A.7 GraphGrammar.Serialized ......................................... 85
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>An example graph in its set form (left) and graphical form (right)</td>
<td>24</td>
</tr>
<tr>
<td>2.2</td>
<td>An example of graph morphism.</td>
<td>24</td>
</tr>
<tr>
<td>2.3</td>
<td>A typed morphism over $T$</td>
<td>26</td>
</tr>
<tr>
<td>2.4</td>
<td>A graph production</td>
<td>27</td>
</tr>
<tr>
<td>2.5</td>
<td>The derivation of a graph $G$ with the production in Figure 2.4.</td>
<td>28</td>
</tr>
<tr>
<td>2.6</td>
<td>Examples of violations of (a) identification condition and (b) dangling condition</td>
<td>29</td>
</tr>
<tr>
<td>2.7</td>
<td>A grammar depicting a train system</td>
<td>31</td>
</tr>
<tr>
<td>2.8</td>
<td>The full state space generated by the execution of the grammar in Figure 2.7</td>
<td>33</td>
</tr>
<tr>
<td>2.9</td>
<td>An example of a grammar that generates an infinite space state.</td>
<td>34</td>
</tr>
<tr>
<td>3.1</td>
<td>A model in its graphical representation</td>
<td>38</td>
</tr>
<tr>
<td>3.2</td>
<td>A model satisfying the formula $a$</td>
<td>39</td>
</tr>
<tr>
<td>3.3</td>
<td>A model satisfying the formula $AX a$</td>
<td>39</td>
</tr>
<tr>
<td>3.4</td>
<td>A model satisfying the formula $EF a$</td>
<td>40</td>
</tr>
<tr>
<td>3.5</td>
<td>A model satisfying the formula $E[a U b]$</td>
<td>41</td>
</tr>
<tr>
<td>3.6</td>
<td>Model in Figure 3.1, with labels on the states</td>
<td>44</td>
</tr>
<tr>
<td>3.7</td>
<td>Graph Grammar translation into a CTL model</td>
<td>45</td>
</tr>
<tr>
<td>3.8</td>
<td>The conversion from a simple state space to a model</td>
<td>46</td>
</tr>
<tr>
<td>4.1</td>
<td>The system architecture's block diagram</td>
<td>49</td>
</tr>
<tr>
<td>4.2</td>
<td>Grammar used in the experiments</td>
<td>61</td>
</tr>
<tr>
<td>4.3</td>
<td>Measurements for the experiment</td>
<td>63</td>
</tr>
</tbody>
</table>
# LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th></th>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The rewrite algorithm.</td>
<td>29</td>
</tr>
<tr>
<td>2</td>
<td>The SAT function.</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>The auxiliary function $\text{pre}_E$.</td>
<td>42</td>
</tr>
<tr>
<td>4</td>
<td>The auxiliary function $\text{pre}_A$.</td>
<td>42</td>
</tr>
<tr>
<td>5</td>
<td>The SAT$_{EX}$ function.</td>
<td>42</td>
</tr>
<tr>
<td>6</td>
<td>The SAT$_{AF}$ function.</td>
<td>43</td>
</tr>
<tr>
<td>7</td>
<td>The SAT$_{EU}$ function.</td>
<td>43</td>
</tr>
</tbody>
</table>
ACRONYMS

**CTL**  computational tree logic. 18, 31–33, 36, 42

**DPO**  double pushout. 19, 21, 24

**LTL**  Linear Temporal Logic. 32

**TV&V**  Test, Verification and Validation. 17
ABSTRACT

In this work, we present the construction of a tool for model checking graph grammars, VeriGraph. The verification is done by first executing a specification written as a graph grammar, resulting in a state space, and then checking properties of this specification with a CTL model checker. These functions are built with low coupling, and can be used in conjunction, as we do here, or reused for other implementations as needed. The tool focuses on flexibility, so it can be used to test new ideas about system simulation and verification techniques using graph grammars.

The graph grammar approach we use is the double pushout approach. We present a brief review of the theory, including the its main structures and algorithms for rewriting. Using double pushout as our approach poses some restrictions on the rewriting, which we take advantage to write a simple algorithm that is $O(VE)$ on the number of nodes $N$ and edges $E$ of the left hand side of a rule. We then use this structure to build a state space of the executed grammar.

The model checker uses a state-transition system based on the graph grammar state space to check for properties expressed in computational tree logic (CTL). We review CTL’s semantics and present an algorithm to perform the satisfaction verification of a CTL formula. The state-transition system is built by using the names of the production that can be applied to a state as the atomic propositions of that state. We discuss the limitations of this method and propose possible solutions to it.

The system is developed in Haskell. We review the relevant features of the language that we used when building this application. The algorithm and structure representation used in graph grammar and model checking are presented with code listings. We then present the integration and one experiment to give a notion of how the developed system performs.

We plan to continue the development of this tool, and we present what we plan to work on next, such as the implementation of second order graph grammar execution, critical pair analysis, and linear time logic implementation, and general improvements.

Keywords: Model Checking, Graph Grammars, Temporal Logic.
**RESUMO**

Neste trabalho é apresentada a construção de uma ferramenta para a verificação de modelos de gramáticas de grafos, VeriGraph. A verificação é feita a partir da execução de uma gramática de grafos, que resulta em um espaço de estados que é usado para verificar propriedades especificadas em CTL. A funcionalidades do sistema são construídas com baixa acoplamento, e podem ser usadas em conjunto, como aqui fazemos, ou isoladamente em outras implementações. A ferramenta é construída com foco em flexibilidade, para que possa ser usada para testar novas idéias sobre simulação verificação de sistemas usando gramáticas de grafos.

Nós usamos o método de reescrita de grafos baseado em *double pushout*, do qual apresentamos uma breve revisão da teoria, e algoritmos e estruturas usadas na reescrita. O uso de *double pushout* implica em limitações no processo de reescrita, e nós usamos estas limitações para construir um algoritmo de complexidade \( O(VE) \) do número de nodos \( N \) e arestas \( E \) no lado esquerdo da produção. Nós usamos estas estruturas para construir o espaço de estados da execução.

O verificador de modelos usa um sistema de transição de estados basedo no espaço de estados da gramática de grafos para executar a verificação de propriedades expressas como fórmulas em lógica de árvores computacionais (CTL). Nós revisamos a semântica do CTL, e apresentamos um algoritmo para executar a verificação de satisfação de uma fórmula. A construção do sistema de transição de estados é feita usando os nomes das regras que podem ser aplicadas a um dado estado como proposições atômicas deste estado. Este modelo tem limitações, que são discutidas junto com possíveis melhorias.

O sistema foi desenvolvido em Haskell. Nós revisamos as principais funcionalidades da linguagem que usamos na implementação. Os algoritmos e estruturas usados na implementação são apresentados junto com as listagens de código relevantes. Por fim, apresentamos a integração de ambos os módulos, e um experimento para dar a noção da performance do sistema.

É planejado continuar o desenvolvimento desta ferramenta, e nós apresentamos quais as funcionalidades que pretendemos desenvolver no futuro, como execução de gramáticas de grafos de segunda ordem, análise de par crítico, e implementação de novas formas de model checking, além de melhorias gerais.

**Keywords:** Verificação de modelos, Gramáticas de Grafos, Logicas Temporais.
1 INTRODUCTION

1.1 Motivation

There are a few tautologies in life. The famed “Death and Taxes” English saying first appeared in (DEFOE, 1840): “Things as certain as Death and Taxes, can be more firmly believ’d.” To those facts of life, we can certainly add a more recent development, Murphy’s Law (BLOCH, 2003). The infamous law states that it is the nature of “things” to go wrong. Large software projects are prone to failures, and we can find news of some catastrophic incidents, like the Ariane-5 satellite launcher, in which a illegal conversion from a 64-bit floating point number into a 16-bit integer number resulted in the explosion of the rocket, and the loss of a significant amount of money.

What Murphy’s Law fails to state is that it may take some time for “things” to fail, and pretty obvious errors can be easily avoided. It is our responsibility, as software engineers and architects, to ensure that our systems work properly for as long as possible, and when it fails, it does not threaten life, property and the public image of our company. Ad-hoc techniques, such as peer reviewing, can sometimes catch these failures, and can fail even when done judiciously. In the satellite launcher’s case, the failure would be an easy catch for an automated verification system.

The Verites project (RIBEIRO et al., 2014) aims to develop new techniques for Test, Verification and Validation (TV&V) of computational systems, and tools to support the developed techniques. One common technique for TV&V is the modeling of the system under verification as a graph grammar, and either checking for structural properties (static verification) without executing the grammar, or executing the grammar to simulate the system, and checking for properties of the system under execution (dynamic verification).

Tools that check for properties of systems modeled as graph grammars exist, but are limited. Some excel in static property verification, but lack or have limited support for dynamic property verification, or vice-versa; to make things worse, these tools are incompatible with the others, which forces the modeler to build more than one model if checking both static and dynamic properties of a grammar is required. This work lies within the scope of the Verites project, and aims at building a TV&V software from scratch that has the following objectives:

- Combine static verification and dynamic verification in a single program;
- Cerve as a test bed for new ideas in the field, in special, second-order graph grammars, a model recently developed to simulate software evolution (MACHADO,
Be as efficient as possible while having a simple and flexible implementation.

The last objective above prioritizes simplicity over efficiency slightly, to guarantee the second objective: it is better to have a code that can be changed easily but is a little inefficient, than to have complex and efficient code that does not allow for significant changes. Besides, it is possible to create a new iteration of the software that is more efficient once we understand what needs to be done in the first place.

The work described here is the work we’ve done over the last year; during this time, we developed several of the features needed to simulate the execution of a graph grammar. My work, exclusive to this monograph, is to develop a computational tree logic (CTL) model checker for VeriGraph. The work was done jointly with Ricardo Herdt; the division of work is described in Chapter 4.

1.2 Goals

The goal of my work in this context is to build the dynamic verification module of this system. This is done by a model checking algorithm that takes the execution of a grammar and a property expressed in temporal logic, and verifies if the property holds for the execution.

1.3 Contribution

The work executed contributes with a Haskell implementation of a tool for model checking graph grammars, and an algorithm for checking CTL formulas using as a model a simulated system that can be used independently of the tool.

1.4 Structure of this work

In Chapter 2 we present graph transformations and state space generation; in Chapter 3, we present the theory behind model checking and how to transform the state space of a graph grammar into a model for CTL model checking; in Chapter 4 we present the system implementation. In the conclusion, we discuss what we expect to build in the future. Appendix A contains the full code listing for the work developed.
2 GRAPH GRAMMARS

The study of graph transformations began in the 1960s, as a generalization on term rewriting. The idea of a graph transformation is one of rule-based transformation of graphs: system states are represented as graphs, and the behavior of the system is represented by rule applications. Rule applications are local – applied to a specific part of the graph –, allowing for more than one rule to be applied at the same time; this makes graph transformation an adequate model to represent systems with non-deterministic behavior.

In this chapter, we focus on the double pushout approach for graph transformation, among the many available in the literature. The chapter focuses on the intuition behind the technique, and we skip any theory that is not necessary for our objectives; if the reader is interested in the formalization of the approach, we recommend the reading of (EHRIG et al., 2010).

This chapter is organized in the following way: Section 2.1 contains the preliminary definitions for the rest of this chapter; Section 2.2 presents the double pushout (DPO) approach; Section 2.3 presents the algorithm for state space generation.

2.1 Preliminary definitions

A graph is a structure composed of a set of nodes connected to each other by edges.

Definition 1 (Graph). A graph is a tuple \( G = \langle V, E, s, t \rangle \) where \( V \) is the set of nodes, \( E \subseteq V \times V \) is the set of edges, and \( s, t : E \rightarrow V \) associate, respectively, the source and target of each edge.

Notation 1. We may refer to nodes and edges in a graph as elements, when no distinction is required. The sets of nodes \( V \) and edges \( E \) of a graph \( G \) are respectively refereed as \( V_G \) and \( E_G \). We may say that an element is in a graph \( e \in G \) if either \( e \in V_G \) or \( e \in E_G \).

Notation 2. We may apply set operations directly on graphs. When we do, it is assumed that the operation will be applied on both set \( V \) and \( E \) on both graphs, e.g. \( G - H \) will generate the graph \( G' = \langle V_G - V_H, E_G - E_H, s', t' \rangle \). It’s also possible to apply operations on graphs and sets; in this case, the operation will be applied to the appropriate set (\( V \) and/or \( E \)), and the appropriate set can be inferred from the context.

Example 1. Figure 2.1 depicts a graph and a possible visualization.
Figure 2.1: An example graph in its set form (left) and graphical form (right).

A graph morphism is a mapping of nodes and edges of a graph to, respectively, nodes and edges of another graph, preserving the structure of the input graph.

**Definition 2** (Graph morphism). Let $G = \langle V, E, s, t \rangle$ and $G' = \langle V', E', s', t' \rangle$ be graphs. A graph morphism $f : G \to G'$ is a pair of functions $\langle f_V, f_E \rangle$ with types $f_V : V \to V'$ and $f_E : E \to E'$ such that $s' \circ f_E = f_V \circ s$ and $t' \circ f_E = f_V \circ t$.

A morphism is said to be injective (surjective, bijective) if both $f_V$ and $f_E$ are injective (surjective, bijective). Graphs $G, H$ are *isomorphic* if there is a total bijective morphism (isomorphism) between them. We denote it by $G \cong G'$.

**Example 2.** Figure 2.2 depicts a graph morphism $f$ between graph $G$ and $G'$. Gray dashed lines are the node mappings, while gray solid lines map the edges.

Figure 2.2: An example of graph morphism.

An equivalence relation between two graphs can be defined by a graph isomorphism.

**Definition 3.** Graphs $G$ and $G'$ are equivalent when a morphism $m : G \to G'$ exists and is an isomorphism. An equivalence is depicted $G \cong_m G'$, or simply $G \cong G'$ if the specific morphism is omitted.
The collection of all graphs with all possible graph morphisms form the category \textbf{Graph}. It’s assumed that the reader has basic knowledge of category theory; otherwise, the user is referred to (HERRLICH; STRECKER, 1973). The DPO method is called an \textit{algebraic method}, since it relies on categories to provide the background theory.

It’s important to be able to categorize the elements of a graph according to what they represent. To achieve that, we use morphisms with a fixed target \(T\).

**Definition 4 (Graph typing).** A graph \(G\) typed over a graph type \(T\) is a pair \((G, t)\), where \(t : G \to T\) is a morphism, called typing morphism.

We can define the morphisms of typed graphs over \(T\).

**Definition 5 (Typed morphism).** A typed morphism over \(T\) is a graph morphism \(m : G \to G'\) such that the following diagram commutes.

![Diagram](image)

The category of all \(T\)-typed graphs and \(T\)-typed morphisms is called \(\textbf{T-Graph}\). The effect of adding typing to morphisms is that an element in the source graph can be mapped to an element in the destination graph only if they are of the same type. It is more intuitive and less cluttered to represent the types of each graph by the shape of the elements when representing a graph graphically, thus for the rest of the text, we will represent types on nodes by different shapes, and types on edges by black lines with different formats (solid, dashed, dotted…). Gray lines will be used to represent morphisms.

**Example 3.** Figure 2.3 depicts a typed morphism over \(T\), by depicting the full morphisms and the graphical notation: gray dashed lines show the specific mappings of each morphism.

2.2 The double pushout approach

This approach was first introduced in (EHRIG; PFENDER; SCHNEIDER, 1973), and was the first use of a categorial approach to graph rewriting. In this approach, \textit{productions} or \textit{rules} are represented by a pair of morphisms with the same origin.

