UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

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

CIP – CATALOGING-IN-PUBLICATION

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.

Monograph – Universidade Federal do Rio Grande do Sul. Curso de Bacharelado em Ciência da Computação, Porto Alegre, BR–RS, 2014. Advisor: Rodrigo Machado.

1. Model Checking. 2. Graph Grammars. 3. Temporal Logic. I. Machado, Rodrigo. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Graduação: Prof. Sérgio Roberto Kieling Franco
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do CIC: Prof. Raul Fernando Weber
Bibliotecário-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

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 Nietzche)

ACKNOWLEDGMENTS

I would like to aknowledge the great people that helped me during this journey, friends and family. In special, I would like to thank my father, mother and sibilings for supporting and helping me during the process of writing this monograhy: 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

CONTENTS 7

LIST OF FIGURES 11

ACRONYMS 15

ABSTRACT 15

RESUMO 17

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

2 GRAPH GRAMMARS 23
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 35
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

APPENDIXA 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

A.8 GraphGrammar.GML 85
A.9 GraphGrammar.Builder.Graph 86
A.10 Logic.Modal 87
A.11 Logic.Modal.Graph 87
A.12 Logic.CTL 89
A.13 Logic.CTL.Base 89
A.14 Logic.CTL.Parser 89
A.15 Logic.CTL.Semantics 91
A.16 Translation 92
A.17 Main 93

GLOSSARY 95

LIST OF FIGURES

2.1 An example graph in its set form (left) and graphical form (right). . . . 24
2.2 An example of graph morphism. 24
2.3 A typed morphism over T. 26
2.4 A graph production. 27
2.5 The derivation of a graph G with the production in Figure 2.4. 28
2.6 Examples of violations of (a) identification condition and (b) dangling condition. 29
2.7 A grammar depicting a train system. 31
2.8 The full state space geneerated by the execution of the grammar in Figure 2.7 33
2.9 An example of an grammar that generates an infinite space state. . . . 34

3.1 A model in its graphical representation. 38
3.2 A model satisfying the formula a. 39
3.3 A model satisfying the formula $AX\ a$. 39
3.4 A model satisfying the formula $EF\ a$. 40
3.5 A model satisfying the formula $E[a\ U\ b]$. 41
3.6 Model in Figure 3.1, with labels on the states. 44
3.7 Graph Grammar translation into a CTL model. 45
3.8 The conversion from a simple state space to a model. 46

4.1 The system architecture's block diagram 49
4.2 Grammar used in the experiments 61
4.3 Measurements for the experiment 63

LIST OF ALGORITHMS

1 The rewrite algorithm. 29

2 The SAT function. 42

3 The auxiliary function pre_E. 42

4 The auxiliary function pre_A 42

5 The SAT_{EX} function. 42

6 The SAT_{AF} function. 43

7 The SAT_{EU} function. 43

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.

VeriGraph: uma ferramenta para verificaçao de modelos de gramáticas de grafos

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,

2012);

- 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 focus 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 $s, t : E \to V$ 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.

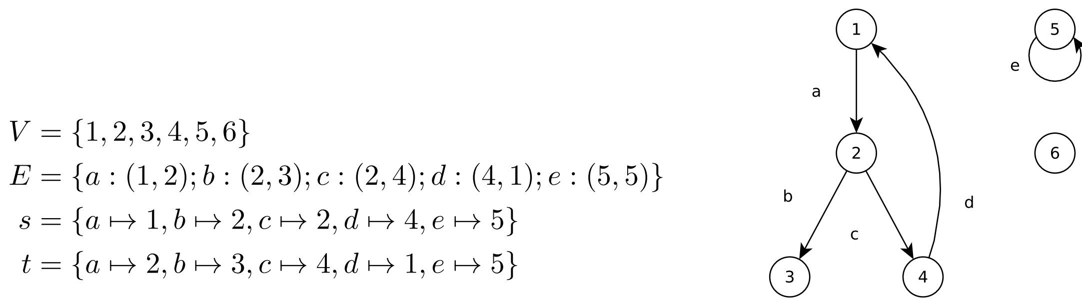


Figure 2.1: An example graph in its set form (left) and graphical form (right).

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.

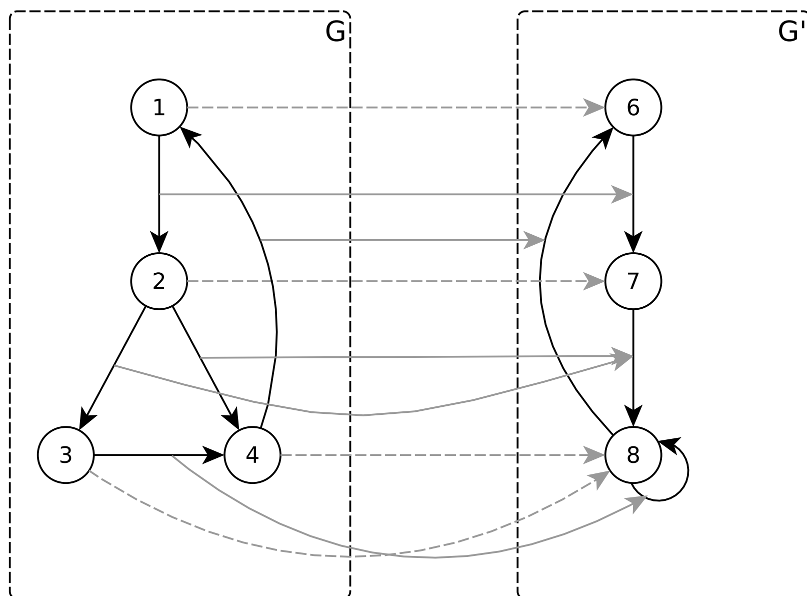


Figure 2.2: An example of graph morphism.

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 **Graph**. It's assumed that the reader has basic knowledge of category theory; otherwise, the user is refereed to (HERRLICH; STRECKER, 1973). The DPO method is called an *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.*

$$\begin{array}{ccc} G & \xrightarrow{m} & G' \\ & {\scriptstyle t}\searrow \quad \swarrow{\scriptstyle t'} & \\ & T & \end{array}$$

The category of all T-typed graphs and T-typed morphisms is called **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.

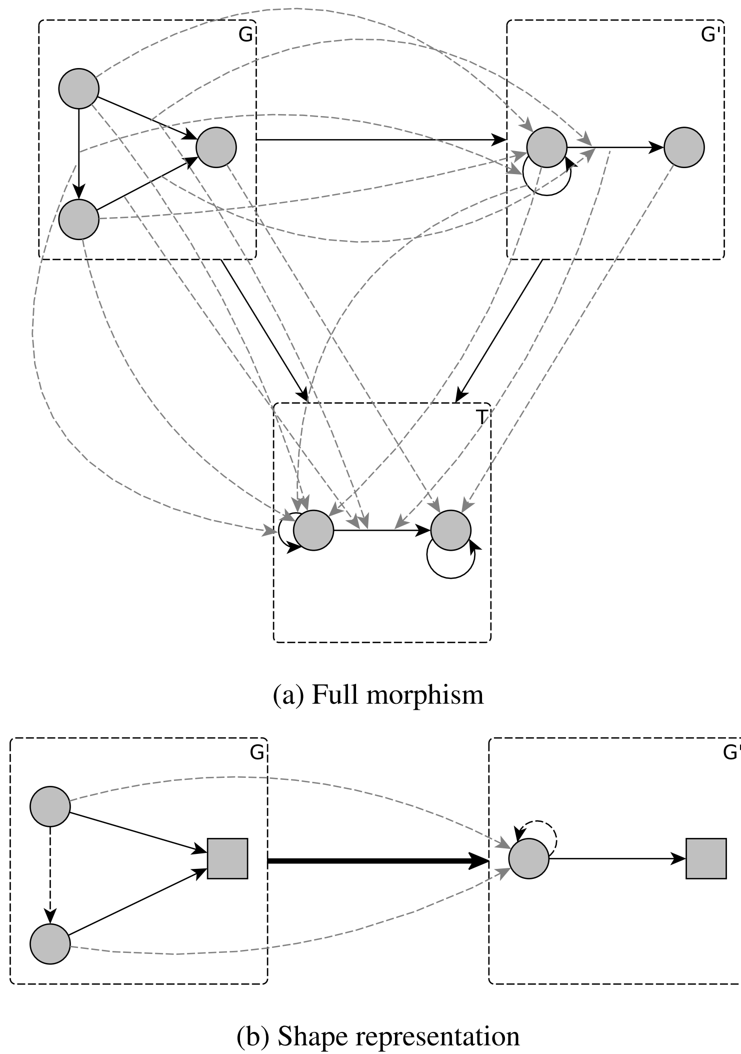
2.2 The double pushout approach

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, *productions* or *rules* are represented by a pair of morphisms with the same origin.

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

$$L \overset{l}{\leftarrowtail} K \overset{r}{\rightarrowtail} R$$

*A production is a span in **T-Graph**, in which l and r are total. The collection of all productions is called $\mathcal{P}$.*



(a) Full morphism

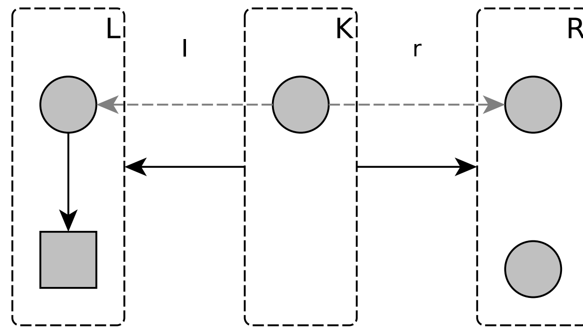
(b) Shape representation

Figure 2.3: A typed morphism over T.