**Definition 6 (Graph production).** A span is a pair of morphisms \((l, r)\) with the same source, as shown by the following diagram

\[
L \xleftarrow{l} K \xrightarrow{r} R
\]

A production is a span in \(\textbf{T-Graph}\), in which \(l\) and \(r\) are total. The collection of all productions is called \(\mathcal{P}\).
A production \( r : L \xleftarrow{l} K \xrightarrow{r} R \) describes how to modify a graph locally. The graph \( L \) is the pattern graph (or simply pattern) that must be found in the target graph \( G \), and the graph \( R \) (also known as replacement graph, or simply replacement) is the graph that replaces the pattern found, generating the graph \( H \). The graph \( K \), known as the interface graph, describes what is kept during the transformation. Both \( l \) and \( r \) are inclusions of \( K \) in the respective targets. Elements in \( L - K \) (in \( L \) but not in \( l(K) \)) are deleted during the rule application, while elements in \( R - K \) are created. Whenever possible, we represent the interface of a production by double pointed line between \( L \) and \( R \).

**Example 4.** The Figure 2.4 shows a graph rule that matches a circle and a square connected by an arrow, deletes the square and the arrow, and adds a new circle to the target graph. Above, a circle is connected to a square, and this structure must be found in the target graph. In the center, the invariant is the circle, both the square and the arrow are deleted during the rewrite. To the right, a second circle is created in the graph. We use the dashed gray arrow to denote the mappings of the morphisms \( l \) and \( r \) (Figure 2.4a). The shorthand representation of the same production (Figure 2.4b).

To apply a production \( p : L \xleftarrow{l} K \xrightarrow{r} R \) to a state graph \( G \), one must find a match of
$L$ into $G$, which is a morphism $m : L \to G$. The rewriting is defined as a double pushout diagram on $\text{T-Graph}$. The precise definition of a pushout is outside the scope of this text; the reader may refer to (HERRLICH; STRECKER, 1973, pp.138-150) to find it.

**Definition 7** (Graph derivation). Let $G$ be a graph in $\text{T-Graph}$, $p : L \leftarrow K \rightarrow R$ a production in $\text{T-Graph}$ and $m : L \to G$ a match. A graph derivation (or graph rewriting) from $G$ to $H$ exists if the diagram

\[
\begin{array}{ccccc}
L & l & K & r & R \\
\downarrow m & & \downarrow k & & \downarrow m' \\
G & l' & D & r' & H
\end{array}
\]

can be constructed in $\text{T-Graph}$, where $(l', m)$ is a pushout of $(l, k)$ and $(r', m')$ is a pushout of $(k, r)$.

A derivation from $G$ to $H$ with rule $p$ and match $m$ has the notation $G \xrightarrow{p,m} H$ in the following text, or simply $G \xrightarrow{p} H$, if the match is omitted.

**Example 5.** Figure 2.5 shows the derivation of graph $H$. The dashed gray lines represent the specific mapping of each morphism.
The first step to rewrite a graph is find a morphism $m : L \rightarrow G$, that matches the elements of $L$ with the elements of $G$, marking that region for rewriting. The match is calculated by solving the subgraph homomorphism problem for $L$ and $G$, or the subgraph isomorphism problem if the application requires an injective match. These problems are known to be intractable in the general case (MEHLHORN, 1984). Several solutions for these problems exist in the literature, and among them we chose to adapt the algorithm described in (RUDOLF, 2000), which finds all possible solutions with a constraint satisfaction algorithm. The algorithm itself and the adaptations made are outside the scope of this text.

The existence of a match does not guarantee that a derivation can be done, because there is the possibility of a match force the left diagram not to form a pushout. Two application conditions must be fulfilled for a rewrite to be possible in the DPO approach:

- **Identification condition**: a match $m : L \rightarrow G$ does not map a deleted element and a preserved element, or two deleted elements, in $L$ to the same element in $G$.

- **Dangling edge condition**: a match $m$ does not map a node $n$ to a node $m(n)$ that is connected to an edge outside the image of $m$.

Violations of such conditions are illustrated in Figure 2.6. The dotted elements show the errors in each case.
Figure 2.6: Examples of violations of (a) identification condition and (b) dangling condition.

Our algorithm for rewriting a graph is depicted in Algorithm 1.

```
input : a graph $G = \langle V_G, E_G, s_G, t_G \rangle$, a production $p : L \xleftarrow{l} K \xrightarrow{r} R$, and a match $m : L \rightarrow G$
output: the rewritten graph $H = \langle V_H, E_H, s_H, t_H \rangle$

1 $H \leftarrow G$
2 for $e \in m(L - l(K)) \mid e \in E_G$ do
3     if $e \notin E_H$ then
4         fail;
5     else
6         $H \leftarrow H - \{e\}$;
7     end
8 end
9 for $n \in m(L - K) \mid n \in V_G$ do
10    if ($\exists e \in E_H \mid s_H(e) = n \lor t_H(e) = n \lor (e \notin V_H)$) then
11        fail;
12    else
13        $H \leftarrow H - \{n\}$;
14    end
15 end
16 for $n \in k(K) \mid n \in G$ do
17    if $n \notin H$ then
18        fail;
19    end
20 end
21 for $n \in m'(R - r(K)) \mid n \in V_R$ do
22    $H \leftarrow H \cup \{n\}$;
23 end
24 for $e \in m'(R - r(K)) \mid e \in E_R$ do
25    $H \leftarrow H \cup \{e\}$;
26 end
27 return $H$
```

Algorithm 1: The rewrite algorithm.
The algorithm works by decomposing the rewriting into basic set operations: deletion, preservation and creation of elements. By ensuring that the deletions occur before the everything else (lines 2-15), we can check for violations of the identification condition (lines 16-20), and by ensuring that edges are deleted before nodes, we guarantee that we can check for violations of the dangling edge condition during the deletion of edges.

2.3 State space generation

Now that we laid out the basic theory behind algebraic graph rewriting, we can discuss system simulation and state space generation. We begin by the definition of a graph grammar.

Definition 8. A graph grammar is a tuple \( G = (G_0, P, \pi) \) where \( P \) is a set of production names, \( \pi : P \rightarrow \beta \) if a function that maps a name to its production and \( G_0 \) is the initial state (or initial graph) of the grammar.

Example 6. Figure 2.7 shows a complete graph grammar depicting a train system. The production moveTrain simulates the train movement between stations; embarkPassenger simulates the passenger in a station embarking the train in the station; disembarkPassenger simulates the passenger disembarking at a station. \( G_0 \) represents the system state at the beginning of the simulation. The lines between the passenger and the station represent his current location (solid line) and destination (dotted line).

To use a graph grammar as a computation abstraction, we model the behavior of the system in the productions, and the initial state of the system as \( G_0 \). The sequence of derivations \( G_n \xrightarrow{p,m} G_{n+1} \), generating new states \( G_{n+1} \) for the system, models the behavior of the system. At any state \( G_i \), any production \( p_j : L_j \xleftarrow{L_j} K_j \xrightarrow{K_j} R_j \) can be applied provided a match \( m_{ij} : L_j \rightarrow G_i \) exists and a derivation is possible (does not violate the application conditions). A production that fulfills these requirements is called a valid production.

Production applications on a system may generate graphs that are isomorphic to each other. To avoid representing isomorphic graphs more than once, it is useful to work with equivalence classes of graphs (derivations) instead of concrete graphs (concrete derivations).

Definition 9 (Equivalence class). An equivalence class of a set (collection) \( S \) is the quotient set (collection) of \( S \) under the equivalence relation \( \sim \), denoted \([a]\) for \( a \in S \), such that \([a] = \{ x \in S \mid a \sim x \}\)

Definition 10 (Rewrite equivalence). Two rewrites \( \delta_1 : G \xrightarrow{p} G' \) and \( \delta_2 : H \xrightarrow{q} H' \) are said equivalent if

- \( H \in [G]_\sim \)
- \( H' \in [G']_\sim \) and
- \( p = q \)
Such equivalence is denoted by $\delta_1 \equiv \delta_2$.

**Definition 11** (Equivalence class of graphs and derivations). An equivalence class of graphs is the quotient class of the class of all graphs under isomorphism. Such class is denoted $[G]_{\cong}$ in which $G$ is the canonical graph that represents the class.

An equivalence class of derivations is the quotient class of the class of all derivations under equivalence. Such class is denoted $[\delta]_{\equiv}$ where $\delta : G \xrightarrow{P} H$ is the canonical derivation that represents the class, and $G, H$ are canonical graphs.

**Notation 3.** For the remainder of this chapter, the equivalence classes of graphs are called **states** and the equivalence classes of derivations are called **transitions**.

A system can be simulated by sequentially applying a random valid production to the current state (generating a new state), and replacing the current state by the new state for the next iteration. This is interesting, since these simulations may reveal faulty transitions on the system, but due to the random selection of productions, this is not guaranteed to happen. A better way of verifying the system for faults is to iteratively build all possible future states from a given initial state, generating a **transition system**, or **state space**. A state space can be defined inductively, as follows.

**Definition 12** (State space). A state space of a graph grammar $\mathcal{G} = \langle G_0, P, \pi \rangle$ is a tuple $S = \langle S, T, \chi_0, \chi_1 \rangle$ where $S$ is a collection of states, $T \subseteq S \times P \times S$ is a collection of transitions labeled over $P$, and $\chi_0, \chi_1 : T \to S$ are functions that map the source and target of a transition, respectively. $S$ and $T$ are defined inductively as:
• $[G_0] \subseteq S$

• for each $[G] \subseteq S$, if $\delta : G \xrightarrow{p} G'$, then $[G] \subseteq S$ and $[\delta] \subseteq T$

**Example 7.** Figure 2.8 depicts the full state space generated by the graph grammar in Example 6. It contains 53 states.

It is clear from this definition above that a transition system forms a graph like structure, with states acting as nodes and transitions acting as edges. Creating an algorithm from this definition is very straightforward.

The state space permits the visualization of all possible computations that may take place in a system in execution, which is very useful for checking for faulty transitions. In small systems, this inspection can be done manually, but in large systems (with transitions in low tens and above), it would be better to have an automated method to check whether the system behaves correctly. For that purpose, we may employ a *model checker*, a tool that allows us to specify the expected system behavior and check it against the actual modeled behavior by inspecting the possible sequences of transitions against the specification.

A state space is possibly infinite, i.e., it is always possible to build a new state for the system. One example is a graph grammar that in which a node is matched, and another is added to the state, as depicted in Figure 2.9. Although the theory of graph grammars allows for infinite transition systems, we may set a limit on the number of iterations used to build a *bounded* state space, which poses limitations on the capacity of verification of such systems. We will discuss this on Chapter 3.

### 2.4 Final remarks

In this chapter, we have reviewed the basic theory and intuition on graph grammars. In special, we have seen how a graph grammar generates a state space, which we will use in Chapter 3 to build a model to be checked.
Figure 2.8: The full state space generated by the execution of the grammar in Figure 2.7
Figure 2.9: An example of a grammar that generates an infinite space state.
3 MODEL CHECKING

As stated in Section 2.3, manually checking transition systems with many transitions by eye inspection can be hard, and automation can be used to verify such systems. This automation is implemented in a model checker. A model checker performs the verification of properties of a given system model by using a formal framework. Model checking is largely discussed in the literature, where several methods and case studies are presented. This chapter treats exclusively of the verification of systems modeled as graph grammars, using CTL, a type of temporal logic. For a thorough discussion on model checking, we suggest the reading of (CLARKE; GRUMBERG; PELED, 1999; KATOEN; BAIER, 2008).

Model checking can be used for formally verifying properties of a system, among them safety – “something bad never happens” – and liveness – “something good will eventually happen” – (LAMPORT, 1977). For example, the question “Will trains never collide?” is a safety question in regards to the Example 6, while the question “Will all the passengers arrive at their destinations?” is a liveness question for the same example1.

In this chapter, we initially present the idea behind a model checker on Section 3.1. We then present model checking using CTL in Section 3.2, and present the syntax, the model and semantics. Lastly, we discuss how to use a graph grammar state space for verifying CTL properties in Section 3.3.

3.1 Model checking

According to (KATOEN; BAIER, 2008, Section 1.2):

Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.

Model checking is a verification technique that explores all possible system states in a systematic manner, to verify that the system satisfies some property. The model is generated by system simulation, and the property is defined in an appropriate temporal logic. In our work, the simulation is generated by the state space of a graph grammar, and the logic used is CTL. This chapter discusses CTL and the use of a state space as a system model.

1The system shown in Figure 2.7 has only one train (so it will never collide with itself), but we can replace the initial state with one with more trains.
3.2 Computational Tree Logic

Since the work of Aristotle we can see logicians dabbling with the idea of formalizing time in logic. Little development was done until Arthur Prior’s *Time and Modality* (PRIOR, 1957) that formalized time with a type of modal logic, with two time operators: “some time in the future” and “some time in the past”. In his work, Prior presents the time as a linear sequence of events, which was further expanded to allow for branching time (alternations in the unfolding of events), after a letter from Saul Kripke. In formal verification, the first system proposed by Prior has an equivalent in Linear Temporal Logic (LTL) (PNUELI, 1977), that suffers from a similar limitation: time is considered linear, and if branching time is present, all alternate sequences of events must validate the formula under verification. CTL (EMERSON; HALPERN, 1985) appears in this context as a more complete alternative to LTL, since it allows for the quantification over the temporal operators. To compare both in plain English, LTL allows for the verification of the statement “In the future, cars will fly”, but not the statement “It is possible that, in the future, cars will fly”, since the second is existentially quantified: there is the possibility of some future in which cars were replaced by jet-packs for personal transportation, and research in flying cars will halt; thus cars will never fly in this alternate future.

A formula in CTL is defined by the following syntax.

**Definition 13 (Syntax of CTL).** The language of well-formed formulas for CTL is generated by the grammar

\[ \varphi ::= \bot \mid \top \mid p \mid \neg \varphi \mid \varphi \land \varphi \mid \varphi \lor \varphi \mid \varphi \Rightarrow \varphi \mid \varphi \Leftrightarrow \varphi \mid (\varphi) \mid AX\varphi \mid AF\varphi \mid AG\varphi \mid A[\varphi U \varphi] \mid EX\varphi \mid EF\varphi \mid EG\varphi \mid E[\varphi U \varphi] \]

where \( p \in P \) and \( P \) is a set of atomic formulas.

**Example 8.** The formulas \( AF \ AX \ a \), \( A[EF \ a \ U \ AX \ b] \) and \( AF \ AG \ a \) are well-formed CTL formulas. The formula \( A[Xv \ U \ b] \) is not a well-formed CTL formula, since \( X \) must be preceded by either \( A \) or \( E \).

Operators \( \neg, \land, \lor, \Rightarrow \) and \( \Leftrightarrow \) work as the same operators in classical logic, and are resolved in the same way. The remaining operators \( (AX, AF, AG, AU, EX, EF, EG, EU) \) are called temporal operators. We will discuss their meaning after we define CTL semantics, in Definition 17.

Not all operators in Definition 13 above are needed to implement the complete semantics of CTL; for instance, the formula \( p \land q \) can be converted to \( \neg(\neg p \lor \neg q) \) by De Morgan’s Law, reducing the set of implemented operators by one. In Section 3.2.1 we discuss one of the possible reductions to a minimal set of operators. The length of a formula is the number of symbols it contains.

**Definition 14.** The length of a formula is defined by structural induction over \( \varphi' \) as

\[
\text{length}(\varphi' \in \{T, \bot\} \cup P) = 1 \\
\text{length}(\varphi' \in \{\neg \varphi, AX\varphi, AF\varphi, AG\varphi\}) = 1 + \text{length}(\varphi) \\
\text{length}(\varphi' \in \{EX\varphi, EF\varphi, EG\varphi, (\varphi)\}) = 1 + \text{length}(\varphi) \\
\text{length}(\varphi' \in \{\varphi_1 \land \varphi_2, \varphi_1 \lor \varphi_2, \varphi_1 \Rightarrow \varphi_2\}) = 1 + \text{length}(\varphi_1) + \text{length}(\varphi_2) \\
\text{length}(\varphi' \in \{\varphi_1 \Leftrightarrow \varphi_2, A[\varphi_1 U \varphi_2], E[\varphi_1 U \varphi_2]\}) = 1 + \text{length}(\varphi_1) + \text{length}(\varphi_2)
\]
Example 9. The lengths of the well-formed CTL formulas in Example 8 are 3, 5 and 3, respectively.

The set of subformulas $\Psi$ of a CTL formula $\varphi$, is defined by induction over the structure of the CTL grammar.

**Definition 15 (CTL Subformulas).** The set of subformulas $\Psi$ of a CTL formula $\varphi'$ is defined as

$\Psi(\varphi' \in \{\top, \bot\} \cup P) = \{\varphi'\}$

$\Psi(\varphi' \in \{\neg \varphi, (\varphi), AX \varphi, AF \varphi, AG \varphi\}) = \{\varphi'\} \cup \Psi(\varphi)$

$\Psi(\varphi' \in \{EX \varphi, EF \varphi, EG \varphi\}) = \{\varphi'\} \cup \Psi(\varphi)$

$\Psi(\varphi' \in \{\varphi_1 \land \varphi_2, \varphi_1 \lor \varphi_2, \varphi_1 \Rightarrow \varphi_2\}) = \{\varphi'\} \cup \Psi(\varphi_1) \cup \Psi(\varphi_2)$

$\Psi(\varphi' \in \{\varphi_1 \Leftrightarrow \varphi_2, A[\varphi_1 U \varphi_2], E[\varphi_1 U \varphi_2]\}) = \{\varphi'\} \cup \Psi(\varphi_1) \cup \Psi(\varphi_2)$

Example 10. The subformulas of the well-formed formulas in Example 8 are

- $\Psi(AF AX a) = \{a, AX a, AF AX a\}$
- $\Psi(A[EF a U AX b]) = \{b, AX b, a, EF a, A[EF a U AX b]\}$
- $\Psi(AF AG a) = \{a, AG a, AF AG a\}$

A CTL formula is interpreted over a transition system.

**Definition 16 (Transition System).** A transition system (or model) is a triple $M = \langle S, \rightarrow, L \rangle$ where

- $S$ is a set of states
- $\rightarrow \subseteq S \times S$ is a transition relation;
- $L : S \rightarrow 2^P$ is a function that maps a state to its labels

Example 11. Figure 3.1 shows a model in its graphical representation. Squares represent states, arrows represent the transitions and the letters separated by inside the states represent the atomic formulas satisfied in each state.

A transition system represents the flow of time of the system, represented by the transition relation. In a system with branching (out degree of a state $\geq 2$), the system represents possible sequences of events. What we see in the definition of transition system is that they form another graph-like structure, with states as nodes, and $\rightarrow$ builds the set of edges.

A path in $M$ is a (possibly infinite) sequence of states $\pi = [s_0, s_1, s_2, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots]$ such that $s_i \rightarrow s_{i+1}$ for all $i \geq 0$. We say that a path starts in $s$ if $s_0 = s$. We say that $s'$ is in the future of $s$ if $s'$ is in a path that starts in $s$.

With the syntax and model, when can define the semantics of each operator.
Definition 17 (Semantics of CTL). The satisfaction of a formula by a given model $M$ and state $s$, written $M, s \models \varphi$, is defined as

- $(M, s \models T), (M, s \not\models \bot)$
- $(M, s \models p)$ iff $p \in L(s)$
- $(M, s \models \varphi_1 \land \varphi_2)$ iff $(M, s \models \varphi_1)$ and $(M, s \models \varphi_2)$
- $(M, s \models \varphi_1 \lor \varphi_2)$ iff $(M, s \models \varphi_1)$ or $(M, s \models \varphi_2)$
- $(M, s \models \varphi_1 \Rightarrow \varphi_2)$ iff $(M, s \models \neg \varphi_1 \lor \varphi_2)$
- $(M, s \models \varphi_1 \Leftrightarrow \varphi_2)$ iff $(M, s \models \varphi_1 \Rightarrow \varphi_2)$ and $(M, s \models \varphi_2 \Rightarrow \varphi_1)$
- $(M, s \models (\varphi))$ iff $(M, s \models \varphi)$
- $(M, s \models AX\varphi)$ iff for all $s'$ such that $s \rightarrow s'$, $(M, s' \models \varphi)$
- $(M, s \models AF\varphi)$ iff for all paths starting in $s$, there is an $i \geq 0$ such that $(M, s_i \models \varphi)$
- $(M, s \models AG\varphi)$ iff for all paths starting in $s$ and for all $i \geq 0$, $(M, s_i \models \varphi)$
- $(M, s \models A[\varphi_1 U \varphi_2])$ iff for all paths starting in $s$ there is a $j \geq 0$ such that $(M, s_j \models \varphi_2)$ and for all $0 \leq i < j$ $(M, s_i \models \varphi_1)$
- $(M, s \models EX\varphi)$ iff exists $s'$ such that $s \rightarrow s'$, $(M, s' \models \varphi)$
- $(M, s \models EF\varphi)$ iff exists a path starting in $s$, there is an $i \geq 0$ such that $(M, s_i \models \varphi)$
- $(M, s \models EG\varphi)$ iff exists a path starting in $s$ and for all $i \geq 0$, $(M, s_i \models \varphi)$
- $(M, s \models E[\varphi_1 U \varphi_2])$ iff exists a path starting in $s$ there is a $j \geq 0$ such that $(M, s_j \models \varphi_2)$ and for all $0 \leq i < j$ $(M, s_i \models \varphi_1)$
For a formula $\varphi$ with $A$ to hold on a model, all paths starting at the initial state must satisfy $\varphi$. For $E$, only one path need to satisfy $\varphi$.

The temporal operators check whether the formula $\varphi$ holds in a path. The operator $X$ checks if the next state of a given state satisfies $\varphi$. $F$ checks if some state in the future of the initial state satisfies $\varphi$. $G$ checks whether all states in the initial state’s future satisfy $\varphi$ holds. $U$ verifies if, in a path, all states satisfy $\varphi_1$ until $\varphi_2$ is satisfied.

**Example 12.** Figure 3.2 shows a model that satisfies $a$. Figure 3.3 show a model satisfying $AX a$. Figure 3.4 shows a model satisfying $EF a$. Figure 3.5 shows a model satisfying $E[a U b]$. The initial state is the gray state; dashed lines represent many similar paths and the dotted lines represent that the paths starting in the source state might continue.

![Figure 3.2: A model satisfying the formula $a$.](image)

**Example 13.** We would like to express the question “Will users embark the train?” in CTL, based in our state space. In Section 3.3, we establish that the name of the rules applied to a given state in the state space correspond to the atoms in the model. Thus, the question may be rephrased as “Will users eventually embark the train?”, which can be expressed as $(M, s_0 \models EF \text{embarkPassenger})$. Other examples:

1. “will the train enter a loop?”: if the train is not embarking and disembarking passengers; $(M, s_0 \models EG(\neg \text{embarkPassenger} \land \neg \text{disembarkPassenger}))$.

2. “Will passengers disembark the train?”: no one wants to be stuck forever in a train; $(M, s_0 \models EF(\text{embarkPassenger} \Rightarrow AF\text{disembarkPassenger}))$.

Formula 1 holds for the state space generated by the grammar in Figure 2.7. Formula 1 does not.
3.2.1 Important equivalences for CTL

Some CTL operators can be written in terms of another operators, like $AG \varphi \equiv \neg EF \neg \varphi$ (for all paths, and all states in those paths, $\varphi$ holds is equivalent to that “there’s not a future in which $\varphi$ is not satisfied”). First we define the equivalence of formulas.

**Definition 18.** Two CTL formulas $\varphi$ and $\phi$ are said equivalent ($\varphi \equiv \phi$) if for all models $M$ and any initial state $s$ in $M$, $(M, s \models \varphi)$ iff $(M, s \models \phi)$.

Several of those equivalences can be found in the literature, so we will only list them in this section. Proofs for the equivalences can be found in (HUTH; RYAN, 2000).

- $\neg AF \varphi \equiv EG \neg \varphi$
- $\neg EF \varphi \equiv AG \neg \varphi$
- $\neg AX \varphi \equiv EX \neg \varphi$
- $AX \varphi \equiv \neg EX \neg \varphi$
- $EG \varphi \equiv \neg AF \neg \varphi$
- $AF \varphi \equiv A[\top U \varphi]$
- $EF \varphi \equiv E[\top U \varphi]$
- $AG \varphi \equiv \neg EF \neg \varphi$
- $A[\varphi U \phi] \equiv \neg (E[\neg \phi U (\neg \varphi \land \neg \phi)] \lor EG \neg \phi)$

These equivalences are very useful to build a minimal core of operators. In fact, it has been proved that, to express any formula in CTL, the only operators required are at least one of $\{AX, EX\}$, at least one of $\{EG, AF, AU\}$, at least one of $\{\land, \lor\}$, $\neg$ and $EU$ (MARTIN, 2002). From the definitions above, we can define a minimal core $\{\top, \land, \neg, EX, EU, AF\}$.
3.2.2 CTL satisfaction algorithm

With these definitions, it’s possible to write the CTL satisfaction algorithm. The algorithm follows one described in (HUTH; RYAN, 2000, pp. 224–229), which we transcribe here.

The CTL satisfaction algorithm \( SAT \) works by labeling each state with all possible subformulas of an input CTL formula \( \varphi \) that the state satisfies. The formula is satisfied in the model if the initial state is labeled with \( \varphi \). The actual labeling is done by, given a set of states \( S' \subseteq S \), excluding from it all elements that don’t satisfy the input formula. If subformulas are evaluated in ascending size order, we benefit from having all subformulas of an input formula already solved, so we are able to reuse them.

The algorithm is presented in Algorithm 2, and uses auxiliary function in Algorithms 3–7.

Example 14. We want to know if the model in Figure 3.6 (which is the same model as the one in Figure 3.1, with labels added in the states) satisfies the Formula \( a \Rightarrow EX b \) in the state 0.

The resolution of \( SAT(\mathcal{M}, a \Rightarrow AX b) \) is

\[
SAT(\mathcal{M}, a \Rightarrow AX b) = SAT(\mathcal{M}, \neg a \lor AX b) = SAT(\mathcal{M}, \neg a) \cup SAT(\mathcal{M}, AX b) = (S - SAT(\mathcal{M}, a)) \cup SAT(\mathcal{M}, \neg EX \neg b) = (S - SAT(\mathcal{M}, a)) \cup (S - SAT(\mathcal{M}, EX \neg b)) = (S - SAT(\mathcal{M}, a)) \cup (S - SAT_{EX}(\neg b, \mathcal{M}))
\]

The result of \( SAT(\mathcal{M}, a) \) is the set \( \{s_0\} \). The result of \( S - SAT(\mathcal{M}, a) = \{s_0, s_1, s_3, s_4, s_5\} \) \( \{s_0\} = \{s_1, s_3, s_4, s_5\} \). The function \( SAT_{EX}(\neg b, \mathcal{M}) \) works by first finding the set of states that satisfy \( \neg b \) (Algorithm 5, line 3), then finding set of states that are antecessors of nodes in \( X \) (Algorithm 5, line 5). The solution for \( SAT(\mathcal{M}, \neg b) \) is

\[
SAT(\mathcal{M}, \neg b) = S - SAT(\mathcal{M}, b) = \{s_0, s_1, s_3, s_4, s_5\} - \{s_1, s_3\} = \{s_0, s_4, s_5\}
\]
Algorithm 2: The SAT function.

1 function SAT (φ, M)  
   input : a formula φ, and a state space M = ⟨S, →, L⟩  
   output: a set of states that satisfy φ  
switch φ do  
   case ⊤ return S;  
   case ⊥ return ∅;  
   case p return {s ∈ S | φ ∈ L(s)};  
   case ¬φ return SAT(φ, M);  
   case φ1 ∧ φ2 return SAT(φ1, M) ∩ SAT(φ2, M);  
   case φ1 ∨ φ2 return SAT(φ1, M) ∪ SAT(φ2, M);  
   case φ1 ⇒ φ2 return SAT(¬φ1 ∨ φ2, M);  
   case φ1 ⇔ φ2 return SAT(φ1 ⇒ φ2 ∧ φ2 ⇒ φ1, M);  
   case AXφ return SAT(¬EX¬φ, M);  
   case AFφ return SAT(AFφ, M);  
   case AGφ return SAT(¬EF¬φ, M);  
   case A[ϕ1 U ϕ2] return SAT(¬(E[¬ϕ2 U (¬ϕ1 ∧ ¬ϕ2)] ∨ EG¬ϕ2), M);  
endsw
Algorithm 3: The auxiliary function pre_E.

1 function pre_E (Y, M)  
   input : set of states Y, and a state space M = ⟨S, →, L⟩  
   output: A set of states that are antecessors of the states in Y  
return {s ∈ S | exists s', (s → s' and s' ∈ Y)}
Algorithm 4: The auxiliary function pre_A.

1 function SAT_EX (φ, M)  
   input : a formula φ, and a state space M = ⟨S, →, L⟩  
   output: a set of states that satisfy φ  
X := SAT(φ, M);  
Y := pre_E(X, M);  
return Y
Algorithm 5: The SAT_EX function.
function \( \text{SAT}_{AF} (\phi, \mathcal{M}) \)

\[ \text{input} : \text{a formula } \phi, \text{ and a state space } \mathcal{M} = \langle S, \rightarrow, L \rangle \]

\[ \text{output} : \text{a set of states that satisfy } AF\phi \]

1. \( X := S \);
2. \( Y := \text{SAT}(\phi, \mathcal{M}) \);
3. while \( X \neq Y \) do
   4. \( X := Y ; \)
   5. \( Y := Y \cup \text{pre}_A(Y, \mathcal{M}) ; \)
4. end
5. return \( Y \)

Algorithm 6: The \( \text{SAT}_{AF} \) function.

function \( \text{SAT}_{EU} (\phi_1, \phi_2, \mathcal{M}) \)

\[ \text{input} : \text{a formula } \phi, \text{ and a state space } \mathcal{M} = \langle S, \rightarrow, L \rangle \]

\[ \text{output} : \text{a set of states that satisfy } \phi \]

1. \( W := \text{SAT}(\phi_1, \mathcal{M}) \);
2. \( X := S ; \)
3. \( Y := \text{SAT}(\phi_2, \mathcal{M}) ; \)
4. while \( X \neq Y \) do
   5. \( X := Y ; \)
   6. \( Y := Y \cup (W \cap \text{pre}_E(Y)) ; \)
5. end
6. return \( Y \)

Algorithm 7: The \( \text{SAT}_{EU} \) function.
Then we solve $\text{SAT}_{\text{EX}}(\neg b, \mathcal{M}) = \{s_1, s_3, s_4\}$. The final solution is given by

$$\text{SAT}(\mathcal{M}, a \Rightarrow AX b) = (S - \text{SAT}(\mathcal{M}, a)) \cup (S - \text{SAT}_{\text{EX}}(\neg b, \mathcal{M}))$$

$$= (S - \{s_0\}) \cup (S - \{s_1, s_3, s_4\})$$

$$= \{s_1, s_3, s_4, s_5\} \cup \{s_0, s_5\} = \{s_0, s_1, s_3, s_4, s_5\}$$

Since $s_0$ is in the set of states that satisfy $a \Rightarrow AX b$, the formula is satisfied.