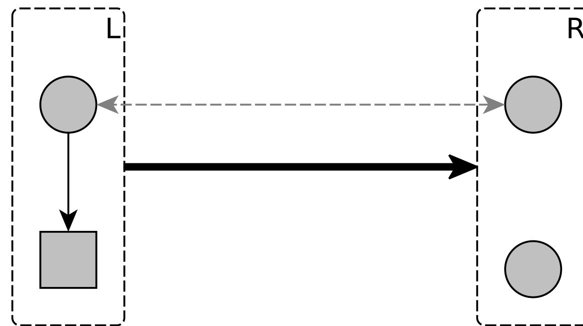
A production $r : L \overset{l}{\hookleftarrow} K \overset{r}{\rightarrowtail} R$ describes how to modify a graph locally. The graph L is the *pattern graph* (or simply *pattern*) that must be found in the target graph G, and the graph R (also known as *replacement graph*, or simply *replacement*) is 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 $p : L \overset{l}{\hookleftarrow} K \overset{r}{\rightarrowtail} R$ to a state graph G, one must find a match of



(a) Full representation



(b) Short representation

Figure 2.4: A graph production.

L into G, which is a morphism $m : L \to G$. The rewriting is defined as a double pushout diagram on **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* **T-*Graph***, *$p : L \overset{l}{\leftarrowtail} K \overset{r}{\rightarrowtail} R$ a production in* **T-*Graph*** *and $m : L \to G$ a match. A* graph derivation *(or* graph rewriting*) from G to H exists if the diagram*

$$\begin{array}{ccccc} L & \overset{l}{\leftarrowtail} & K & \overset{r}{\rightarrowtail} & R \\ {\scriptstyle m}\downarrow & & {\scriptstyle k}\downarrow & & \downarrow{\scriptstyle m'} \\ G & \overset{l'}{\longleftarrow} & D & \overset{r'}{\longrightarrow} & H \end{array}$$

can be constructed in **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 $G \overset{p,m}{\Longrightarrow} H$ in the following text, or simply $G \overset{p}{\Longrightarrow} H$, 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.

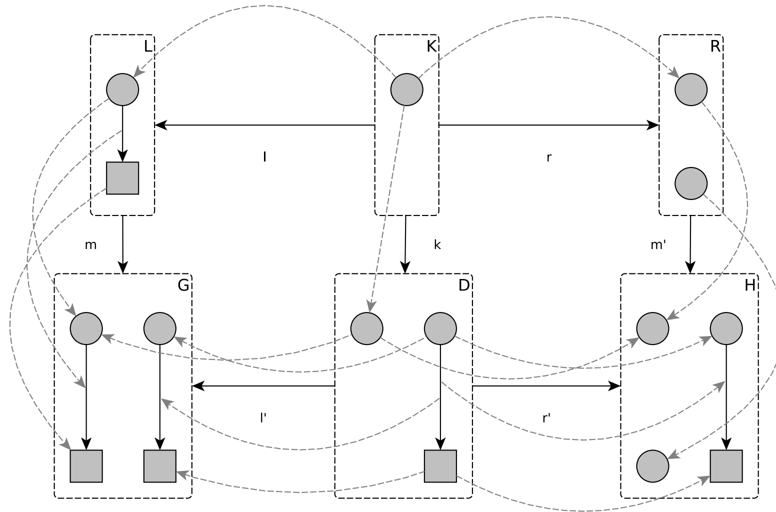


Figure 2.5: The derivation of a graph G with the production in Figure 2.4.

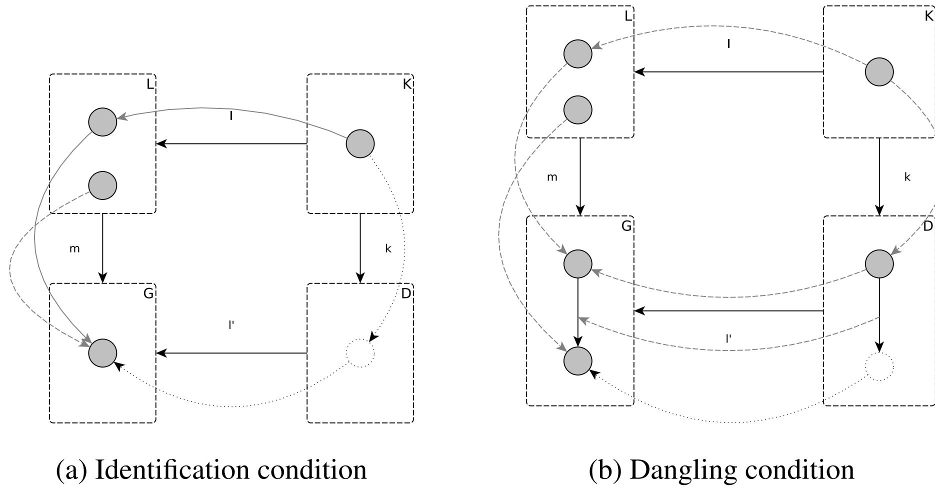
The first step to rewrite a graph is find a morphism $m : L \to G$, that matches the elements of L with the elements of G, marking that region for rewriting. The match is calculated by solving the *subgraph homomorphism problem* for L and G, or the *subgraph isomorphism problem* if the application requires 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 $m : L \to G$ 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.



(a) Identification condition

(b) Dangling condition

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 $p : L \overset{l}{\hookleftarrow} K \overset{r}{\rightarrowtail} R$, and a match $m : L \to G$

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** $(\exists e \in E_H \mid s_H(e) = n \vee t_H(e) = n) \vee (e \notin V_H)$ **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* $\mathcal{G} = \langle G_0, P, \pi \rangle$ *where* P *is a set of production names,* $\pi : P \to \mathcal{P}$ *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 $G_n \overset{p,m}{\Longrightarrow} G_{n+1}$, generating new states G_{n+1} for the system, models the behavior of the system. At any state G_i, any production $p_j : L_j \overset{l_j}{\leftarrowtail} K_j \overset{r_j}{\rightarrowtail} R_j$ can be applied provided a match $m_{ij} : L_j \to G_i$ 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* $\delta_1 : G \overset{p}{\Longrightarrow} G'$ *and* $\delta_2 : H \overset{q}{\Longrightarrow} H'$ *are said* equivalent *if*

- $H \in [G]_{\cong}$
- $H' \in [G']_{\cong}$ *and*
- $p = q$

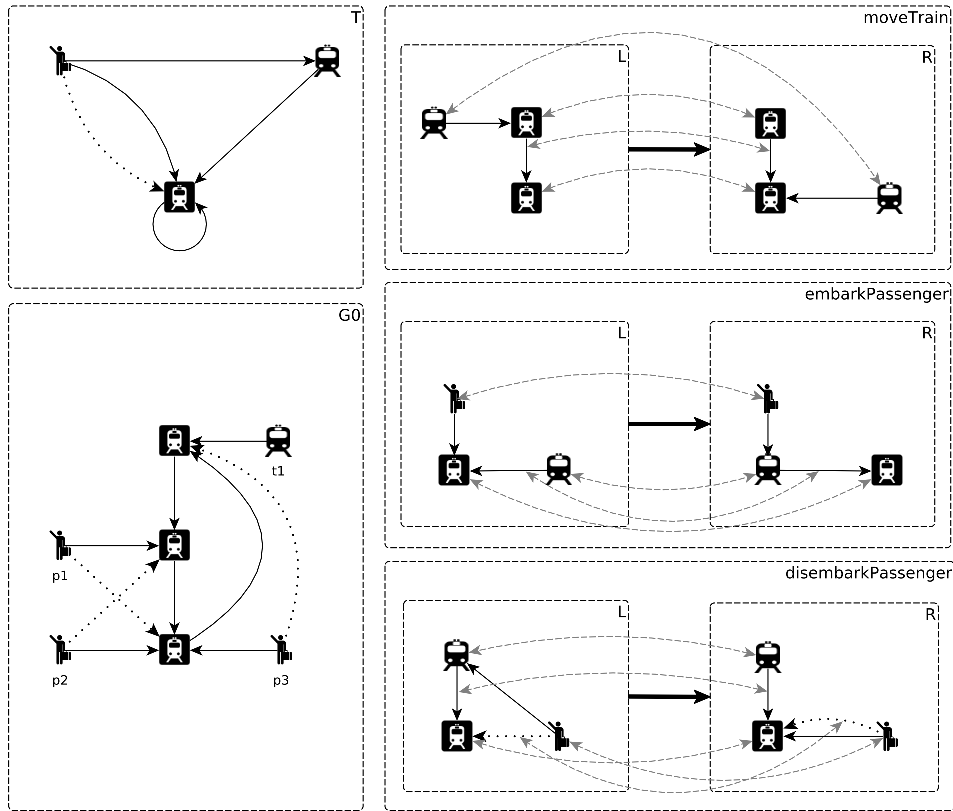


Figure 2.7: A grammar depicting a train system.

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* $\delta : G \stackrel{p}{\Longrightarrow} H$ *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* $\mathcal{S} = \langle S, T, \chi_0, \chi_1 \rangle$ *where* S *is a collections 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:*

- $[G_0]_{\cong} \in S$
- *for each* $[G]_{\cong} \in S$*, if* $\delta : G \overset{p}{\Longrightarrow} G'$*, then* $[G']_{\cong} \in S$ *and* $[\delta]_{\equiv} \in T$

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.