### 3.3 Model checking graph grammars

The missing piece to model check a graph grammar is how to transform the graph grammar state space (Definition 12) into a CTL (Definition 16). The main structural difference between both is that a graph grammar state space is labeled in the transitions, while the CTL model is labeled in the states, therefore a translation step is required.

**Definition 19** (Translation from state space to a model). To convert a state space $\mathcal{S} = \langle S, T, \chi_0, \chi_1 \rangle$ to a model $\mathcal{M} = \langle S, \rightarrow, L \rangle$, we define the following translations:

- $S_\mathcal{M} = S_{\mathcal{S}}$
- for all $[G] \models \frac{\text{p}}{\text{p}} \triangleright [G'] \models \in T$, $[G] \models \rightarrow [G'] \models \in \rightarrow$
- for all $[G] \models \frac{\text{p}}{\text{p}} \triangleright [G'] \models \in T$, $p \in L([G] \models)$

**Example 15.** Figure 3.7 shows the model generated by the state space in Figure 3.7a, according to the rules in Definition 19. The state space generation was cut after 5 iterations, and the nodes without atoms were cut from the CTL model.

The schema above defines that a state in the state space is a state in the model, and that transitions in the state space define the transition relation in the model. The labeling of nodes is done by the name of the productions whose patterns can be found in the source
(a) The limited state space generated by the graph grammar in Figure 2.7.

(b) CTL model generated by the state space on Example 3.7a.

Figure 3.7: Graph Grammar translation into a CTL model.
state of the transition, i.e., if a production can be applied to a state, the production’s name is an atomic proposition in the model.

We are interested in checking two types of specifications in a graph grammar: the sequence of transformations it can have and states that have a certain internal conformation (conditions). The first is already verifiable by the schema above, with some limitations (see below). To find conditions, we can use a production that does not change the graph \( (c : K \xleftarrow{t} K \xrightarrow{t} K) \), where \( t \) is a graph isomorphism). This way, only a match of the condition is performed, and the state is tagged with the condition’s name for the model checker.

As observed in Section 2.3, the state space of a graph grammar can be infinite (unbounded number of states), but for a model checker, models need to be finite to check for properties. In case of a grammar that has an infinite space state – like the one presented in Figure 2.9 –, or very large grammars in general, we can limit the state space generation, but this only allows the verification of properties that represent deviations from the system specification. Also, in case we don’t find such deviations, we can not assert that such deviations will not occur in the future, since we did not verify the future states of the system. Other technique that can be applied is bounded model checking (CLARKE et al., 2001).

The translation schema in Definition 19 is very similar to the one used in Groove (RENSINK, 2008). One issue with this schema is that it is not possible to guarantee the sequentiality of computations, i.e. a formula \( \varphi \Rightarrow EX\phi \) can hold for a model even if \( \varphi \) and \( \phi \) are never executed in sequence. For a concrete example, see Figure 3.8, which satisfies the formula \( r_1 \Rightarrow EXr_2 \), but \( r_2 \) does not happen after \( r_1 \), but after \( r_0 \). This happens because the atomic proposition are pulled from the transition (in the graph grammar state space) to its source state (in the state-transition system), forgetting which transition in the state-transition system is generated by the production application of interest. There are ways to work around this problem: we can use only conditions as atomic formulas, avoiding this situation by always looking at the internal node conformation. This approach is readily available in the system, but is not complete, since it does not allow for the verification of computation sequences. The development of a logic that “follow paths” – considers the path labels instead or in conjunction with the node labels – requires further study.
3.4 Final Remarks

This chapter we reviewed the concepts of formal verification, and one of the systems used in it, CTL, and discussed a schema for translating a graph grammar state space into a model. We then discussed some of the limitations of such schema, and of graph grammars as formal methods of verification in regards to resources consumed, presenting some alternatives to attenuate these limitations. Some of these alternatives may be unfit to the system in development, as the objectives of the project prioritize simplicity and flexibility, so solutions that create complexity must be avoided.
4 IMPLEMENTATION

In this chapter, we will see how we implemented the structures presented in the previous two chapters. We start with the choice of language in Section 4.1, followed by the implementation of graph transformation in Section 4.2 and CTL model checking in Section 4.3. The two subsystems were developed independently from each other, and we look at the integration of both in Section 4.4. Lastly, we look at some some experiments to measure the execution time and memory usage of the system in Section 4.5.

VeriGraph was developed in conjunction with Ricardo Herdt. My individual contribution in this work is the architecture of the system, and the design and implementation of the rewriting and state space modules, and the implementation of the CTL satisfaction algorithm. Here we also present some modules that are not developed by myself, but developed jointly with Ricardo, or fully by him. In Figure 4.1 we have a depiction of the system architecture’s block diagram and contributions: I contributed fully with modules in dark gray, partially with modules in light gray, and Ricardo’s individual contributions are in white.

4.1 Haskell

VeriGraph is implemented in Haskell (JONES, 2003). Haskell is a pure functional language that uses the Hindley-Milner type system (MILNER, 1978) with type classes. It’s assumed that the user has some knowledge of Haskell, otherwise we recommend (LIPOVAČA, 2011) as a good introductory text.

Haskell is a language that was first developed for academic purposes, but met some success in the industry. It has a strong type system, which helps in development since once the types are checked to be correct, the implementation is most likely correct. The

Figure 4.1: The system architecture’s block diagram
The language also uses *call-by-need* semantics (lazy evaluation), in which a value is only computed when it is needed; in comparison, languages like OCaml have strict evaluation, in which the value is computed immediately. The advantage of lazy evaluation is that “dead” computations are never executed, but computing values takes more time. Performance is still good, hitting at most 5 times the time of a comparable C implementation in the Computer Language Benchmarks Game (http://benchmarksgame.alioth.debian.org/).

Since Haskell is a pure language, it does not allow for global mutable state, or any kind of non-determinism in the results of each function, which includes input and output. To overcome this limitation, Haskell provides a language feature called *monads*, which we see in Section 4.1.2.

In this section, we discuss the features of the language that are used in VeriGraph.

### 4.1.1 Type classes

Type classes (HALL et al., 1996) were introduced in Haskell to provide overloading of arithmetic operations. This is achieved by adding constraints to functions, which can be only be used in conjunction with members of a given type class. It has been used with success for implementing other language features such as Monads, because it can be used in a way very similar to interfaces in object oriented languages. For instance, the `Eq` type class is shown in Listing 4.1, which provides value comparison, and instances.

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

instance Eq Int where
  x == y = intEq x y
  x /= y = not (x == y)

instance Eq Float where
  x == y = floatEq x y
  x /= y = not (x == y)

Listing 4.1: The `Eq` type class
```

Here, we suppose that `intEq` and `floatEq` return `True` if the values provided are equal. We can then use type classes to provide polymorphic functions like `elem`, which checks whether a value is in a list, in Listing 4.2.

```haskell
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem x (y:ys) = x == y || 'elem' ys

Listing 4.2: The `elem` function
```

The declaration before `=>` defines that the polymorphic type `a` must be a member of the type class `Eq`, which allows the `elem` function to be used only to compare values of types that have a defined equality. If the reader is proficient with Java, this is very similar to what the interface `Comparable` does. Type classes are used in the implementation of monads.
4.1.2 Monads

Monads were first introduced as a way of modeling certain types of effects in catego-
rical semantics (MOGGI, 1991), such as exceptions, state, and input/output. Wadler
(WADLER, 1993) then used these concepts as a way to implement such features in pure
functional languages.

A monad, in Haskell, is a type class with four associated functions, as seen in Listing
4.3.

class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  fail :: String -> m a

Listing 4.3: TheMonad type class

The return function takes a value and returns the same value inside the monad. The
operator >>= takes a monadic value and a computation that returns a monadic value, and
returns the value of the computation; this operation is called “bind”. The operator >> is
usually defined as v >> v' = v >>= \_ -> v', which executes the first computation,
discards its return value and then executes the second computation. fail serves to report
an error. The implementation of the Maybe monad is provided in Listing 4.4.

data Maybe a = Just a | Nothing

instance Monad Maybe where
  return x = Just x
  (Just x) >>= f = f x
  Nothing >>= _ = Nothing
  fail m = Nothing

Listing 4.4: Maybe monad implementation

Monads allow for programming in Haskell in imperative style, with the do notation.
We see the do notation in Listing 4.16.

Some standard monads:

- **Maybe**: A type that can return either a value in the type or a null-like value. It can
  be used to represent errors.

- **State s a = State { runState :: s -> (a, s)}**: Represents a local shared
  state s. To modify the state in the monad, two functions are provided: get ::
s returns the state stored in the monad, and put :: s -> () replaces the current
state with a new one. Working with the state monad usually revolves around get-
ing the current state, manipulating it and putting it back in the monad with put.
runState executes the monad with a initial state. We use the state monad in the
code that builds the state space.

- **IO a**: The monad that runs all the input/output methods in Haskell. The IO monad
  has no run function.
4.1.3 Parser Combinators

Parser combinators (HUTTON, 1992) are functions that accept parsers as inputs and return a new parser as output. A parser is a function that accepts strings as input and returns some structure as the output.

Parsec (LEIJEN; MEIJER, 2001) is one framework for monadic combinator parsing, probably the most well-known and successful for Haskell. Its success inspired the development of parser combinators in other languages. A good introduction to Parsec can be found in (O’SULLIVAN; GOERZEN; STEWART, 2008, Ch. 16). An example of a parser combinator is the `<>` operator in Parsec, which takes two parsers, `parserA` and `parserB`, and returns a parser `parserA <> parserB` that returns the result of `parserA` if it parsed the string successfully, otherwise returns the result of `parserB`. Parsec allows for backtracking parsing with the operator `try`, that restores the input’s state to what it was before the parser executed in case of a parsing failure.

4.2 Implementation of Graph Rewriting

In this section, we describe the main algorithms and data types used in the graph transformation subsystem of VeriGraph, whose theory was described in Chapter 2.

4.2.1 Graphs

We start by looking at the graph structure in Listing 4.5. Full code listing for the `GraphGrammar.Graph` module is presented in Appendix A.2.

```haskell
data Node a = Node Int Int a deriving (Show, Read)
data Edge b = Edge Int (Int, Int) Int b deriving (Show, Read)

instance Eq (Node a) where
  Node lid _ _ == Node gid _ _ = lid == gid

instance Eq (Edge b) where
  Edge lid _ _ _ == Edge gid _ _ _ = lid == gid

data Digraph a b = Digraph (IntMap (Node a)) (IntMap (Edge b))
  deriving (Show, Read, Eq)

Listing 4.5: Graph representation
```

A graph is defined as a set of nodes and a set of edges. The elements of a graph are represented by the data types `Node` and `Edge`. Nodes and edges are parameterized, so they can carry extra information, like labels, weights and other data; we’ll refer to this data as the element’s `payload`. Since graph grammar state spaces are graphs, we use this structure to represent them as well. Besides that, nodes are parameterized by two `Int`, one is the node’s identity and the other is the node’s type, and edges are parameterized by identity, source, target and type.

The `IntMap` datatype is a map type from `Int` to its contents. The identity of an element is used as the key in the map. It is important to keep the keys consistent with the corresponding element identities. It is easier to guarantee this if we use accessor functions, as depicted in Listing 4.6. `IntMap` is preferable over lists in this implementation due to the
efficient implementation of this structure. The structure also shares several functions with
lists, such as fold, map and filter. The biggest advantage of lists is that we don’t need to
guarantee the consistency between the element’s key and its identity.

```
addNode :: (Monad m) => Node a -> Digraph a b -> m (Digraph a b)
addNode n@(Node id _ _) g@(Digraph nm em) =
  if id 'IM.member' nm
  then fail $ "addNode: node " ++ show id ++ " already in digraph"
  else return $ Digraph (IM.insert n nm) em

addEdge :: (Monad m) => Edge b -> Digraph a b -> m (Digraph a b)
addEdge e@(Edge id (s, t) _ _) g@(Digraph nm em) |
  id 'IM.member' em =
    fail $ "addEdge: edge " ++ show id ++ " already in digraph"
  | s 'IM.member' nm && t 'IM.member' nm =
    return $ Digraph nm (IM.insert id e em)
  | otherwise =
    fail $ "addEdge: edge points to nodes not found in digraph"

removeNode :: (Monad m) => Node a -> Digraph a b -> m (Digraph a b)
removeNode n@(Node id _ _) g@(Digraph nm em) |
  id 'IM.notMember' nm =
    fail $ "removeNode: node " ++ show id ++ " not in digraph"
  | IM.fold
    (
      (Edge eid (s, t) _ _) acc -> acc || s == id || t == id)
    False em =
    fail $ "removeNode: node " ++ show id ++ " has some edge
    pointing to it"
  | otherwise =
    return $ Digraph (IM.delete id nm) em

removeEdge :: (Monad m) => Edge b -> Digraph a b -> m (Digraph a b)
removeEdge e@(Edge eid _ _ _) g@(Digraph nm em) |
  id 'IM.member' em =
    then return $ Digraph nm (IM.delete eid em)
  else fail $ "removeEdge: edge " ++ show eid ++ " not in digraph"

keepNode :: (Monad m) => Node a -> Digraph a b -> m (Digraph a b)
keepNode (Node nid _ _) g@(Digraph ns es) =
  if nid 'IM.member' ns
    then return g
  else fail $ "keepNode: node " ++ show nid ++ " does not exist"

keepEdge :: (Monad m) => Edge b -> Digraph a b -> m (Digraph a b)
keepEdge (Edge eid _ _ _) g@(Digraph ns es) =
  if eid 'IM.member' es
    then return g
  else fail $ "keepEdge: edge " ++ show eid ++ " does not exist"
```

Listing 4.6: Graph accessor functions

In Listing 4.6 we see monadic actions being used. The functions removeNode, removeEdge,
keepNode and keepEdge check if the target element exists in the graph, and fail in case it
does not. Also, removeNode checks if the target node is connected to an edge, and fail in
case it is. Ordering the application of these functions as described in Algorithm 1 allows
us to detect violations of the (DPO) application conditions.

---

1The order is removeEdge, removeNode, keepNode, keepEdge, addNode, addEdge.
A typed graph is a pair of graphs, as in Listing 4.7.

```haskell
data TypedDigraph a b = TypedDigraph (Digraph a b) (TGraph a b)
deriving (Show, Read, Eq)
```

Listing 4.7: Typed graph representation

\(TGraph\) is a type synonym for \(\text{Digraph}\), used to differentiate the type graph. To encode the type morphism, we store the identity of the type in the typed element.

Graph elements have a type class to unify access to the common features of elements: payload, identity and type. The \(\text{GraphElement}\) type class and the two instances are presented in Listing 4.8.

```haskell
class GraphElement a where
type Payload a :: *
payload :: a -> Payload a
elemId :: a -> Int
typeId :: a -> Int

instance GraphElement (Node a) where
type Payload (Node a) = a
payload (Node _ _ q) = q
elemId (Node i _ _) = i
typeId (Node _ t _) = t

instance GraphElement (Edge a) where
type Payload (Edge a) = a
payload (Edge _ _ _ q) = q
elemId (Edge i _ _ _) = i
typeId (Edge _ _ t _) = t
```

Listing 4.8: \(\text{GraphElement}\) definition and instances

We see two features being applied in the above listing: type classes and type families. Type families is a language extension that can be best described as “type level functions”, since it allows to replace a type family with a concrete type associated with some other type, avoiding some errors related to the monomorphism restriction introduced in Haskell’s type system. For more on type families, we suggest the reading of (SCHRIJVERS et al., 2008) (a gentle introduction can be found at (KISELYOV; JONES; SHAN, 2010)). In this case, type families are used to limit the output type of the payload function; otherwise the function type would be \(a \rightarrow b\), which is an invalid type in the Hindley-Milner type system.

A graph morphism is depicted in Listing 4.9.

```haskell
type Morphism = ([(Int, Int)], [(Int, Int)])
```

Listing 4.9: The representation of a morphism

A morphism is a pair of lists of pairs. The first list represent the node mappings, and the second one the mapping of edges, both referring to the identities of the elements in the graph.

The module also contains some helper functions, that can be checked in Appendix A.2.
4.2.2 Rules

The rule (production) representation is depicted in Listing 4.10.

```haskell
import Data.Map (delete)

-- Rule representation
rule :: Tree a b

-- Function to extract graphs
left :: (Eq a, Eq b) => Rule a b -> [a] -> [b]
right :: (Eq a, Eq b) => Rule a b -> [b] -> [a]
```

Similar to morphisms, productions \((p: L \xrightarrow{l} K \xrightarrow{r} R)\) are composed of two lists of mappings. We encode the set \(l(K) - K\) of elements that are deleted \(r(K) - K\) of the elements that are created) as mappings that contain Nothing on the right (left). \(K\) is the set of actions that does not contain Nothing on either side. The functions in Listing 4.11 allow us to extract the graphs \(L, R\) and \(K\). Function left returns \(L\), right returns \(R\) and glue returns \(K\).

```haskell
-- Auxiliary functions for Rule
addAction (Nothing, Just t) = True
addAction _ = False
removeAction (Just s, Nothing) = True
removeAction _ = False
```

For rewriting, we need to extract the actions as functions, therefore, we also define the functions to extract it, in Listing 4.12.

```haskell
-- Functions to extract actions
addAction (Nothing, Just t) = True
addAction _ = False
removeAction (Just s, Nothing) = True
removeAction _ = False
```
keepAction (Just s, Just t) = True
keepAction _ = False

nodeAction :: (Monad m, Eq a) => NodeAction a -> (Digraph a b -> m (Digraph a b))
nodeAction (Nothing, Just n) = addNode n
nodeAction (Just n, Nothing) = removeNode n
nodeAction (Just n, Just n') = if n /= n'
  then const $ fail "Node transformation is unhandled"
  else keepNode n
nodeAction (Nothing, Nothing) = return

dgeAction :: (Monad m, Eq b) => EdgeAction b -> (Digraph a b -> m (Digraph a b))
dgeAction (Nothing, Just e) = addEdge e
dgeAction (Just e, Nothing) = removeEdge e
dgeAction (Just e, Just e') = if e /= e'
  then const $ fail "Edge transformation is unhandled"
  else keepEdge e
dgeAction (Nothing, Nothing) = return

actionSet :: (Monad m, Eq a, Eq b) => Rule a b -> [Digraph a b -> m (Digraph a b)]
actionSet (Rule n mt na ea) = let
  nodeActions f = map nodeAction . filter f
  edgeActions f = map edgeAction . filter f
  knSet = nodeActions keepAction na
  keSet = edgeActions keepAction ea
  anSet = nodeActions addAction na
  aeSet = edgeActions addAction ea
  dnSet = nodeActions removeAction na
  deSet = edgeActions removeAction ea
  in deSet ++ dnSet ++ knSet ++ keSet ++ anSet ++ aeSet

Listing 4.12: Function extraction for rules

Functions addAction, removeAction and keepAction serve to filter the actions that add, remove and keep nodes, respectively. nodeAction and edgeAction translate an action to the appropriate function seen in Listing 4.6. actionSet perform the conversion from a production into a list of functions, ordered to catch violations of the application conditions.

The production application is done by the applyRule function in Listing 4.13.

applyActions :: Monad m => [a -> m a] -> a -> m a
applyActions as g = foldM (\g f -> f g) g as

applyRule :: (Monad m, Eq a, Eq b) => Rule a b -> TypedDigraph a b -> m (TypedDigraph a b)
applyRule m tg = let (TypedDigraph g t) = tg
  actions = actionSet m
  in do g' <- applyActions actions g
       return $ TypedDigraph g' t

Listing 4.13: Rule application to a typed graph

\(^2\)To recap: delete edges, delete nodes, keep nodes, keep edges, create nodes and finally create edges.
The reason for it was presented in Section 2.2
**Function** *applyActions* executes a fold inside the monad, taking an initial graph and a list of actions, and applies the functions of the list to the current graph; the graph for the first application is the initial graph and the graph for the following applications is the graph generated by the previous application. *applyRule* takes a production and a typed graph, and extracts the functions to be applied, and uses *applyActions* to apply them to the initial graph.

### 4.2.3 Rewriting

During rewriting, we would like to avoid identity collision in the graph, in which we create a node with an id that is already in use, overriding the old element. The rewrite function handles it, as presented in Listing 4.14.

```haskell
rewrite :: (Monad m, Eq a, Eq b) => Rule a b -> TypedDigraph a b -> Morphism -> m (TypedDigraph a b)
rewire rule tGraph match = let (TypedDigraph graph _) = tGraph
ns = renameStart $ nodes graph
es = renameStart $ edges graph
renamedRule = rename ns es rule match
in applyRule renamedRule tGraph

renameStart :: GraphElement e => [e] -> Int
renameStart es = 1 + (maximum $ map elemId es)

lookup’ i = maybe i id . lookup i

renameNode :: [(Int, Int)] -> Node a -> Node a
renameNode nm (Node id t p) = Node (lookup’ id nm) t p

renameEdge nm em (Edge id st t p) = Edge (lookup’ id em)
(double (\x -> lookup’ x nm) st) t p

renameAction :: GraphElement e => (e -> e) -> Action e -> Action e
renameAction f = double (liftM f)

double :: (a -> b) -> (a, a) -> (b, b)
double f (x, y) = (f x, f y)

rename :: (Eq a, Eq b) => Int -> Int -> Rule a b -> Morphism -> Rule a b
rename ns es (Rule n mt nr er) (nm, em) = Rule n mt
(map nodeRename nr) (map edgeRename er)

where

  elemIds :: (Eq e, GraphElement e) => [e] -> [Int]
  elemIds = map elemId

  addElements :: (Eq e, GraphElement e) => [Action e] -> [e]
  addElements = map (fromJust . snd) . filter ((== Nothing) . fst)

  idMap :: (Eq e, GraphElement e) => [Action e] -> [Int]
  idMap xs ys = zip (elemIds $ addElements xs) ys
```
nodeIdMap = (idMap nr [ns..]) ++ nm
edgeIdMap = (idMap er [es..]) ++ em

nodeRename = renameAction (renameNode nodeIdMap)
edgeRename = renameAction (renameEdge nodeIdMap
edgeIdMap)

Listing 4.14: The rewrite function, and auxiliary functions

The rename function handles the renaming of the rule. It takes two numbers, one to
start the renaming of nodes, the other to start the renaming of the edges. These two values
are determined by the renameStart function, that returns the largest identifier in the set,
incremented by 1. With the rule elements renamed, the applyRule function in the last
section is called to run the rewrite.

4.2.4 Graph matching

The graph matching algorithm works by generating a sequence of functions that check
whether a given element in the mapped graphs does not violates the structural properties of
a morphism, as explained in Definition 2. The function signatures are in Listing 4.15 The
matching algorithm takes into account the type of matching we want to use: a Normal
match is an subgraph morphism, a Mono match is a subgraph isomorphism, an Epi match
is a graph morphism and an Iso match is a graph isomorphism. One increment that
we did in the algorithm, is to include information from the rule to avoid violating the
application conditions. We also used the type information from the grammar to limit the
number of possible matches by removing from the possible matches for each element in
L the elements in G that do not have the same type.

findMatches :: D.MatchType -> D.TypedDigraph a b -> D.TypedDigraph a b -> [D.Morphism]
findMatchesR :: D.Rule a b -> D.MatchType -> D.TypedDigraph a b -> DTypedDigraph a b -> [D.Morphism]

Listing 4.15: Graph matching functions

The time and memory measurements for the algorithm with and without these condi-
tions are presented in section 4.5.

4.2.5 State space generation

The state space is generated by the function runStateSpace in Listing 4.16.

runStateSpace :: (Eq a, Eq b, MonadIO m) => Int -> TypedDigraph a b -> [Rule a b] -> m (StateSpace a b)
ruseStateSpace n g r = buildGraph \do i <- createNode g 1
mkStateSpace n i g r
mkStateSpace :: (Eq a, Eq b, MonadIO m) => Int -> Int -> TypedDigraph a b -> [Rule a b] -> m (StateSpace a b)
mkStateSpace n i g r = do
forM_ r $ \r' ->
do let (Rule _ mt _ _) = r'
forM_ (findMatchesR r' mt (left r' $ tpg g) g) $ \m -> do
  t <- rewrite r' g m

mkStateSpace 0 _ _ _ = return ()

Listing 4.16: State space generation
The function receives the iteration limit, the initial graph and a list of rules. The state space creation works by depth first, finding all matches to all rules in the grammar and applying the rewrite. If the rewrite fails (due to violations of the application conditions), it is not added to the state. If it succeed, the state and the transition are added to the state space by the putState function. This function takes care of checking for if the state being put is isomorphic to another state. In such case, the function only adds the new transition, and returns 0. Otherwise, the new state is added, and the identity of the new node is returned. If a new identity is returned, mkStateSpace calls itself recursively with the generated graph, reducing the iteration count. When the iteration count reaches zero, the iteration stops. buildGraph will execute the State monad, and is defined in the GraphGrammar.Builder.Graph module on Appendix A.9.

4.2.6 Other modules

4.2.6.1 Grammar serialization

To be able to read a grammar file, we use the Show and Read type classes. Show provides a way to transform a data structure into a String, and when derived implicitly (using the deriving) keyword) will print the structure of the data. Read provides a way of reading a String into a data structure, and when defined implicitly will read the string provided by Show. Currently, the system reads the format provided Show that is stored in a file. The serialization and unserialization of this data is handled in the module GraphGrammar.Serialized in Appendix A.7.

4.2.6.2 Graph modeling language output

To be able to see the states, state space and model, we use the graph modeling language (GML) file format, that can be opened in several graph visualization tools. The GML generation is handled by the GraphGrammar.GML module on Appendix A.8.

4.3 CTL implementation

The CTL implementation is composed of the model, syntax semantics, parser and translation. The graph grammar state space and CTL model figures in this text were generated by running the grammar in VeriGraph and opening the files for visualization and layouting in the yEd graph editor (WIESE; EIGLSPERGER; KAUFMANN, 2004). The theory for this implementation is presented in Chapter 3.

4.3.1 Model

The model for our CTL implementation is a graph. To keep implementation independent from the graph grammar, we decided to build another graph instance, that is similar
to the one previously seen. With this decision, we opened up the possibility of separating this subsystem from the graph rewriting implementation. The implementation can be seen in the module Logic.Modal.Graph in Appendix A.11.

4.3.2 Parser

The syntax representation of CTL is defined in the module Logic.CTL.Base in Appendix A.13.

```
data CTL = Literal Bool  -- true, false
  | Atom String            -- p
  | Not CTL                -- ¬p
  | And CTL CTL            -- f && f
  | Or CTL CTL             -- f || f
  | Implies CTL CTL        -- f → f
  | AllNext CTL            -- AX f
  | SomeNext CTL           -- EX f
  | AllFuture CTL          -- AF f
  | SomeFuture CTL         -- EF f
  | AllGlobal CTL          -- AG f
  | SomeGlobal CTL         -- EG f
  | AllUntil CTL CTL       -- A[f U f]
  | SomeUntil CTL CTL      -- E[f U f]
  deriving (Show, Eq, Read)
```

Literal contains a boolean literal, Atom contains an atomic proposition. Not, And, Or and Implies operators represent the logic operators ¬, ∧, ∨ and →, respectively. The All* constructors represent the A* operator in CTL and the Some* operators, the E* operators. *Next, *Future *Global and *Until represent the operators *X, *F, *G and *U, respectively. To the right of each constructor, we see the implemented syntax of the operators.

The parser was developed with Parsec. For the implementation, the syntax in Definition 13 must be factored as a right recursive grammar, below.