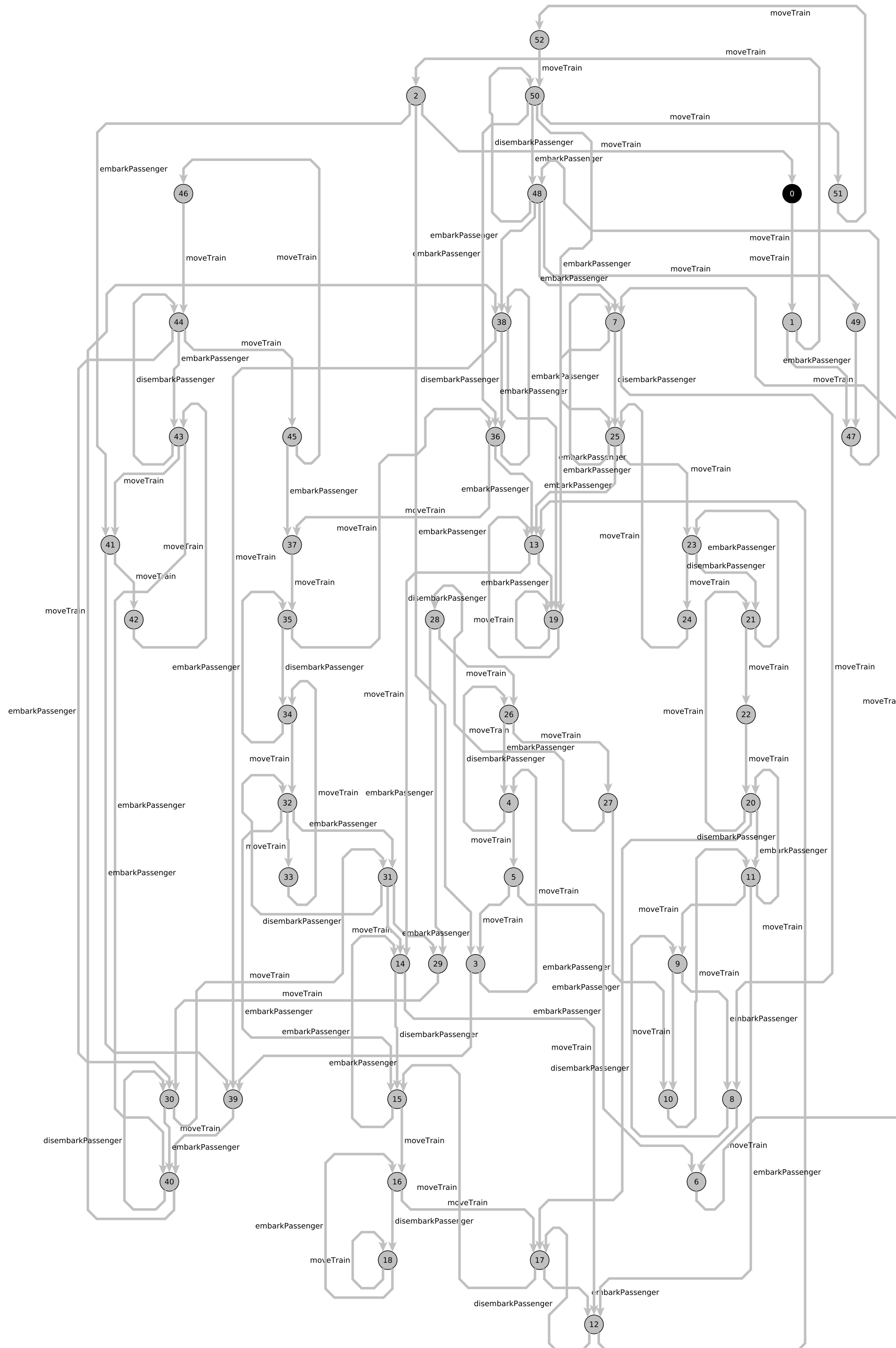


Figure 2.8: The full state space geneerated by the execution of the grammar in Figure 2.7
.

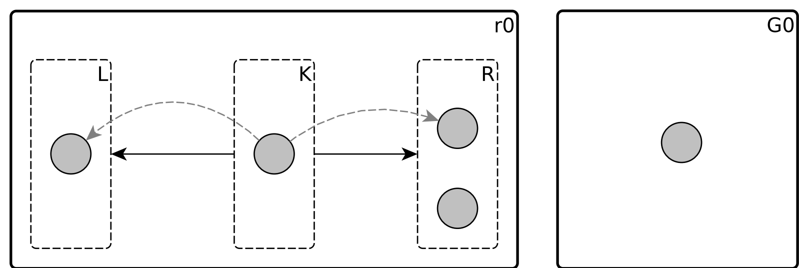


Figure 2.9: An example of an grammar that generates an infinite space state.

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 example[1].

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.

[1] The 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*

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

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 $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\iff$ work as the same operators in classical logic, and are resolved in the same way. The remainig 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 $p \wedge q$ can be converted to $\neg(\neg p \vee \neg q)$ 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* φ' *as*

$$\begin{aligned} length(\varphi' \in \{\top, \bot\} \cup P) &= 1 \\ length(\varphi' \in \{\neg\varphi, AX\varphi, AF\varphi, AG\varphi\}) &= 1 + length(\varphi) \\ length(\varphi' \in \{EX\varphi, EF\varphi, EG\varphi, (\varphi)\}) &= 1 + length(\varphi) \\ length(\varphi' \in \{\varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2, \varphi_1 \Rightarrow \varphi_2\}) &= 1 + length(\varphi_1) + length(\varphi_2) \\ length(\varphi' \in \{\varphi_1 \Leftrightarrow \varphi_2, A[\varphi_1 U \varphi_2], E[\varphi_1 U \varphi_2]\}) &= 1 + length(\varphi_1) + length(\varphi_2) \end{aligned}$$

Example 9. The lengths of the well-formed CTL formulas in Example 8 are 3, 5 and 3, respectively.

The set of subformulas Ψ of a CTL formula φ, is defined by induction over the structure of the CTL grammar.

Definition 15 (CTL Subformulas)**.** *The set of subformulas Ψ of a CTL formula φ' is defined as*

$$\begin{aligned}
\Psi(\varphi' \in \{\top, \bot\} \cup P) &= \{\varphi'\} \\
\Psi(\varphi' \in \{\neg\varphi, (\varphi), AX\varphi, AF\varphi, AG\varphi\}) &= \{\varphi'\} \cup \Psi(\varphi) \\
\Psi(\varphi' \in \{EX\varphi, EF\varphi, EG\varphi\}) &= \{\varphi'\} \cup \Psi(\varphi) \\
\Psi(\varphi' \in \{\varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2, \varphi_1 \Rightarrow \varphi_2\}) &= \{\varphi'\} \cup \Psi(\varphi_1) \cup \Psi(\varphi_2) \\
\Psi(\varphi' \in \{\varphi_1 \Leftrightarrow \varphi_2, A[\varphi_1 U \varphi_2], E[\varphi_1 U \varphi_2]\}) &= \{\varphi'\} \cup \Psi(\varphi_1) \cup \Psi(\varphi_2)
\end{aligned}$$

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 $\mathcal{M} = \langle S, \rightarrow, L\rangle$ 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 ≥ 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 $\mathcal{M}$ 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.

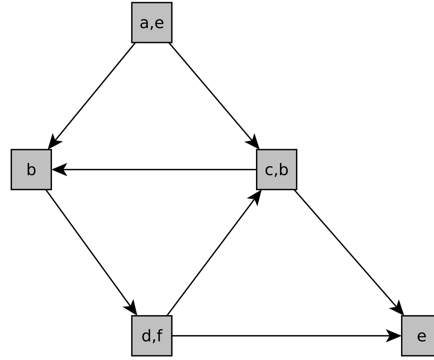


Figure 3.1: A model in its graphical representation.

Definition 17 (Semantics of CTL)**.** *The satisfaction of a formula by a given model $\mathcal{M}$ and state s, written $\mathcal{M}, s \models \varphi$, is defined as*

- $(\mathcal{M}, s \models \top)$, $(\mathcal{M}, s \not\models \bot)$
- $(\mathcal{M}, s \models p)$ *iff* $p \in L(s)$
- $(\mathcal{M}, s \models \varphi_1 \wedge \varphi_2)$ *iff* $(\mathcal{M}, s \models \varphi_1)$ *and* $(\mathcal{M}, s \models \varphi_2)$
- $(\mathcal{M}, s \models \varphi_1 \vee \varphi_2)$ *iff* $(\mathcal{M}, s \models \varphi_1)$ *or* $(\mathcal{M}, s \models \varphi_2)$
- $(\mathcal{M}, s \models \varphi_1 \Rightarrow \varphi_2)$ *iff* $(\mathcal{M}, s \models \neg\varphi_1 \vee \varphi_2)$
- $(\mathcal{M}, s \models \varphi_1 \Leftrightarrow \varphi_2)$ *iff* $(\mathcal{M}, s \models \varphi_1 \Rightarrow \varphi_2)$ *and* $(\mathcal{M}, s \models \varphi_2 \Rightarrow \varphi_1)$
- $(\mathcal{M}, s \models (\varphi))$ *iff* $(\mathcal{M}, s \models \varphi)$
- $(\mathcal{M}, s \models AX\varphi)$ *iff for all s' such that $s \rightarrow s'$,* $(\mathcal{M}, s' \models \varphi)$
- $(\mathcal{M}, s \models AF\varphi)$ *iff for all paths starting in s, there is an $i \geq 0$ such that* $(\mathcal{M}, s_i \models \varphi)$
- $(\mathcal{M}, s \models AG\varphi)$ *iff for all paths starting in s and for all $i \geq 0$,* $(\mathcal{M}, s_i \models \varphi)$
- $(\mathcal{M}, s \models A[\varphi_1 U \varphi_2])$ *iff for all paths starting in s there is a $j \geq 0$ such that* $(\mathcal{M}, s_j \models \varphi_2)$ *and for all $0 \leq i < j$* $(\mathcal{M}, s_i \models \varphi_1)$
- $(\mathcal{M}, s \models EX\varphi)$ *iff exists s' such that $s \rightarrow s'$,* $(\mathcal{M}, s' \models \varphi)$
- $(\mathcal{M}, s \models EF\varphi)$ *iff exists a path starting in s, there is an $i \geq 0$ such that* $(\mathcal{M}, s_i \models \varphi)$
- $(\mathcal{M}, s \models EG\varphi)$ *iff exists a path starting in s and for all $i \geq 0$,* $(\mathcal{M}, s_i \models \varphi)$
- $(\mathcal{M}, s \models E[\varphi_1 U \varphi_2])$ *iff exists a path starting in s there is a $j \geq 0$ such that* $(\mathcal{M}, s_j \models \varphi_2)$ *and for all $0 \leq i < j$* $(\mathcal{M}, s_i \models \varphi_1)$

For a formula φ with A to hold on a model, all paths starting at the initial state must satisfy φ. For E , only one path need to satisfy φ.

The temporal operators check wheter the formula φ holds in a path. The operator X check if the next state of a given state satisfies φ. F checks if some state in the future of the initial state satisfies φ. G checks wheter all states in the initial state's future satisfy φ holds. U verifies if, in a path, all states satisfy φ_1 until φ_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.

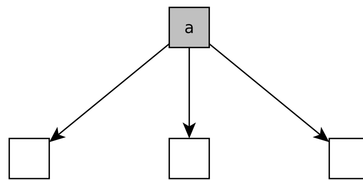


Figure 3.2: A model satisfying the formula a.

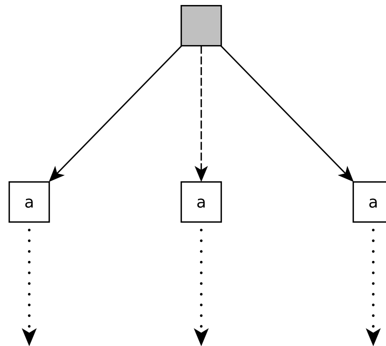


Figure 3.3: A model satisfying the formula $AX\ a$.

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 $(\mathcal{M}, s_0 \models EF\ embarkPassenger)$. Other examples:

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

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

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

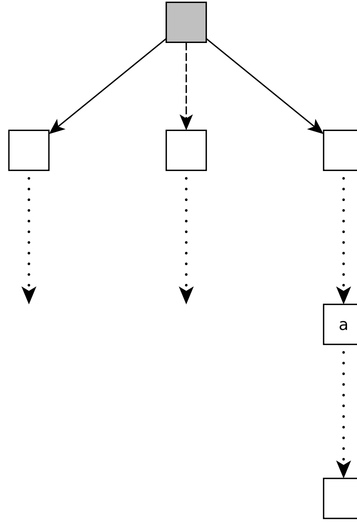


Figure 3.4: A model satisfying the formula $EF\ a$.

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, φ holds is equivalent to that "there's not a future in which φ is not satisfied"). First we define the equivalence of formulas.

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

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).

$$
\begin{aligned}
\neg AF\varphi &\equiv EG\neg\varphi \\
\neg EF\varphi &\equiv AG\neg\varphi \\
\neg AX\varphi &\equiv EX\neg\varphi \\
AX\varphi &\equiv \neg EX\neg\varphi \\
EG\varphi &\equiv \neg AF\neg\varphi \\
AF\varphi &\equiv A[\top U\varphi] \\
EF\varphi &\equiv E[\top U\varphi] \\
AG\varphi &\equiv \neg EF\neg\varphi \\
A[\varphi U\phi] &\equiv \neg(E[\neg\phi U(\neg\varphi \wedge \neg\phi)] \vee EG\neg\phi)
\end{aligned}
$$

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 $\{\wedge, \vee\}$, $\neg$ and EU (MARTIN, 2002). From the definitions above, we can define a minimal core $\{\top, \wedge, \neg, EX, EU, AF\}$.

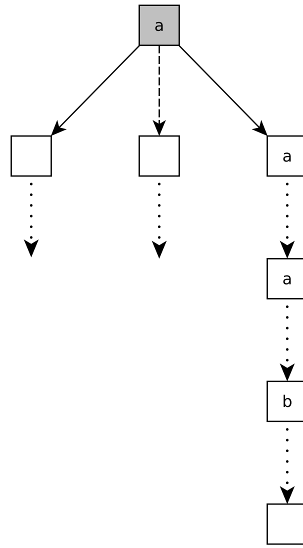


Figure 3.5: A model satisfying the formula $E[a\ U\ b]$.

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 φ that the state satisfies. The formula is satisfied in the model if the initial state is labeled with φ. 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 $\texttt{SAT}(\mathcal{M}, a \Rightarrow AX\ b)$ is

$$\begin{aligned}
\texttt{SAT}(\mathcal{M}, a \Rightarrow AX\ b) &= \texttt{SAT}(\mathcal{M}, \neg a \vee AX\ b) = \texttt{SAT}(\mathcal{M}, \neg a) \cup \texttt{SAT}(\mathcal{M}, AX\ b) \\
&= (S - \texttt{SAT}(\mathcal{M}, a)) \cup \texttt{SAT}(\mathcal{M}, \neg EX \neg b) \\
&= (S - \texttt{SAT}(\mathcal{M}, a)) \cup (S - \texttt{SAT}(\mathcal{M}, EX \neg b)) \\
&= (S - \texttt{SAT}(\mathcal{M}, a)) \cup (S - \texttt{SAT}_{EX}(\neg b, \mathcal{M}))
\end{aligned}$$

The result of $\texttt{SAT}(\mathcal{M}, a)$ is the set $\{s_0\}$. The result of $S - \texttt{SAT}(\mathcal{M}, a) = \{s_0, s_1, s_3, s_4, s_5\} - \{s_0\} = \{s_1, s_3, s_4, s_5\}$. The function $\texttt{SAT}_{EX}(\neg b, \mathcal{M})$ 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 $\texttt{SAT}(\mathcal{M}, \neg b)$ is

$$\texttt{SAT}(\mathcal{M}, \neg b) = S - \texttt{SAT}(\mathcal{M}, b) = \{s_0, s_1, s_3, s_4, s_5\} - \{s_1, s_3\} = \{s_0, s_4, s_5\}$$

```
1  function SAT(φ, M)
     input : a formula φ, and a state space M = ⟨S, →, L⟩
     output: a set of states that satisfy φ
2    switch φ do
3      case ⊤ return S;
4      case ⊥ return ∅;
5      case p return {s ∈ S | φ ∈ L(s)};
6      case ¬φ return S − SAT(φ, M);
7      case φ1 ∧ φ2 return SAT(φ1, M) ∩ SAT(φ2, M);
8      case φ1 ∨ φ2 return SAT(φ1, M) ∪ SAT(φ2, M);
9      case φ1 ⇒ φ2 return SAT(¬φ1 ∨ φ2, M);
10     case φ1 ⟺ φ2 return SAT(φ1 ⇒ φ2 ∧ φ2 ⇒ φ1, M);
11     case AXφ return SAT(¬EX¬φ, M);
12     case AFφ return SAT_AF(φ, M);
13     case AGφ return SAT(¬EF¬φ, M);
14     case A[φ1Uφ2] return SAT(¬(E[¬φ2U(¬φ1 ∧ ¬φ2)] ∨ EG¬φ2), M);
15     case EXφ return SAT_EX(φ, M);
16     case EFφ return SAT(E[⊤Uφ], M);
17     case EGφ return SAT(¬AF¬φ1, M);
18     case E[φ1Uφ2] return SAT_EU(φ1, φ2, M);
19     case (φ) return SAT(φ, M);
20   endsw
```

Algorithm 2: The SAT function.

```
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
2    return {s ∈ S | exists s', (s → s' and s' ∈ Y)}
```

Algorithm 3: The auxiliary function pre_E.

```
1  function pre_A(Y, M)
     input : set of states Y, and a state space M = ⟨S, →, L⟩
     output: A set of states that are antecessors only to states in Y
2    return {s ∈ S | for all s', (s → s' implies that 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
3    X := SAT(φ, M);
5    Y := pre_E(X, M);
6    return Y
```

Algorithm 5: The SAT_{EX} function.

```
1  function SAT_AF (φ, M)
     input  : a formula φ, and a state space M = ⟨S, →, L⟩
     output: a set of states that satisfy AFφ
3    X := S;
5    Y := SAT(φ, M) ;
6    while X ≠ Y do
8      X := Y ;
10     Y := Y ∪ pre_A(Y, M) ;
11   end
12   return Y
```

Algorithm 6: The SAT$_{AF}$ function.

```
1  function SAT_EU (φ1, φ2, M)
     input  : a formula φ, and a state space M = ⟨S, →, L⟩
     output: a set of states that satisfy φ
3    W := SAT(φ1, M) ;
5    X := S ;
7    Y := SAT(φ2, M) ;
8    while X ≠ Y do
10     X := Y ;
12     Y := Y ∪ (W ∩ pre_E(Y)) ;
13   end
14   return Y
```

Algorithm 7: The SAT$_{EU}$ function.

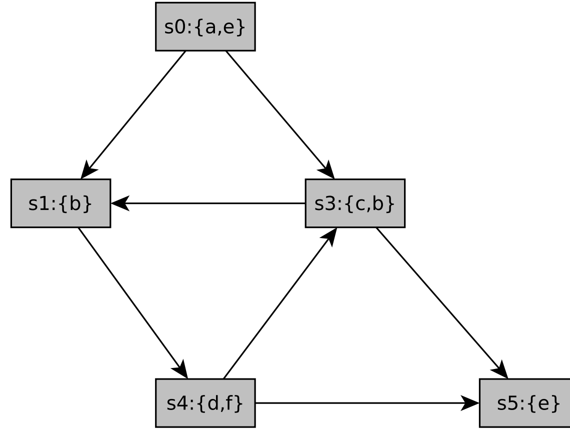


Figure 3.6: Model in Figure 3.1, with labels on the states.