```
ϕ₁ ::= A[ϕ₁ U ϕ₁] | E[ϕ₁ U ϕ₁] | ϕ₂ → ϕ₁ | ϕ₂
ϕ₂ ::= ϕ₃ && ϕ₂ | ϕ₃ && ϕ₂ | ϕ₃
ϕ₃ ::= ¬ ϕ₃ | AX ϕ₃ | AF ϕ₃ | AG ϕ₃ | EX ϕ₃ | EF ϕ₃ | EG ϕ₃ | (ϕ₁) | p
```

The parser implementation is in Appendix A.14.

4.3.3 Semantics

The CTL semantics implementation is a translation of the Algorithms 2–7, as we can see in Appendix A.15. The ctlSat is the implementation of the SAT function in Algorithm 2, and uses pattern matching to mimic the definition of the algorithm.

4.4 Integration

In this section, we talk about the integration of the subsystems for graph rewriting and CTL satisfaction to form the model checker. The integration is done in the main
program, which is shown in Appendix A.17. The function reads the grammar from the file indicated in the command line in the command line, and generates the state space from that grammar. It then translates it into a CTL model with the function translate (Appendix A.16. Rules are entered in the command line also, between quotes. Each formula is parsed, and then checked in the generated model.

4.5 Experiments

For our experiments, we defined an grammar with infinite state space that is a little more complex than the grammar presented in Figure 2.9; the grammar is depicted in Figure 4.2. The grammar is one of the worst case situations for matching, since the type is the simplest possible, not allowing the algorithm to cut many branches of the search tree early.

The experiment was designed to measure how the number of elements of the graph affect the performance of the system in the worst case. A normal use case will usually execute faster. All rules are set to Normal matches. The was run by varying the depth of the state space from 10 to 100 in steps of 10, and repeated three times. We took measures of the time used to execute and the peak memory usage during execution. The experiments were done in a Intel Core i3-2120 running at 3.3 GHz, with 8GB of memory. The operating system is Fedora 20 64 bit.

The mean time and memory usage are presented in the table below, and plotted in Figure 4.3.
<table>
<thead>
<tr>
<th>Iteration size</th>
<th>time (s)</th>
<th>memory (kb)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rule</td>
<td>no rule</td>
</tr>
<tr>
<td>10</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>20</td>
<td>0.16</td>
<td>0.14</td>
</tr>
<tr>
<td>30</td>
<td>0.71</td>
<td>0.73</td>
</tr>
<tr>
<td>40</td>
<td>2.42</td>
<td>2.62</td>
</tr>
<tr>
<td>50</td>
<td>6.24</td>
<td>6.46</td>
</tr>
<tr>
<td>60</td>
<td>13.98</td>
<td>14.32</td>
</tr>
<tr>
<td>70</td>
<td>27.95</td>
<td>28.74</td>
</tr>
<tr>
<td>80</td>
<td>51.24</td>
<td>52.78</td>
</tr>
<tr>
<td>90</td>
<td>91.14</td>
<td>96.74</td>
</tr>
<tr>
<td>100</td>
<td>151.38</td>
<td>156.5</td>
</tr>
</tbody>
</table>

From the results, we conclude that more the rule information affects the execution time and memory usage slightly, as we expected. We would need to run experiments with more complex grammars to reveal how this modification results in faster execution of the grammar. Comparisson with other tools was not done.

### 4.6 Final remarks

In this chapter, we saw how VeriGraph was implemented. VeriGraph is now undergoing a major remodeling to be closer to the theory presented in Chapter 2. The CTL implementation is isolated from the graph grammar code, and can be used with the new system. One of the major limitations of the system is the graph grammar input format. Ideally, we want to enter the grammar as either a standard graph grammar language or visually. The project will, at some point, be headed into the direction of loading the graph grammar file formats from other tools, like GROOVE. Visual inputs are more complex, since graphical interfaces are not a strong point of Haskell, but are still feasible with ports of GTK for Haskell.
Figure 4.3: Measurements for the experiment
5 CONCLUSIONS AND FUTURE WORK

In this monograph, we reviewed graph grammars and CTL model checking, and de-cried the development of VeriGraph, a CTL model checker for typed graph grammars. The tool was developed in Haskell, and the implementation favored correctness and architectural flexibility rather than fast execution. Despite its steep learning curve, choosing Haskell allowed us to keep the implementation close to the DPO graph transformation approach.

Some interesting aspects of our implementation lie in the rewriting algorithm, that converts the rewrite into a set of atomic graph transformation actions, and the use of rule information to improve the matching algorithm. Both take advantage of the limitations implied by the application conditions of the DPO approach to avoid violating such conditions. We have also found some issues with the translation from a graph grammar state space to a CTL model, which we discuss in Section 3.3.

We conclude this work wanting to release this tool as soon as some features in Section 5.1 are implemented.

5.1 Future work

To improve VeriGraph, we would like to implement the following features:

- Implementation of negative application conditions;
- Implementation of second-order graph grammars;
- $CTL^*$ model checker to unify the backends;
- Model checking for second order graph grammars;
- Critical pair analysis for first- and second-order graph grammars;
- Loading of grammars from well known file formats used in other tools;
- Single pushout and sesqui pushout transformation;
- Matching algorithm with GPU acceleration (CHAKRAVARTY et al., 2011);
- User interface for graph and production edition.
REFERENCES


LIPOVÁČA, M. Learn You a Haskell for Great Good!: a beginner’s guide. [S.l.]: no starch press, 2011.

MACHADO, R. Higher-Order Graph Rewriting Systems. 2012. Tese (Doutorado em Ciência da Computação) — Instituto de Informática - UFRGS.


Appendix A

A.1 GraphGrammar

module GraphGrammar ( module GraphGrammar.Graph
, module GraphGrammar.StateSpace
, module GraphGrammar.Rule
, runSerializedGrammar ) where

import Control.Monad.IO.Class

import GraphGrammar.Graph
import GraphGrammar.Rule
import GraphGrammar.Match
import GraphGrammar.Transformation
import GraphGrammar.StateSpace
import GraphGrammar.Serialized

unserializeString :: MonadIO m => String -> m (Serialized String String)
unserializeString s = do system <- liftIO $ readFile s
return $ unserialize system

runSerializedGrammar :: MonadIO m => Int -> String -> m (StateSpace String String)
runSerializedGrammar l s = do (Serialized graphs rules) <-
unserializeString s
runStateSpace l (head graphs) rules

A.2 GraphGrammar.Graph

{-# LANGUAGE TypeFamilies #-)
module GraphGrammar.Graph
( Edge (..)
, Node (..)
, Digraph (..)
, TypedDigraph (..)
, GraphElement (..)
, Morphism (..)
, TGraph
, empty
, null
, fromLists
, node


, edge
, addNode
, addEdge
, removeNode
, removeEdge
, keepNode
, keepEdge
, insNode
, insEdge
, delNode
, delEdge
, nodes
, edges
, findNode
, source
, target
, hasEdge
, sourceID
, targetID
, srcType
, tarType
) where

import Control.Monad

import Prelude hiding (null)

import Data.IntMap (IntMap)

import qualified Data.IntMap as IM

import qualified Data.List as L

import Assorted.PrettyPrint

data Edge a = Edge
  Int
  (Int
   , Int
   , Int
   )
  a
  deriving (Show, Read)

instance Eq (Edge a) where
  Edge lid _ _ _ == Edge gid _ _ _ = lid == gid

data Node a = Node
  Int
  Int
  a
  deriving (Show, Read)

instance Eq (Node a) where
  Node lid _ _ == Node gid _ _ = lid == gid

class GraphElement a where
  type Payload a :: *
  payload :: a -> Payload a
  elemId :: a -> Int
  typeId :: a -> Int

instance GraphElement (Node a) where
  type Payload (Node a) = a
  payload (Node _ _ q) = q
  elemId (Node i _) = i
  typeId (Node _ t _) = t

instance GraphElement (Edge a) where
  type Payload (Edge a) = a
  payload (Edge _ _ _ q) = q
  elemId (Edge i _ _) = i
  typeId (Edge _ _ t _) = t
data Digraph a b = Digraph (IntMap (Node a)) (IntMap (Edge b))
                     deriving (Show, Read, Eq)

data TypedDigraph a b = TypedDigraph (Digraph a b) (TGraph a b)
                     deriving (Show, Read, Eq)

type TGraph a b = Digraph a b

empty :: Digraph a b
empty = Digraph (IM.empty) (IM.empty)

null :: Digraph a b -> Bool
null (Digraph nm em) =
    IM.null nm && IM.null em

fromLists :: [Node a] -> [Edge b] -> Digraph a b
fromLists ns es = Digraph (fromElementList ns) (fromElementList es)

fromElementList :: GraphElement a => [a] -> IntMap a
fromElementList = IM.fromList . map (\x -> (elemId x, x))

node :: Int -> Digraph a b -> Maybe (Node a)
node i (Digraph n _) = IM.lookup i n

edge :: Int -> Digraph a b -> Maybe (Edge b)
edge i (Digraph _ e) = IM.lookup i e

addNode :: (Monad m) => Node a -> Digraph a b -> m (Digraph a b)
addNode n@(Node id _ _) g@(Digraph nm em) =
    if id `IM.member` nm
        then fail $ "addNode: node " ++ show id ++ " already in digraph"
        else return $ Digraph (IM.insert id n nm) em

addEdge :: (Monad m) => Edge b -> Digraph a b -> m (Digraph a b)
addEdge e@(Edge id (s, t) _ _) g@(Digraph nm em) =
    | id `IM.member` em =
        fail $ "addEdge: edge " ++ show id ++ " already in digraph"
        | s `IM.member` nm && t `IM.member` nm =
            return $ Digraph nm (IM.insert id e em)
    | otherwise =
        fail $ "addEdge: edge points to nodes not found in digraph"

removeNode :: (Monad m) => Node a -> Digraph a b -> m (Digraph a b)
removeNode n@(Node id _ _) g@(Digraph nm em) =
    | id `IM.notMember` nm =
        fail $ "removeNode: node " ++ show id ++ " not in digraph"
        | IM.fold
            \(Edge eid (s, t) _ _) acc -> acc || s == id || t == id)
            False em =
        fail $ "removeNode: node " ++ show id ++ " has some edge pointing to it"
    | otherwise =

return \$ Digraph (IM.delete id nm) em

removeEdge :: (Monad m) \rightarrow Edge b \rightarrow Digraph a b \rightarrow m (Digraph a b)
removeEdge e@(Edge id _ _) g@(Digraph nm em) =
  if id 'IM.member' em
  then return \$ Digraph nm (IM.delete id em)
  else fail \$ "removeEdge: edge " ++ show id ++ " not in digraph"

keepNode :: (Monad m) \rightarrow Node a \rightarrow Digraph a b \rightarrow m (Digraph a b)
keepNode (Node nid _ _) g@(Digraph ns es) =
  if nid 'IM.member' ns
  then return g
  else fail \$ "keepNode: node " ++ show nid ++ " doesn't exist"

keepEdge :: (Monad m) \rightarrow Edge b \rightarrow Digraph a b \rightarrow m (Digraph a b)
keepEdge (Edge eid _ _ _) g@(Digraph ns es) =
  if eid 'IM.member' es
  then return g
  else fail \$ "keepEdge: edge " ++ show eid ++ " doesn't exist"

findNode :: Int \rightarrow Digraph a b \rightarrow Maybe (Node a)
findNode id (Digraph nm _) =
  IM.lookup id nm

nodes :: Digraph a b \rightarrow [(Node a)]
nodes (Digraph nm _) = map snd $ IM.toList nm

edges :: Digraph a b \rightarrow [(Edge b)]
edges (Digraph _ em) = map snd $ IM.toList em

source :: Edge b \rightarrow Digraph a b \rightarrow Node a
source e d =
  let (Just n) = findNode (sourceID e) d
  in n

target :: Edge b \rightarrow Digraph a b \rightarrow Node a
target e d =
  let (Just n) = findNode (targetID e) d
  in n

sourceID :: Edge b \rightarrow Int
sourceID (Edge _ (src, _) _ _) = src

targetID :: Edge b \rightarrow Int
targetID (Edge _ (_, tar) _ _) = tar

hasEdge :: TypedDigraph a b \rightarrow Node a \rightarrow Bool
hasEdge (TypedDigraph dg _) n =
  let
    nid = elemId n
    found = L.find \((Edge _ (s, t) _ _) \rightarrow s == nid ||
      t == nid) $ edges dg
  in case found of
    Just _ -> True
    Nothing -> False
nodePayload :: Node a -> a
nodePayload (Node _ _ p) = p

insNode :: Node a -> Digraph a b -> Digraph a b
insNode n@(Node id _ _) g@(Digraph nm em) =
  if id 'IM.member' nm
    then g
    else Digraph (IM.insert id n nm) em

insEdge :: Edge b -> Digraph a b -> Digraph a b
insEdge e@(Edge id (s, t) _ _) g@(Digraph nm em) =
  if id 'IM.member' em
    then g
    else s 'IM.member' nm && t 'IM.member' nm =
      Digraph nm (IM.insert id e em)
  otherwise =
    g

delNode :: Node a -> Digraph a b -> Digraph a b
delNode n@(Node id _ _) g@(Digraph nm em) =
  if id 'IM.notMember' nm
    then g
    else IM.fold (
      (Edge eid (s, t) _ _) acc -> acc || s == id || t == id)
        False em
    else
      Digraph (IM.delete id nm) em

delEdge :: Edge b -> Digraph a b -> Digraph a b
delEdge e@(Edge id _ _ _) g@(Digraph nm em) =
  if id 'IM.member' em
    then Digraph nm (IM.delete id em)
    else g

findNodeType :: Int -> TypedDigraph a b -> Maybe Int
findNodeType id td@(TypedDigraph (Digraph nm em) _) =
  let n = IM.lookup id nm
      in case n of
    Nothing -> Nothing
    Just (Node _ tid _) -> Just tid

srcType :: Edge b -> TypedDigraph a b -> Maybe Int
srcType (Edge _ (s, _) _ _) l =
  findNodeType s l

tarType :: Edge b -> TypedDigraph a b -> Maybe Int
tarType (Edge _ (_, t) _ _) l =
  findNodeType t l

type Morphism = (((Int, Int)), [(Int, Int)])
import Control.Monad
import Data.Maybe
import qualified Data.List as L
import Data.IntMap hiding (map, filter)
import GraphGrammar.Graph
import Assorted.PrettyPrint

type Action a = (Maybe a, Maybe a)
type NodeAction a = Action (Node a)
type EdgeAction a = Action (Edge a)
data MatchType = Normal | Mono | Epi | Iso
  deriving (Show, Read, Eq)
data Rule a b = Rule String MatchType [NodeAction a] [EdgeAction b]
  deriving (Show, Read)
emptyRule = Rule [] Normal [] []
addNodeAction :: Node a -> Node a -> Rule a b -> Rule a b
  addNodeAction ln rn m@(Rule n mt nal eal) =
    if (Just ln, Just rn) `L.notElem` nal
      then Rule n mt ((Just ln, Just rn) : nal) eal
      else m
addEdgeAction :: Edge b -> Edge b -> Rule a b -> Rule a b
  addEdgeAction le re m@(Rule n mt nal eal) =
    Rule n mt nal ((Just le, Just re) : eal)
nodeAction :: (Monad m, Eq a) => NodeAction a -> (Digraph a b -> m (Digraph a b))
nodeAction (Nothing, Just n) = addNode n
nodeAction (Just n, Nothing) = removeNode n
nodeAction (Just n, Just n') = if n /= n'
  then const $ fail "Node transformation is unhandled"
  else keepNode n
nodeAction (Nothing, Nothing) = return
edgeAction :: (Monad m, Eq b) => EdgeAction b -> (Digraph a b -> m (Digraph a b))
edgeAction (Nothing, Just e) = addEdge e
edgeAction (Just e, Nothing) = removeEdge e