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

$$\begin{aligned}\texttt{SAT}\,(\mathcal{M}, a \Rightarrow AX\ b) &= (S - \texttt{SAT}\,(\mathcal{M}, a)\,) \cup (S - \texttt{SAT}_{EX}\,(\neg b, \mathcal{M})\,) \\ &= (S - \{s_0\}) \cup (S - \{s_1, s_3, s_4\}) \\ &= \{s_1, s_3, s_4, s_5\} \cup \{s_0, s_5\} = \{s_0, s_1, s_3, s_4, s_5\}\end{aligned}$$

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

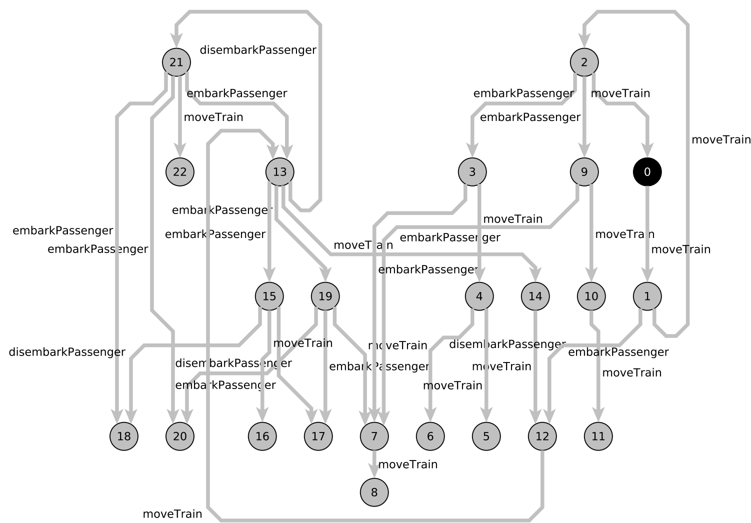
The missing piece to model check a graph grammar is how to transform the graph grammar state space (Definition 12) into a CTL (Definition 16). The main structural difference between both is that a graph grammar state space is labeled in the transitions, while the CTL model is labeled in the states, therefore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

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

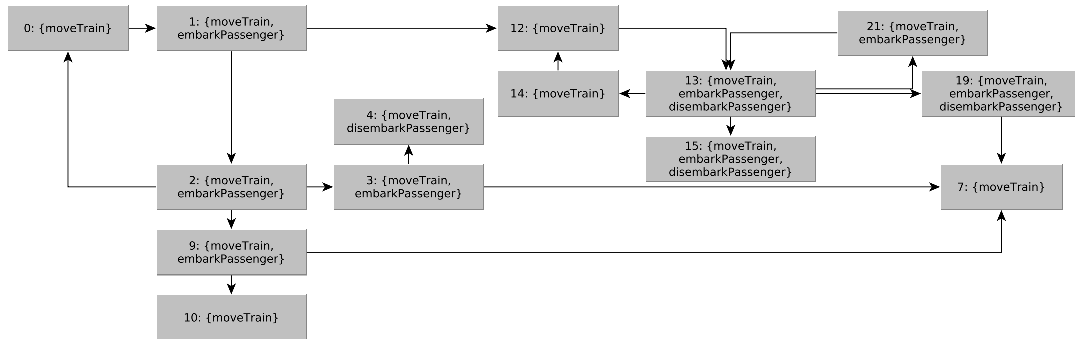
- $S_{\mathcal{M}} = S_{\mathbb{S}}$
- *for all* $[G]_{\equiv} \stackrel{p}{\Longrightarrow} [G']_{\equiv} \in T$, $[G]_{\equiv} \rightarrow [G']_{\equiv} \in\rightarrow$
- *for all* $[G]_{\equiv} \stackrel{p}{\Longrightarrow} [G']_{\equiv} \in T$, $p \in L([G]_{\equiv})$

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.

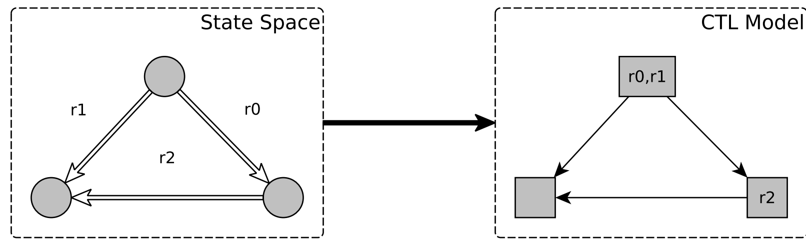


Figure 3.8: The conversion from a simple state space to a model.

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 ($c : K \overset{t}{\leftarrowtail} K \overset{t}{\rightarrowtail} K$, 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 φ and ϕ 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 conjuntction 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

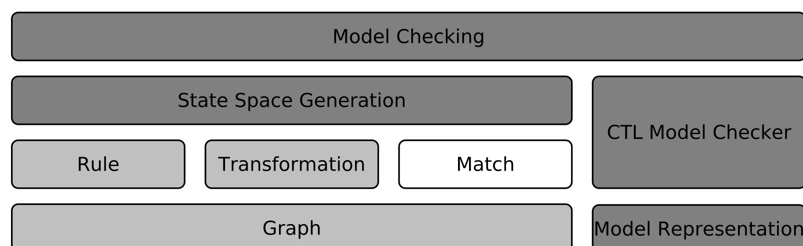


Figure 4.1: The system architecture's block diagram

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.

```
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 wheter a value is in a list, in Listing 4.2.

```
elem :: Eq a => a -> [a] -> Bool
elem _ []     = False
elem x (y:ys) = x == y || x `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 define 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 categorial 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: The `Monad` 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 getting 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 accept 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 successfuly, 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.

```
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 paramterized 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 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) => Edge b -> Digraph a b -> 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) => 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[1] allows us to detect violations of the (DPO) application conditions.

[1]The order is `removeEdge`, `removeNode`, `keepNode`, `keepEdge`, `addNode`, `addEdge`.

A typed graph is a pair of graphs, as in Listing 4.7.

```
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 `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 `GraphElement` type class and the two instances are presented in Listing 4.8.

```
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: `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 -> b`, which is an invalid type in the Hindley-Milner type system.

A graph morphism is depicted in Listing 4.9.

```
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.

```
type NodeAction a = (Maybe (Node a), Maybe (Node a))
type EdgeAction a = (Maybe (Edge a), Maybe (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 [] []
```

Listing 4.10: Rule's representation

Similar to morphisms, productions ($p : L \overset{l}{\leftarrowtail} K \overset{r}{\rightarrowtail} R$) 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.

```
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)
```

Listing 4.11: Auxiliary functions for `Rule`

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

```
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

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

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[2].

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 a 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.

```
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
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]
              -> [(Int, 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 wheter 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 -> D
   .TypedDigraph a b -> [D.Morphism]
```

Listing 4.15: Graph matching functions

The time and memory measurements for the algorithm with and without these conditions 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)
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] -> SSBuilder a b m ()
mkStateSpace 0  _ _ _ = return ()
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
```

```
i' <- putState i t (r', m)
if i' == 0
  then return ()
  else mkStateSpace (n - 1) i' t r
```

Listing 4.16: The state space building function

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` represent the logic operators $\neg$, $\wedge$, $\vee$ and $\Rightarrow$, 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.

$$\varphi_1 ::= A[\varphi_1\ U\ \varphi_1] \mid E[\varphi_1\ U\ \varphi_1] \mid \varphi_2 \rightarrow \varphi_1 \mid \varphi_2$$
$$\varphi_2 ::= \varphi_3 \&\& \varphi_2 \mid \varphi_3 || \varphi_2 \mid \varphi_3$$
$$\varphi_3 ::= \sim \varphi_3 \mid AX\varphi_3 \mid AF\varphi_3 \mid AG\varphi_3 \mid EX\varphi_3 \mid EF\varphi_3 \mid EG\varphi_3 \mid (\varphi_1) \mid p$$

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

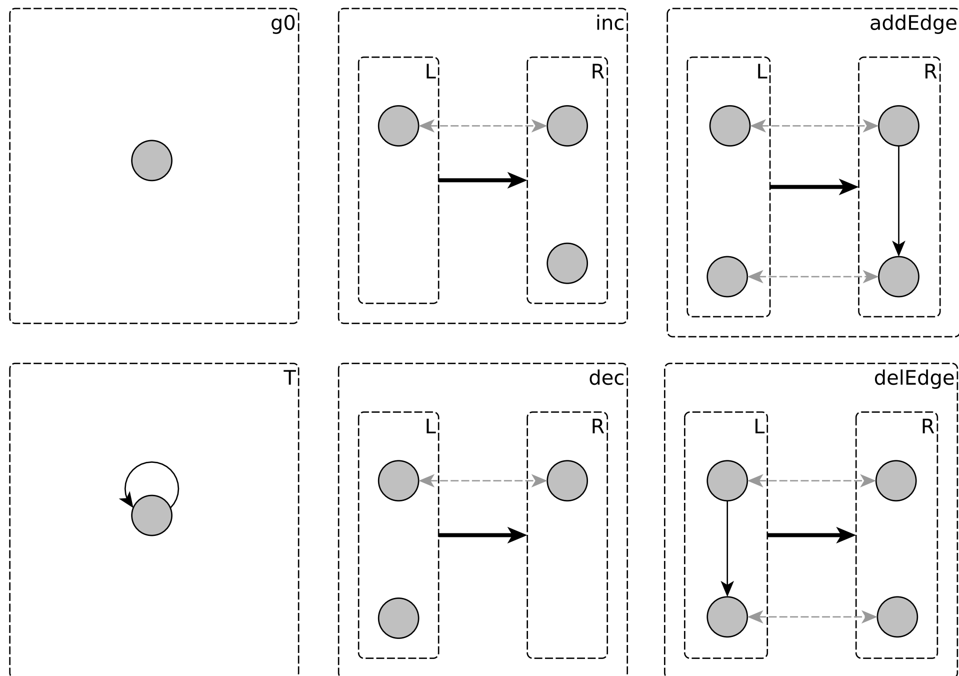


Figure 4.2: Grammar used in the experiments

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.

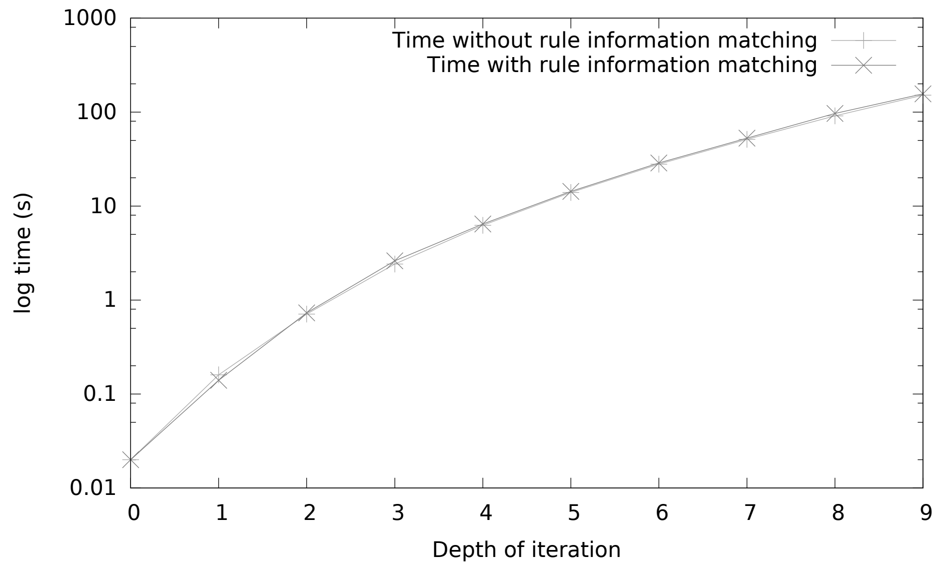
The mean time and memory usage are presented in the table below, and plotted in Figure 4.3.

Iteration size	**time (s)**		**memory (kb)**	
	rule	**no rule**	**rule**	**no rule**
10	0.02	0.02	5576	5608
20	0.16	0.14	8520	8760
30	0.71	0.73	16460	17332
40	2.42	2.62	33832	33784
50	6.24	6.46	62440	62536
60	13.98	14.32	119864	120844
70	27.95	28.74	159812	158796
80	51.24	52.78	269268	269296
90	91.14	96.74	347136	348156
100	151.38	156.5	564204	565232

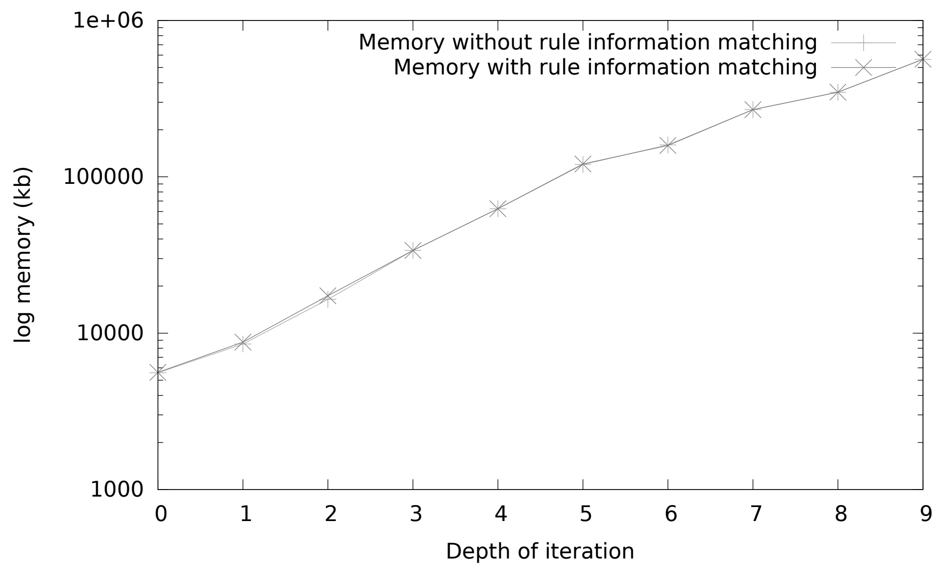
From the results, we conclude that more the rule information affects the execution time and memory usage slightly, as we expected. We would need to run experiments with more complex grammars to reveal how this modification results in faster execution of the grammar. Comparisson with other tools was not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



(a) Execution time



(b) Peak memory usage

Figure 4.3: Measurements for the experiment

5 CONCLUSIONS AND FUTURE WORK

In this monograph, we reviwed graph grammars and CTL model checking, and decribed the developement 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 analisys for first- and second-order graph grammars;
- Loading of grammars from well known file formats used in other tools;
- Single pushout and sesqui pushout tranformation;
- Matching algorithm with GPU acceleration (CHAKRAVARTY et al., 2011);
- User interface for graph and production edition.

REFERENCES

BLOCH, A. **Murphy's law**. [S.l.]: Penguin, 2003.

CHAKRAVARTY, M. M. et al. Accelerating Haskell array codes with multicore GPUs. In: DECLARATIVE ASPECTS OF MULTICORE PROGRAMMING. **Proceedings...** [S.l.: s.n.], 2011. p.3–14.

CLARKE, E. et al. Bounded model checking using satisfiability solving. **Formal Methods in System Design**, [S.l.], v.19, n.1, p.7–34, 2001.

CLARKE, E. M.; GRUMBERG, O.; PELED, D. **Model checking**. [S.l.]: MIT press, 1999.

DEFOE, D. **The political history of the devil**. [S.l.]: DA Talboys, 1840. v.10.

EHRIG, H. et al. **Fundamentals of Algebraic Graph Transformation**. 1st.ed. [S.l.]: Springer Publishing Company, Incorporated, 2010.

EHRIG, H.; PFENDER, M.; SCHNEIDER, H. J. Graph-grammars: an algebraic approach. In: Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on. **Anais...** IEEE, 1973. p.167–180.

EMERSON, E.; HALPERN, J. Y. Decision procedures and expressiveness in the temporal logic of branching time. **Journal of Computer and System Sciences**, [S.l.], v.30, n.1, p.1 – 24, 1985.

HALL, C. V. et al. Type Classes in Haskell. **ACM Trans. Program. Lang. Syst.**, New York, NY, USA, v.18, n.2, p.109–138, Mar. 1996.

HERRLICH, H.; STRECKER, G. E. **Category theory**. [S.l.]: Allyn and Bacon Boston, 1973.

HUTH, M. R. A.; RYAN, M. **Logic in Computer Science**: modelling and reasoning about systems. New York, NY, USA: Cambridge University Press, 2000.

HUTTON, G. Higher-order functions for parsing. **J. Funct. Program.**, [S.l.], v.2, n.3, p.323–343, 1992.

JONES, S. L. P. **Haskell 98 language and libraries**: the revised report. [S.l.]: Cambridge University Press, 2003.

KATOEN, J.-P.; BAIER, C. **Principles of model checking**. [S.l.]: The MIT Press, 2008.

KISELYOV, O.; JONES, S. P.; SHAN, C.-c. Fun with type functions. In: **Reflections on the Work of CAR Hoare**. [S.l.]: Springer, 2010. p.301–331.

LAMPORT, L. Proving the Correctness of Multiprocess Programs. **IEEE Trans. Softw. Eng.**, Piscataway, NJ, USA, v.3, n.2, p.125–143, Mar. 1977.

LEIJEN, D.; MEIJER, E. **Parsec**: direct style monadic parser combinators for the real world. [S.l.]: Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

LIPOVAČA, 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.

MARTIN, A. Adequate Sets of Temporal Connectives in {CTL}. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.52, n.1, p.21 – 31, 2002. EXPRESS'01, 8th International Workshop on Expressiveness in Concurrency (Satellite Event of {CONCUR} 2001).

MEHLHORN, K. **Graph Algorithms and NP-completeness**. New York, NY, USA: Springer-Verlag New York, Inc., 1984.

MILNER, R. A theory of type polymorphism in programming. **Journal of computer and system sciences**, [S.l.], v.17, n.3, p.348–375, 1978.

MOGGI, E. Notions of computation and monads. **Information and computation**, [S.l.], v.93, n.1, p.55–92, 1991.

O'SULLIVAN, B.; GOERZEN, J.; STEWART, D. B. **Real World Haskell**: code you can believe in. [S.l.]: " O'Reilly Media, Inc.", 2008.

PNUELI, A. The temporal logic of programs. In: FOUNDATIONS OF COMPUTER SCIENCE, 1977., 18TH ANNUAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 1977. p.46–57.

PRIOR, A. **Time and Modality**. [S.l.]: Oxford, 1957.

RENSINK, A. Explicit State Model Checking for Graph Grammars. In: DEGANO, P.; NICOLA, R.; MESEGUER, J. (Ed.). **Concurrency, Graphs and Models**. Berlin, Heidelberg: Springer-Verlag, 2008. p.114–132.

RIBEIRO, L. et al. **Grupo de Verificação, Validação e Teste de Sistemas Computacionais**. 2014.

RUDOLF, M. Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In: EHRIG, H. et al. (Ed.). **Theory and Application of Graph Transformations**. [S.l.]: Springer Berlin Heidelberg, 2000. p.238–251. (Lecture Notes in Computer Science, v.1764).

SCHRIJVERS, T. et al. Type Checking with Open Type Functions. **SIGPLAN Not.**, New York, NY, USA, v.43, n.9, p.51–62, Sept. 2008.

WADLER, P. Monads for functional programming. In: BROY, M. (Ed.). **Program Design Calculi**. [S.l.]: Springer Berlin Heidelberg, 1993. p.233–264. (NATO ASI Series, v.118).