edgeAction (Just e, Just e') = if e /= e'
  then const $ fail "Edge transformation is unhandled"
  else keepEdge e
edgeAction (Nothing, Nothing) = return
addAction (Nothing, Just t) = True
addAction _ = False
removeAction (Just s, Nothing) = True
removeAction _ = False
keepAction (Just s, Just t) = True
keepAction _ = False

left :: (Eq a, Eq b) => Rule a b -> TGraph a b -> TypedDigraph a b
left (Rule n mt nr er) t = let
    f e = fst e /= Nothing
    ns = toElemList fst $ filter f nr
    es = toElemList fst $ filter f er
    in TypedDigraph (fromLists ns es) t

right :: (Eq a, Eq b) => Rule a b -> TGraph a b -> TypedDigraph a b
right (Rule n mt nr er) t = let
    f e = snd e /= Nothing
    ns = toElemList snd $ filter f nr
    es = toElemList snd $ filter f er
    in TypedDigraph (fromLists ns es) t

glue :: (Eq a, Eq b) => Rule a b -> TGraph a b -> TypedDigraph a b
glue (Rule n mt nr er) t = let
    f e = snd e == fst e
    ns = toElemList fst $ filter f nr
    es = toElemList fst $ filter f er
    in TypedDigraph (fromLists ns es) t

toElemList :: (Action a -> Maybe a) -> [Action a] -> [a]
toElemList f = map (fromJust . f)

actionSet :: (Monad m, Eq a, Eq b) => Rule a b -> [Digraph a b -> m (Digraph a b)]
actionSet (Rule n mt na ea) = let
    nodeActions f = map nodeAction . filter f
    edgeActions f = map edgeAction . filter f
    knSet = nodeActions keepAction na
    keSet = edgeActions keepAction ea
    anSet = nodeActions addAction na
    aeSet = edgeActions addAction ea
    dnSet = nodeActions removeAction na
    deSet = edgeActions removeAction ea
    in deSet ++ dnSet ++ knSet ++ keSet ++ anSet ++ aeSet

applyActions :: Monad m => [a -> m a] -> a -> m a
applyActions as g = foldM (\g f -> f g) g as

applyRule :: (Monad m, Eq a, Eq b) => Rule a b -> TypedDigraph a b -> m (TypedDigraph a b)
applyRule m tg = let (TypedDigraph g t) = tg
                 actions = actionSet m
                 in do g' <- applyActions actions g
                      return $ TypedDigraph g' t

A.4 GraphGrammar.Transformations
module GraphGrammar.Transformation (rewrite) where

import Data.Maybe
import Data.List
import Data.IntMap (IntMap, keys, fromList)
import GraphGrammar.Graph
import GraphGrammar.Rule
import GraphGrammar.Match (findMatches)
import Assorted.PrettyPrint
import Control.Monad

rewrite :: (Monad m, Eq a, Eq b) => Rule a b -> TypedDigraph a b -> Morphism -> m (TypedDigraph a b)
rewrite rule tGraph match = let
  (TypedDigraph graph _) = tGraph
  ns = renameStart $ nodes graph
  es = renameStart $ edges graph
  renamedRule = rename ns es rule match
  in applyRule renamedRule tGraph

renameStart :: GraphElement e => [e] -> Int
renameStart es = 1 + (maximum $ map elemId es)

lookup' i = maybe i id . lookup i

renameNode :: ([Int, Int]) -> Node a -> Node a
renameNode nm (Node id t p) = Node (lookup' id nm) t p

renameEdge nm em (Edge id st t p) = Edge (lookup' id em) (double (x -> lookup' x nm) st) t p

renameAction :: GraphElement e => (e -> e) -> Action e -> Action e
declareAction f = double (liftM f)

double :: (a -> b) -> (a, a) -> (b, b)
double f (x, y) = (f x, f y)

rename :: (Eq a, Eq b) => Int -> Int -> Rule a b -> Morphism -> Rule a b
rename ns es (Rule n mt nr er) (nm, em) = let
  elemIds :: (Eq e, GraphElement e) => [e] -> [Int]
  elemIds = map elemId
  addElements :: (Eq e, GraphElement e) => [Action e] -> [e]
  addElements = map (fromJust . snd) . filter ((== Nothing) . fst)
  idMap :: (Eq e, GraphElement e) => [Action e] -> [Int] -> [(Int, Int)]
  idMap xs ys = zip (elemIds $ addElements xs) ys
  nodeIdMap = (idMap nr [ns..]) ++ nm
  edgeIdMap = (idMap er [es..]) ++ em
  nodeIdMap = renameAction (renameNode nodeIdMap)
  edgeIdMap = renameAction (renameEdge edgeIdMap edgeIdMap)
\begin{verbatim}
modules GraphGrammar.Match
  (findMatches
   , findMatchesR
   , isSurjective
   , findIsoMorphisms
   , isIsomorphic
   )
where

import Control.Monad -- foldM
import Data.Maybe
import qualified GraphGrammar.Graph as D
import qualified GraphGrammar.Rule as D
import qualified Data.IntMap as IM
import qualified Data.List as L
import qualified Data.Set as S

type MapSet = (S.Set (Int, Int), S.Set (Int, Int))

findMatches :: D.MatchType -> D.TypedDigraph a b -> D.TypedDigraph a b -> [D.Morphism]
findMatches mt l g = findMatchesR D.emptyRule mt l g

findMatchesR :: D.Rule a b -> D.MatchType -> D.TypedDigraph a b -> D.TypedDigraph a b -> [D.Morphism]
findMatchesR r mt l g = let matches = matchGraphs r mt l g
                        in map (\(nm, em) -> (S.toList nm, S.toList em)) matches

edgeTypeCondGen :: D.Edge b -> EdgeCondition b
edgeTypeCondGen le = (\ge -> D.typeId le == D.typeId ge)

srcTypeCondGen :: D.TypedDigraph a b -> D.Edge b -> D.TypedDigraph a b -> EdgeCondition b
srcTypeCondGen le ge l = (\le l -> D.srcType le l == D.srcType ge l)

srcIDCondGen :: D.Edge b -> MapSet
srcIDCondGen le m@(nmatches, _) = (\ge -> let lsrc = D.sourceID le
                        gsrc = D.sourceID ge
                        )
    
\end{verbatim}
matched = L.find \(s, t \rightarrow s == lsrc\) \$ S.
    toList nmatches

    in case matched of
        Just (_, n) -> gsrc == n
        otherwise -> True

:: D.Edge b
    -> MapSet
    -> EdgeCondition b
tarIDCondGen le m@(nmatches, _) =
    \(\ge \rightarrow \)
    let ltar = D.targetID le
gtar = D.targetID ge

    matched = L.find \(s, t \rightarrow s == ltar\) \$ S.
toList nmatches

    in case matched of
        Just (_, n) -> gtar == n
        otherwise -> True

loopCondGen :: D.Edge b -> EdgeCondition b
loopCondGen le =
    \(\ge \rightarrow \)
    let
        lsrc = D.sourceID le
        ltar = D.targetID le
        gsrc = D.sourceID ge
        gtar = D.targetID ge

    in if lsrc == ltar
        then gsrc == gtar
        else True

generateEdgeConds
    :: D.TypedDigraph a b
    -> D.Edge b
    -> D.TypedDigraph a b
    -> MapSet
    -> [EdgeCondition b]
generateEdgeConds l le g m =
    edgeTypeCondGen le :
    srcIDCondGen le m :
    tarIDCondGen le m :
    loopCondGen le :

processEdge :: [EdgeCondition b] -> D.Edge b -> Bool
processEdge cl e =
    L.foldr \(\c acc \rightarrow (c e) && acc\) True cl

type NodeCondition a = D.Node a -> Bool

nodeTypeCondGen :: D.Node a -> NodeCondition a
nodeTypeCondGen ln =
    \(\n \rightarrow D.typeId ln == D.typeId n\)

danglingCondGen ::
    D.Rule a b
    -> D.TypedDigraph a b
    -> D.Node a
danglingCondGen r g ln = 
  if toBeDeleted r (D.elemId ln) 
    then (\gn -> not $ D.hasEdge g gn) 
    else (\gn -> True)

delCondGen :: 
  D.Rule a b -> D.Node a 
  -> MapSet 
  -> NodeCondition a 
  delCondGen r ln m = 
    (\gn -> (not $ isMapped gn m) || 
      (toBeDeleted r (D.elemId ln) == mappedToDel r m gn))

isMapped :: D.Node a -> MapSet -> Bool 
isMapped gn (nmaps, _) = 
  let found = S.filter (\(_, gnode) -> gnode == D.elemId gn) nmaps 
    in not $ S.null found

mappedToDel :: D.Rule a b -> MapSet -> D.Node a -> Bool 
mappedToDel r (nmaps, _) n = 
  let nmap = S.filter (\(lnid, gnid) -> 
                        gnid == nid && toBeDeleted r lnid) nmaps 
    in not $ S.null nmap 
  where nid = D.elemId n

toBeDeleted :: D.Rule a b -> Int -> Bool 
toBeDeleted r@(D.Rule _ _ nal _) nid = 
  let naction = L.find (\na -> 
                        case na of 
                          (Just ln, _) -> D.elemId ln == nid 
                          otherwise -> False 
                        ) nal 
    in case naction of 
      Just (_, Nothing) -> True 
      otherwise -> False

generateConds :: 
  D.Rule a b 
  -> D.TypedDigraph a b 
  -> D.Node a 
  -> D.TypedDigraph a b 
  -> MapSet 
  -> [NodeCondition a] 
generateConds r l ln g m = 
  nodeTypeCondGen ln 
  delCondGen r ln m 
  danglingCondGen r g ln 
  []

processNode :: [NodeCondition a] -> D.Node a -> Bool 
processNode cl n = 
  L.foldr (\c acc -> (c n) && acc) True cl 

mapGraphs
:: D.Rule a b
-> D.MatchType
-> D.TypedDigraph a b  -- ^ @l@, the "left side" graph
-> (MapSet, D.TypedDigraph a b, [D.Edge b], [D.Node a])  -- ^
@m@, what already got mapped
-> [(MapSet, D.TypedDigraph a b, [D.Edge b], [D.Node a])]
mapGraphs _ mt _ ml@(\((nmap, emap), D.TypedDigraph dg@(D.Digraph gnm
gem), __, [], []) =
case mt of
  D.Epi -> let
    gMappedNodes = S.fold (\(ln, gn) acc -> S.insert gn
acc) S.empty nmap
    gMappedEdges = S.fold (\(le, ge) acc -> S.insert ge
acc) S.empty emap
    in
      if S.size gMappedNodes == IM.size gnm && S.size
    gMappedEdges == IM.size gem
        then [ml]
      else []
  D.Iso -> if D.
null dg
      then [ml]
      else []
  otherwise -> [ml]
mapGraphs r mt l (m@(nmatch, ematch),
g@(D.TypedDigraph dg@(D.Digraph gnm gem) tg),
(le:les), lns) =
let
  conds = generateEdgeConds l le g m
  edgeList = D.edges dg
  candidates = filter (processEdge conds) $ edgeList
  newMapSets = fmap
    (\ge ->
      let
        sid = D.sourceID ge
        tid = D.targetID ge
        eid = D.elemId ge
        newLNodeList = L.filter (\n          in
            let nid = D.elemId n
                in (nid /= D.
                    sourceID le) &&
                    (nid /= D.
                        targetID le)
          ) lns
        in
            S.insert (D.sourceID le, sid) $ 
            S.insert (D.targetID le, tid) $ 
            nmatch,
            S.insert (D.elemId le, eid) ematch

      ),
      if mt == D.Normal || mt == D.Epi
        then g
      else (D.TypedDigraph (D.Digraph (IM.
          delete sid $ IM.delete tid gnm)
mapGraphs r mt l =
  let
    candidates = filter (processNode conds) $ D.nodes dg
    newMapSets = fmap
      (\gn ->
        let gid = D.elemId gn
        in
          ((S.insert (D.elemId ln, gid) nmatch, ematch),
           if mt == D.Normal || mt == D.Epi
           then g
           else D.TypedDigraph (D.Digraph (IM.delete gid gnm) gem) tg,
           []), (ln:lns))
      ) candidates
  in newMapSets >>= mapGraphs r mt l

matchGraphs :: D.Rule a b -> D.MatchType -> D.TypedDigraph a b ->
            D.TypedDigraph a b -> [MapSet]
matchGraphs r mt l@(D.TypedDigraph (D.Digraph lnm lem) _) g =
  map (\(m, _, _, _) -> m) $
  mapGraphs r mt l ((S.empty, S.empty), g, D.edges dl,
    D.nodes dl)

isSurjective :: D.TypedDigraph a b -> MapSet -> Bool
isSurjective (D.TypedDigraph (D.Digraph gnm gem) _) m@(nmaps, emaps) =
  IM.size gnm == S.size nmaps && IM.size gem == S.size emaps
findIsoMorphisms :: D.TypedDigraph a b -> D.TypedDigraph a b -> [D.Morphism]
findIsoMorphisms l@(D.TypedDigraph (D.Digraph lnm lem) _) g@(D.TypedDigraph (D.Digraph qnm qem) _) =
  if IM.size lnm /= IM.size qnm ||
  IM.size lem /= IM.size qem
  then []
  else findMatchesR D.emptyRule D.Iso l g

isIsomorphic :: D.TypedDigraph a b -> D.TypedDigraph a b -> Bool
isIsomorphic a b = findIsoMorphisms a b /= []
---
LANGUAGE DoAndIfThenElse --
module GraphGrammar.StateSpace ( StateSpace (..)
                   , runStateSpace
                   ) where

import Prelude

import GraphGrammar.Builder.Graph
import GraphGrammar.Graph
import GraphGrammar.Match
import GraphGrammar.Rule
import GraphGrammar.Transformation
import Assorted.PrettyPrint

import Data.List
import Data.Maybe
import Data.IntMap (fromList,toList,elems)
import qualified Data.IntMap as IM

import Control.Monad
import Control.Monad.IO.Class

type StateSpace a b = Digraph (TypedDigraph a b) (Rule a b, Morphism)
type SSBuilder a b m r = GraphBuilder (TypedDigraph a b) (Rule a b, Morphism) m r

tpg (TypedDigraph _ t) = t
ns (Digraph n _) = n

onSnd :: (b -> c) -> (a, b) -> (a, c)
onSnd f (x, y) = (x, f y)

runStateSpace :: (Eq a, Eq b, MonadIO m) => Int -> TypedDigraph a b -> [Rule a b] -> m (StateSpace a b)
runStateSpace n g r = buildGraph $ do
  i <- createNode g 1
  mkStateSpace n i g r

mkStateSpace :: (Eq a, Eq b, MonadIO m) => Int -> Int
  -> TypedDigraph a b -> [Rule a b] -> m (SSBuilder a b)
mkStateSpace 0 _ _ _ = return ()
mkStateSpace n i g r = do
  forM_ r $ \r' -> do
    let (Rule _ mt _ _) = r'
    forM_ (findMatches mt (left r' $ tpg g) g) $ \m -> do
      let t' = rewrite r' g m
      if t' == Nothing
        then return ()
      else do
        let (Just t) = t'
        i' <- putState i t (r', m)
        if i' == 0
then return ()
else mkStateSpace (n - 1) i' t r

putState :: (Eq a, Eq b, MonadIO m) => Int -> TypedDigraph a b -> (Rule a b, Morphism) -> SSBuilder a b m Int
putState i g e = do
  states <- liftM (map (onSnd payload) . toList . ns) getG
  let isos = filter snd $ map (onSnd (isIsomorphic g)) states
  case isos of
    [] -> do
      i' <- createNode g 1
    return i'
    [(i', True)] -> do
      createEdge e 1 (i, i')
    return 0
    _ -> error $ "Undefined case for " ++ show isos

A.7 GraphGrammar.Serialized

module GraphGrammar.Serialized where

import GraphGrammar.Graph
import GraphGrammar.Rule

data Serialized a b = Serialized [TypedDigraph a b] [Rule a b]
  deriving (Show, Read)

serialize :: (Show a, Show b, Read a, Read b) => [TypedDigraph a b] -> [Rule a b] -> String
serialize gs = show . Serialized gs

unserialize :: (Show a, Show b, Read a, Read b) => String -> Serialized a b
unserialize = read

A.8 GraphGrammar.GML

module GraphGrammar.GML where

data N = N Int String deriving (Show)
data E = E Int Int String deriving (Show)
data Document = Document [N] [E] deriving (Show)

newDocument :: Document
newDocument = Document [] []

newNode :: (Show l) => Int -> l -> Document -> Document
newNode i l (Document ns es) = Document ((N i (show l)):ns) es

newEdge :: (Show l) => Int -> Int -> l -> Document -> Document
newEdge s t l (Document ns es) = Document ns ((E s t (show l)):es)

strN (N i l) = unlines [ "node ", "id " ++ show i, "label " ++ l, " "]

strE (E s t l) = unlines [ "edge ", "source " ++ show s, "target " ++ show t, "label " ++ l, ""]

writeDocument :: Document -> String -> IO ()
A.9  GraphGrammar.Builder.Graph

{-# LANGUAGE GADTs #-}
module GraphGrammar.Builder.Graph (GraphBuilderT(..),
                                   GraphBuilder(..),
                                   buildGraphT,
                                   buildGraphTFrom,
                                   buildGraph,
                                   buildGraphFrom,
                                   createNode,
                                   createEdge,