WIESE, R.; EIGLSPERGER, M.; KAUFMANN, M. **yfiles—visualization and automatic layout of graphs**. [S.l.]: Springer, 2004.

AppendixA

A.1 GraphGrammar

```
1  module GraphGrammar ( module GraphGrammar.Graph
2                      , module GraphGrammar.StateSpace
3                      , module GraphGrammar.Rule
4                      , runSerializedGrammar
5                      ) where
6
7  import Control.Monad.IO.Class
8
9  import GraphGrammar.Graph
10 import GraphGrammar.Rule
11 import GraphGrammar.Match
12 import GraphGrammar.Transformation
13 import GraphGrammar.StateSpace
14 import GraphGrammar.Serialized
15
16 unserializeString :: MonadIO m => String -> m (Serialized String
       String)
17 unserializeString s = do system <- liftIO $ readFile s
18                          return $ unserialize system
19
20 runSerializedGrammar :: MonadIO m => Int -> String -> m (StateSpace
       String String)
21 runSerializedGrammar l s = do (Serialized graphs rules) <-
       unserializeString s
22                               runStateSpace l (head graphs) rules
```

A.2 GraphGrammar.Graph

```
1  {-# LANGUAGE TypeFamilies #-}
2  module GraphGrammar.Graph
3          ( Edge (..)
4          , Node (..)
5          , Digraph (..)
6          , TypedDigraph (..)
7          , GraphElement (..)
8          , Morphism (..)
9          , TGraph
10         , empty
11         , null
12         , fromLists
13         , node
```

```
14          , edge
15          , addNode
16          , addEdge
17          , removeNode
18          , removeEdge
19          , keepNode
20          , keepEdge
21          , insNode
22          , insEdge
23          , delNode
24          , delEdge
25          , nodes
26          , edges
27          , findNode
28          , source
29          , target
30          , hasEdge
31          , sourceID
32          , targetID
33          , srcType
34          , tarType
35          ) where
36
37  import Control.Monad
38
39  import Prelude hiding (null)
40
41  import Data.IntMap (IntMap)
42  import qualified Data.IntMap    as IM
43  import qualified Data.List      as L
44  import Assorted.PrettyPrint
45
46  data Edge a = Edge Int (Int, Int) Int a deriving (Show, Read)
47  instance Eq (Edge a) where
48          Edge lid _ _ _ == Edge gid _ _ _ = lid == gid
49
50  data Node a = Node Int Int a deriving (Show, Read)
51  instance Eq (Node a) where
52          Node lid _ _ == Node gid _ _ = lid == gid
53
54  class GraphElement a where
55      type Payload a :: *
56      payload :: a -> Payload a
57      elemId :: a -> Int
58      typeId :: a -> Int
59
60  instance GraphElement (Node a) where
61      type Payload (Node a) = a
62      payload (Node _ _ q) = q
63      elemId (Node i _ _) = i
64      typeId (Node _ t _) = t
65
66  instance GraphElement (Edge a) where
67      type Payload (Edge a) = a
68      payload (Edge _ _ _ q) = q
69      elemId (Edge i _ _ _) = i
70      typeId (Edge _ _ t _) = t
71
```

```
72  data Digraph a b = Digraph (IntMap (Node a)) (IntMap (Edge b))
        deriving (Show, Read, Eq)
73
74  data TypedDigraph a b = TypedDigraph (Digraph a b) (TGraph a b)
75          deriving (Show, Read, Eq)
76
77  type TGraph a b = Digraph a b
78
79  empty :: Digraph a b
80  empty = Digraph (IM.empty) (IM.empty)
81
82  null :: Digraph a b -> Bool
83  null (Digraph nm em) =
84          IM.null nm && IM.null em
85
86  fromLists :: [Node a] -> [Edge b] -> Digraph a b
87  fromLists ns es = Digraph (fromElementList ns) (fromElementList es)
88
89
90  fromElementList :: GraphElement a => [a] -> IntMap a
91  fromElementList = IM.fromList . map (\x -> (elemId x, x))
92
93  node :: Int -> Digraph a b -> Maybe (Node a)
94  node i (Digraph n _) = IM.lookup i n
95
96  edge :: Int -> Digraph a b -> Maybe (Edge b)
97  edge i (Digraph _ e) = IM.lookup i e
98
99  addNode :: (Monad m) => Node a -> Digraph a b -> m (Digraph a b)
100 addNode n@(Node id _ _) g@(Digraph nm em) =
101         if id `IM.member` nm
102                 then fail $ "addNode: node " ++ show id ++ " already
                         in digraph"
103                 else return $ Digraph (IM.insert id n nm) em
104
105 addEdge :: (Monad m) => Edge b -> Digraph a b -> m (Digraph a b)
106 addEdge e@(Edge id (s, t) _ _) g@(Digraph nm em)
107         | id `IM.member` em =
108                 fail $ "addEdge: edge " ++ show id ++ " already in
                       digraph"
109         | s `IM.member` nm && t `IM.member` nm =
110                 return $ Digraph nm (IM.insert id e em)
111         | otherwise =
112                 fail $ "addEdge: edge points to nodes not found in
                       digraph"
113
114 removeNode :: (Monad m) => Node a -> Digraph a b -> m (Digraph a b)
115 removeNode n@(Node id _ _) g@(Digraph nm em)
116         | id `IM.notMember` nm =
117                 fail $ "removeNode: node " ++ show id ++ " not in
                       digraph"
118         | IM.fold
119                 (\(Edge eid (s, t) _ _) acc -> acc || s == id || t
                       == id)
120                 False em =
121                 fail $ "removeNode: node " ++ show id ++ " has some
                       edge pointing to it"
122         | otherwise =
```

```
123                 return $ Digraph (IM.delete id nm) em
124
125 removeEdge :: (Monad m) => Edge b -> Digraph a b -> m (Digraph a b)
126 removeEdge e@(Edge id _ _ _) g@(Digraph nm em) =
127         if id `IM.member` em
128                 then return $ Digraph nm (IM.delete id em)
129                 else fail $ "removeEdge: edge " ++ show id ++ " not
                        in digraph"
130
131 keepNode :: (Monad m) => Node a -> Digraph a b -> m (Digraph a b)
132 keepNode (Node nid _ _) g@(Digraph ns es) =
133         if nid `IM.member` ns
134                 then return g
135                 else fail $ "keepNode: node " ++ show nid ++ " doesn
                        't exist"
136
137 keepEdge :: (Monad m) => Edge b -> Digraph a b -> m (Digraph a b)
138 keepEdge (Edge eid _ _ _) g@(Digraph ns es) =
139         if eid `IM.member` es
140                 then return g
141                 else fail $ "keepEdge: edge " ++ show eid ++ " doesn
                        't exist"
142
143 findNode :: Int -> Digraph a b -> Maybe (Node a)
144 findNode id (Digraph nm _) =
145         IM.lookup id nm
146
147 nodes :: Digraph a b -> [(Node a)]
148 nodes (Digraph nm _) = map snd $ IM.toList nm
149
150 edges :: Digraph a b -> [(Edge b)]
151 edges (Digraph _ em) = map snd $ IM.toList em
152
153 source :: Edge b -> Digraph a b -> Node a
154 source e d =
155         let (Just n) = findNode (sourceID e) d
156         in n
157
158 target :: Edge b -> Digraph a b -> Node a
159 target e d =
160         let (Just n ) = findNode (targetID e) d
161         in n
162
163 sourceID :: Edge b -> Int
164 sourceID (Edge _ (src, _) _ _) = src
165
166 targetID :: Edge b -> Int
167 targetID (Edge _ (_, tar) _ _) = tar
168
169 hasEdge :: TypedDigraph a b -> Node a -> Bool
170 hasEdge (TypedDigraph dg _) n =
171         let
172                 nid = elemId n
173                 found = L.find (\(Edge _ (s, t) _ _) -> s == nid ||
                        t == nid) $ edges dg
174         in case found of
175                 Just _ -> True
176                 Nothing -> False
```

```
177
178 nodePayload :: Node a -> a
179 nodePayload (Node _ _ p) = p
180
181 insNode :: Node a -> Digraph a b -> Digraph a b
182 insNode n@(Node id _ _) g@(Digraph nm em) =
183         if id `IM.member` nm
184                 then g
185                 else Digraph (IM.insert id n nm) em
186
187 insEdge :: Edge b -> Digraph a b -> Digraph a b
188 insEdge e@(Edge id (s, t) _ _) g@(Digraph nm em)
189         | id `IM.member` em =
190                 g
191         | s `IM.member` nm && t `IM.member` nm =
192                 Digraph nm (IM.insert id e em)
193         | otherwise =
194                 g
195
196 delNode :: Node a -> Digraph a b -> Digraph a b
197 delNode n@(Node id _ _) g@(Digraph nm em)
198         | id `IM.notMember` nm =
199                 g
200         | IM.fold
201                 (\(Edge eid (s, t) _ _) acc -> acc || s == id || t
                        == id)
202                 False em =
203                 g
204         | otherwise =
205                 Digraph (IM.delete id nm) em
206
207 delEdge :: Edge b -> Digraph a b -> Digraph a b
208 delEdge e@(Edge id _ _ _) g@(Digraph nm em) =
209         if id `IM.member` em
210                 then Digraph nm (IM.delete id em)
211                 else g
212
213 findNodeType :: Int -> TypedDigraph a b -> Maybe Int
214 findNodeType id td@(TypedDigraph (Digraph nm em) _) =
215         let n = IM.lookup id nm
216         in case n of
217                 Nothing -> Nothing
218                 Just (Node _ tid _) -> Just tid
219
220 srcType :: Edge b -> TypedDigraph a b -> Maybe Int
221 srcType (Edge _ (s, _) _ _) l =
222         findNodeType s l
223
224 tarType :: Edge b -> TypedDigraph a b -> Maybe Int
225 tarType (Edge _ (_, t) _ _) l =
226         findNodeType t l
227
228 type Morphism = ([(Int, Int)], [(Int, Int)])
```