                                   getG,
                                   putG,
                                   getS,
                                   putS)
where

import Prelude hiding (null,lookup,elem)
import Control.Monad.State
import Control.Monad.Identity
import Data.IntMap as IM hiding (empty,map,filter,(\))
import Data.Maybe
import Data.List hiding (lookup,null)
import GraphGrammar.Graph hiding (nodes,edges,node,edge)
import GraphGrammar.Transformation
import Assorted.Maybe

{-# A graph builder that carries extra state (s). #-}
type GraphBuilderT a b s m r = StateT (Digraph a b, s) m r

{-# allow me to help you parse the type. a is the node payload type,
b is the edge payload type, s is the carried state. Also
m is the embedded monad and r is the return type. #-
type GraphBuilder a b m r = GraphBuilderT a b () m r

{-# Builds from an empty graph #-
buildGraphT :: (Monad m) => s -> GraphBuilderT a b s m r -> m (Digraph a b)
buildGraphT = buildGraphTFrom empty

{-# Builds from an specified graph #-
buildGraphTFrom :: (Monad m) => Digraph a b -> s -> GraphBuilderT a b s m r -> m (Digraph a b)
buildGraphTFrom g s b = do (a, (g', s')) <- runStateT b (g, s);
                           return g' )

buildGraphFrom :: Monad m => Digraph a b -> GraphBuilder a b m r -> m (Digraph a b)
buildGraphFrom g = buildGraphTFrom g ()
buildGraph :: Monad m => GraphBuilder a b m r -> m (Digraph a b)
buildGraph = buildGraphFrom empty

nodes (Digraph g _) = g
edges (Digraph _ e) = e

nextId m = if IM.null m then 0 else 1 + fst (findMax m)
nextNodeId = nextId . nodes
nextEdgeId = nextId . edges

getG :: Monad m => GraphBuilderT a b s m (Digraph a b)
getG = do { s <- get; return $ fst s }

putG :: Monad m => Digraph a b -> GraphBuilderT a b s m ()
putG g = do s <- get
            put (g, snd s)

getS :: Monad m => GraphBuilderT a b s m s
getS = do { s <- get; return $ snd s }

putS :: Monad m => s -> GraphBuilderT a b s m ()
putS s = do s' <- get
            put (fst s', s)

createNode :: Monad m => a -> Int -> GraphBuilderT a b s m Int
createNode p t = do g <- getG
                     let k = nextId $ nodes g
g' <- flip addNode g $ Node k t p
                 putG g'
                 return k

createEdge :: Monad m => b -> Int -> (Int, Int) -> GraphBuilderT a b s m Int
createEdge p t c = do g <- getG
                      let k = nextId $ edges g
g' <- flip addEdge g $ Edge k c t p
                 putG g'
                 return k

A.10 Logic.Modal

module Logic.Modal ( module Logic.Modal.Graph
                     ) where

import Logic.Modal.Graph

A.11 Logic.Modal.Graph

{-# LANGUAGE TypeFamilies #-)
module Logic.Modal.Graph where

import Data.List

data Node a = Node Int [a] deriving (Show, Read, Eq)
data Edge a = Edge Int Int Int [a] deriving (Show, Read, Eq)
data Graph a = Graph [Node a] [Edge a] deriving (Show, Read, Eq)
class Element e where
  type Payload e :: *
  elementId :: e -> Int
  values :: e -> [Payload e]

instance Element (Node a) where
  type Payload (Node a) = a
  elementId (Node i _) = i
  values (Node _ v) = v

instance Element (Edge a) where
  type Payload (Edge a) = a
  elementId (Edge i _ _ _) = i
  values (Edge _ _ _ v) = v

src :: Edge a -> Int
src (Edge _ s _) = s

tgt :: Edge a -> Int
tgt (Edge _ _ t _) = t

nexts :: Graph a -> Int -> [Int]
nexts (Graph _ es) nId = map tgt $ filter (\x -> src x == nId) es

follows :: Graph a -> Int -> Int -> Bool
follows g a b = a 'elem' nexts g b

prevs :: Graph a -> Int -> [Int]
prevs g@ (Graph ns _) i = let ids = map elementId ns
  in filter (\i' -> i 'elem' nexts g i') ids

precedes :: Graph a -> Int -> Int -> Bool
precedes g a b = a 'elem' prevs g b

findById :: Element a => [a] -> Int -> [a]
findById es i = filter (\x -> elementId x == i) es

node :: Graph a -> Int -> Node a
node (Graph ns _) = head . findById ns

nodes (Graph ns _) = ns

edge :: Graph a -> Int -> Edge a
edge (Graph _ es) = head . findById es

edges (Graph _ es) = es

pathsStartingAt :: Graph a -> Int -> [[Int]]
pathsStartingAt g i = let
  ns = nexts g i
in if null ns then
  return [i]
else do
  n <- ns
  ns' <- pathsStartingAt g n
  return $ if n 'elem' ns' then ns' else i:ns'
successors :: Graph a -> Int -> [Int]
successors g s = foldl (union) [] $ pathsStartingAt g s

elementIds :: Element a => [a] -> [Int]
elementIds = map elementId

nodeIds = elementIds . nodes
eedgeIds = elementIds . edges

A.12 Logic.CTL

module Logic.CTL (check, module Logic.CTL.Parser, module Logic.CTL.Base, module Logic.CTL.Semantics, module Logic.Modal)

where

import Logic.CTL.Parser
import Logic.CTL.Base
import Logic.CTL.Semantics
import Logic.Modal

check :: Graph String -> CTL -> Int -> Bool
check g f s0 = s0 'elem' (map elementId $ ctlSat g f)

A.13 Logic.CTL.Base

data CTL = Literal Bool
  | Atom String
  | Not CTL
  | And CTL CTL
  | Or CTL CTL
  | Implies CTL CTL
  | AllNext CTL
  | SomeNext CTL
  | AllFuture CTL
  | SomeFuture CTL
  | AllGlobal CTL
  | SomeGlobal CTL
  | AllUntil CTL CTL
  | SomeUntil CTL CTL

  deriving (Show, Eq, Read)

A.14 Logic.CTL.Parser

module Logic.CTL.Parser (parseCTL) where

import Text.ParserCombinators.Parsec
import Data.List
import Logic.CTL.Base

parseCTL :: Parser CTL
parseCTL = do p <- pctl1
    eof
    return p

pctl1 = do spaces
    try parseAU <|> try parseEU <|> try implies

implies = do p <- pctl2
    try (impliesCont p) <|> return p

impliesCont p = do spaces
    string "->"
    spaces
    q <- pctl1
    return $ Implies p q

parseAU = do char 'A'
    spaces
    char '['
    spaces
    p <- pctl1
    spaces
    char 'U'
    spaces
    q <- pctl1
    spaces
    char ']'
    return $ AllUntil p q

parseEU = do char 'E'
    spaces
    char '['
    spaces
    p <- pctl1
    spaces
    char 'U'
    spaces
    q <- pctl1
    spaces
    char ']'
    return $ SomeUntil p q

pctl2 = do p <- pctl3
    try (pAndOr p) <|> return p

pAndOr p = do spaces
    try (do string "&&" ; return And) <|> try (do string "||" ; return Or)

binaryOP = do try (do string "&&" ; return And) <|> try (do string "||" ; return Or)

pctl3 = do try pNot <|> try pTempOp <|> try parens <|> try ident

pNot = do char '~'
    spaces
91 p <- pclt3
92 return $ Not p
93
94 pTempOp = do op <- tempOp
95 spaces
96 p <- pclt3
97 return $ op p
98
99 tempOp = try ax <|> try af <|> try ag <|> try ex <|> try ef <|> try eg
100
101 parens = do char '('
102 p <- pclt1
103 spaces
104 char ')
105 return p
106
107 ax = do { string "AX"; return AllNext }
108 ag = do { string "AG"; return AllGlobal }
109 af = do { string "AF"; return AllFuture }
110 ex = do { string "EX"; return SomeNext }
111 eg = do { string "EG"; return SomeGlobal }
112 ef = do { string "EF"; return SomeFuture }
113
114 ident = do
115 c <- letter
116 cs <- many (letter <|> digit)
117 let tIdent = c:cs
118 if tIdent == "true" then
119 return true
120 else if tIdent == "false" then
121 return false
122 else if any (==tIdent) reservedWords then
123 unexpected (tIdent ++ " is a reserved word")
124 else
125 return $ Atom tIdent
126
127 reservedWords = [ "AX", "AF", "AG", "EX", "EF", "EG" ]
128 true = Literal True
129 false = Literal False

A.15 Logic.CTL.Semantics

1 module Logic.CTL.Semantics where
2 import Data.List
3 import Logic.CTL.Base
4 import Logic.Modal
5
6 ctlSat :: Graph String -> CTL -> [Node String]
7 ctlSat g (Literal True) = nodes g
8 ctlSat g (Literal False) = []
9 ctlSat g (Atom v) = filter (\x -> v 'elem' values x) $ nodes g
10 ctlSat g (Not p) = nodes g \ ctlSat g p
11 ctlSat g (And p q) = ctlSat g p 'intersect' ctlSat g q
\[
\begin{align*}
\text{ctlSat } g \ (\text{Or } p \ q) & = \text{ctlSat } g \ p \ 'union' \ \text{ctlSat } g \ q \\
\text{ctlSat } g \ (\text{Implies } p \ q) & = \text{ctlSat } g \ $ \ Or \ q \ $ \ Not \ p \\
\text{ctlSat } g \ (\text{AllNext } p) & = \text{ctlSat } g \ $ \ Not \ $ \ SomeNext \ $ \ Not \ p \\
\text{ctlSat } g \ (\text{AllFuture } p) & = \text{afSat } g \ p \\
\text{ctlSat } g \ (\text{AllGlobal } p) & = \text{ctlSat } g \ $ \ Not \ $ \ SomeFuture \ $ \ Not \ p \\
\text{ctlSat } g \ (\text{AllUntil } p \ q) & = \text{ctlSat } g \ $ \ Not \ (\text{Or } \ (\text{SomeUntil } (\text{Not } q)) \ (\text{SomeGlobal } (\text{Not } q))) \\
\text{ctlSat } g \ (\text{SomeNext } p) & = \text{exSat } g \ p \\
\text{ctlSat } g \ (\text{SomeFuture } p) & = \text{ctlSat } g \ $ \ SomeUntil \ (\text{Literal } \text{True}) \ p \\
\text{ctlSat } g \ (\text{SomeGlobal } p) & = \text{ctlSat } g \ $ \ Not \ $ \ AllFuture \ $ \ Not \ p \\
\text{ctlSat } g \ (\text{SomeUntil } p \ q) & = \text{euSat } g \ p \ q \\
\end{align*}
\]

\[
\begin{align*}
\text{afSat} :: \text{Graph } \text{String} \rightarrow \text{CTL} \rightarrow \left[\text{Node } \text{String}\right] \\
\text{afSat } g \ p & = \text{let } x = \text{elementIds } $ \ \text{nodes } g \\
& \quad \ y = \text{elementIds } $ \ \text{ctlSat } g \ p \\
& \quad \text{in } \ \text{nodesById } g \ $ \ \text{recurse } x \ y \\
& \quad \text{where } \ \text{recurse } x \ y \\
& \quad \quad | \ x == y \ = y \\
& \quad \quad \ | \ \text{otherwise} \ = \text{let } x’ = y \\
& \quad \quad \quad y’ = y \ 'union' \ \text{preA } g \ y \\
& \quad \quad \ \quad \text{in } \ \text{recurse } x’ \ y’ \\
\end{align*}
\]

\[
\begin{align*}
\text{exSat} :: \text{Graph } \text{String} \rightarrow \text{CTL} \rightarrow \left[\text{Node } \text{String}\right] \\
\text{exSat } g \ p & = \text{let } x = \text{elementIds } $ \ \text{nodes } g \\
& \quad \ y = \ \text{preE } g \ x \\
& \quad \text{in } \ \text{nodesById } g \ y \\
\text{euSat } g \ p \ q & = \text{let } w = \text{elementIds } $ \ \text{ctlSat } g \ p \\
& \quad \ x = \text{elementIds } $ \ \text{nodes } g \\
& \quad \ y = \text{elementIds } $ \ \text{ctlSat } g \ q \\
& \quad \text{in } \ \text{nodesById } g \ $ \ \text{recurse } w \ x \ y \\
& \quad \text{where } \ \text{recurse } w \ x \ y \\
& \quad \quad | \ x == y \ = y \\
& \quad \quad \ | \ \text{otherwise} \ = \text{let } x’ = y \\
& \quad \quad \quad y’ = y \ 'union' \ (w \ 'intersect' \ \text{preE } g \ y) \\
& \quad \quad \ \quad \text{in } \ \text{recurse } w’ \ y’ \\
\end{align*}
\]

\[
\begin{align*}
\text{preA} :: \text{Graph } a \rightarrow \left[\text{Int}\right] \rightarrow \left[\text{Int}\right] \\
\text{preA } g \ y & = \text{let } \text{ant} = \text{foldl } \text{union } [\ ] \ $ \ \text{map} \ (\text{prevs } g) \ y \\
& \quad \quad \ \text{condition } s = \text{nexts } g \ s \ 'subsetOf' \ y \\
& \quad \quad \ \text{in } \ \text{filter} \ \text{condition } \ \text{ant} \\
\text{preE} :: \text{Graph } a \rightarrow \left[\text{Int}\right] \rightarrow \left[\text{Int}\right] \\
\text{preE } g \ \text{ids} & = \text{foldl } \text{union } [\ ] \ $ \ \text{map} \ (\text{prevs } g) \ \text{ids} \\
\text{subsetOf} :: \text{Eq } a \Rightarrow \left[a\right] \rightarrow \left[a\right] \rightarrow \text{Bool} \\
\text{subsetOf } as \ bs & = \text{all } ('elem' \ bs) \ as \\
\text{nodesById } g & = \text{map} \ (\text{node } g)
\end{align*}
\]

A.16 Translation
import qualified Logic.Modal as Model

translate :: Grammar.StateSpace String String -> Model.Graph String
translate g = let gns = Grammar.nodes g
               ges = Grammar.edges g
               edgesConnected n = filter (\x -> Grammar.elemId n == Grammar.sourceID x) ges
               ruleName (Grammar.Edge _ _ _ (Grammar.Rule n _ _ _, _)) = n
               atoms = nub . map ruleName . edgesConnected
               nodes = map (\n@(Grammar.Node i _ _) -> Model.Node i $ atoms n) gns
               edges = map (\(Grammar.Edge i (s, t) _ _) -> Model .Edge i s t []) ges
           in Model.Graph nodes edges

A.17 Main

module Main where

import System.Console.GetOpt
import System.Environment
import Control.Monad
import Control.Monad.IO.Class
import Text.ParserCombinators.Parsec
import GraphGrammar
import Logic.CTL
import Translation
import GML

main = do
  (options, strs) <- parseOptions
  stateSpace <- runSerializedGrammar (stepsToStop options) (head strs)
  let model = translate stateSpace
  forM_ (tail strs) $ \f -> do
    let f' = parseFormula f
    if check model f' $ initialState options
    then putStrLn $ concat ["Formula ", f, ", " holds for state ", show $ initialState options]
    else putStrLn $ concat ["Formula ", f, ", " fails to hold on state ", show $ initialState options]
  if graphicalOutput options
  then writeStateSpace stateSpace "state_space.gml" >>
    writeStates stateSpace >>
    writeModel model "model.gml"
  else return ()

data SystemOptions = SystemOptions { stepsToStop :: Int
                                      , graphicalOutput :: Bool
                                      }
, initialState :: Int

parseOptions :: MonadIO m => m (SystemOptions, [String])
parseOptions = do args <- liftIO getArgs
  case getOpt Permute options args of
    (o, (h:t), []) -> return (foldl (flip id)
      defaultOptions o, (h:t))
    (_, _, es) -> fail $ concat es ++
      usageInfo header options
  where header = "Usage: main [args] file checks"

options = [ Option [‘s’] ["stop-after"] (ReqArg (\d o -> o { stepsToStop = read d }) "n") "stops after n iterations"
  , Option [‘d’] ["draw"] (NoArg (\o -> o { graphicalOutput = True })) "writes the state space as a series of images"
  , Option [‘i’] ["initial-state"] (ReqArg (\d o -> o { initialState = read d }) "n") "evaluates the formulas for state n"
]

defaultOptions = SystemOptions { stepsToStop = 10000
  , graphicalOutput = False
  , initialState = 0
}

parseFormula :: String -> CTL
parseFormula s = case parse parseCTL "" s of
  Right ctl -> ctl
  Left msg -> error $ show msg
**GLOSSARY**

**double pushout**  One technique for graph rewriting, based on category theory. 19

**dynamic verification**  Verification done in the execution of a system. 17, 18

**graph grammar**  A rewriting system build on graphs and graph derivations. 26

**model checker**  A tool that performs model checking. 31, 42

**model checking**  A method of dynamic verification. 31

**second-order graph grammars**  A method for transforming graph transformations. 17

**static verification**  Verifications done with on the structure of a system. 17