A.3 GraphGrammar.Rule

```
1 module GraphGrammar.Rule ( Action (..)
2                          , Rule (..)
```

```
3                               , MatchType (..)
4                             , emptyRule
5                             , applyRule
6                             , left
7                               ) where
8
9  import Control.Monad
10
11 import Data.Maybe
12 import qualified Data.List as L
13 import Data.IntMap hiding (map, filter)
14
15 import GraphGrammar.Graph
16
17 import Assorted.PrettyPrint
18
19 type Action a     = (Maybe a, Maybe a)
20 type NodeAction a = Action (Node a)
21 type EdgeAction a = Action (Edge a)
22
23 data MatchType = Normal | Mono | Epi | Iso
24         deriving (Show, Read, Eq)
25
26 data Rule a b = Rule String MatchType [NodeAction a] [EdgeAction b]
27     deriving (Show,Read)
28
29 emptyRule = Rule [] Normal [] []
30
31 addNodeAction :: Node a -> Node a -> Rule a b -> Rule a b
32 addNodeAction ln rn m@(Rule n mt nal eal) =
33     if (Just ln, Just rn) `L.notElem` nal
34         then Rule n mt ((Just ln, Just rn) : nal) eal
35         else m
36
37
38 addEdgeAction :: Edge b -> Edge b -> Rule a b -> Rule a b
39 addEdgeAction le re m@(Rule n mt nal eal) =
40     Rule n mt nal ((Just le, Just re) : eal)
41
42
43 nodeAction :: (Monad m, Eq a) => NodeAction a -> (Digraph a b -> m (
       Digraph a b))
44 nodeAction (Nothing, Just n) = addNode n
45 nodeAction (Just n, Nothing) = removeNode n
46 nodeAction (Just n, Just n') = if n /= n'
47     then const $ fail "Node transformation is unhandled"
48     else keepNode n
49 nodeAction (Nothing, Nothing) = return
50
51 edgeAction :: (Monad m, Eq b) => EdgeAction b -> (Digraph a b -> m (
       Digraph a b))
52 edgeAction (Nothing, Just e) = addEdge e
53 edgeAction (Just e, Nothing) = removeEdge e
54 edgeAction (Just e, Just e') = if e /= e'
55     then const $ fail "Edge transformation is unhandled"
56     else keepEdge e
57 edgeAction (Nothing, Nothing) = return
58
```

```
59 addAction (Nothing, Just t) = True
60 addAction _ = False
61
62 removeAction (Just s, Nothing) = True
63 removeAction _ = False
64
65 keepAction (Just s, Just t) = True
66 keepAction _ = False
67
68 left :: (Eq a, Eq b) => Rule a b -> TGraph a b -> TypedDigraph a b
69 left (Rule n mt nr er) t = let
70     f e = fst e /= Nothing
71     ns = toElemList fst $ filter f nr
72     es = toElemList fst $ filter f er
73     in TypedDigraph (fromLists ns es) t
74
75 right :: (Eq a, Eq b) => Rule a b -> TGraph a b -> TypedDigraph a b
76 right (Rule n mt nr er) t = let
77     f e = snd e /= Nothing
78     ns = toElemList snd $ filter f nr
79     es = toElemList snd $ filter f er
80     in TypedDigraph (fromLists ns es) t
81
82 glue :: (Eq a, Eq b) => Rule a b -> TGraph a b -> TypedDigraph a b
83 glue (Rule n mt nr er) t = let
84     f e = snd e == fst e
85     ns = toElemList fst $ filter f nr
86     es = toElemList fst $ filter f er
87     in TypedDigraph (fromLists ns es) t
88
89 toElemList :: (Action a -> Maybe a) -> [Action a] -> [a]
90 toElemList f = map (fromJust . f)
91
92 actionSet :: (Monad m, Eq a, Eq b) => Rule a b -> [Digraph a b -> m
       (Digraph a b)]
93 actionSet (Rule n mt na ea) = let
94     nodeActions f = map nodeAction . filter f
95     edgeActions f = map edgeAction . filter f
96     knSet = nodeActions keepAction na
97     keSet = edgeActions keepAction ea
98     anSet = nodeActions addAction na
99     aeSet = edgeActions addAction ea
100    dnSet = nodeActions removeAction na
101    deSet = edgeActions removeAction ea
102    in deSet ++ dnSet ++ knSet ++ keSet ++ anSet ++ aeSet
103
104 applyActions :: Monad m => [a -> m a] -> a -> m a
105 applyActions as g = foldM (\g f -> f g) g as
106
107 applyRule :: (Monad m, Eq a, Eq b) => Rule a b -> TypedDigraph a b
       -> m (TypedDigraph a b)
108 applyRule m tg = let (TypedDigraph g t) = tg
109                      actions = actionSet m
110                  in do g' <- applyActions actions g
111                        return $ TypedDigraph g' t
```

A.4 GraphGrammar.Transformation

```
1  module GraphGrammar.Transformation ( rewrite ) where
2
3  import Data.Maybe
4  import Data.List
5  import Data.IntMap (IntMap,keys,fromList)
6
7  import GraphGrammar.Graph
8  import GraphGrammar.Rule
9  import GraphGrammar.Match (findMatches)
10
11 import Assorted.PrettyPrint
12
13 import Control.Monad
14
15 rewrite :: (Monad m, Eq a, Eq b) => Rule a b -> TypedDigraph a b ->
      Morphism -> m (TypedDigraph a b)
16 rewrite rule tGraph match = let (TypedDigraph graph _) = tGraph
17                                 ns = renameStart $ nodes graph
18                                 es = renameStart $ edges graph
19                                 renamedRule = rename ns es rule
                                      match
20                             in applyRule renamedRule tGraph
21
22 renameStart :: GraphElement e => [e] -> Int
23 renameStart es = 1 + (maximum $ map elemId es)
24
25 lookup' i = maybe i id . lookup i
26
27 renameNode :: [(Int, Int)] -> Node a -> Node a
28 renameNode nm (Node id t p) = Node (lookup' id nm) t p
29
30 renameEdge nm em (Edge id st t p) = Edge (lookup' id em)
31     (double (\x -> lookup' x nm) st) t p
32
33 renameAction :: GraphElement e => (e -> e) -> Action e -> Action e
34 renameAction f = double (liftM f)
35
36 double :: (a -> b) -> (a, a) -> (b, b)
37 double f (x, y) = (f x, f y)
38
39 rename :: (Eq a, Eq b) => Int -> Int -> Rule a b -> Morphism -> Rule
       a b
40 rename ns es (Rule n mt nr er) (nm, em) = let
41     elemIds :: (Eq e, GraphElement e) => [e] -> [Int]
42     elemIds = map elemId
43
44     addElements :: (Eq e, GraphElement e) => [Action e] -> [e]
45     addElements = map (fromJust . snd) . filter ((== Nothing) . fst)
46
47     idMap :: (Eq e, GraphElement e) => [Action e] -> [Int] -> [(Int,
          Int)]
48     idMap xs ys = zip (elemIds $ addElements xs) ys
49
50     nodeIdMap = (idMap nr [ns..]) ++ nm
51     edgeIdMap = (idMap er [es..]) ++ em
52
53     nodeRename = renameAction (renameNode nodeIdMap)
54     edgeRename = renameAction (renameEdge nodeIdMap edgeIdMap)
```

```
55     in Rule n mt (map nodeRename nr) (map edgeRename er)
```

A.5 GraphGrammar.Match

```
1  module GraphGrammar.Match
2          ( findMatches
3          , findMatchesR
4          , isSurjective
5          , findIsoMorphisms
6      , isIsomorphic
7          )
8          where
9
10 import Control.Monad -- foldM
11 import Data.Maybe
12 import qualified GraphGrammar.Graph as D
13 import qualified GraphGrammar.Rule as D
14 import qualified Data.IntMap as IM
15 import qualified Data.List as L
16 import qualified Data.Set as S
17
18 type MapSet = (S.Set (Int, Int), S.Set (Int, Int))
19
20 findMatches :: D.MatchType -> D.TypedDigraph a b -> D.TypedDigraph a
       b -> [D.Morphism]
21 findMatches mt l g =
22         findMatchesR D.emptyRule mt l g
23
24 findMatchesR :: D.Rule a b -> D.MatchType -> D.TypedDigraph a b -> D
     .TypedDigraph a b -> [D.Morphism]
25 findMatchesR r mt l g =
26         let matches = matchGraphs r mt l g
27         in map (\(nm, em) -> (S.toList nm, S.toList em)) matches
28
29 type EdgeCondition b = D.Edge b -> Bool
30
31 edgeTypeCondGen :: D.Edge b -> EdgeCondition b
32 edgeTypeCondGen le = (\ge -> D.typeId le == D.typeId ge)
33
34 srcTypeCondGen :: D.TypedDigraph a b -> D.Edge b -> D.TypedDigraph a
       b -> EdgeCondition b
35 srcTypeCondGen l le g =
36         (\ge -> D.srcType le l == D.srcType ge g)
37
38 tarTypeCondGen :: D.TypedDigraph a b -> D.Edge b -> D.TypedDigraph a
       b -> EdgeCondition b
39 tarTypeCondGen l le g =
40         (\ge -> D.tarType le l == D.tarType ge g)
41
42 srcIDCondGen
43         :: D.Edge b
44         -> MapSet
45         -> EdgeCondition b
46 srcIDCondGen le m@(nmatches, _) =
47         (\ge ->
48                 let
49                         lsrc = D.sourceID le
50                         gsrc = D.sourceID ge
```

```
51                          matched = L.find (\(s, t) -> s == lsrc) $ S.
                               toList nmatches
52                  in case matched of
53                          Just (_, n) -> gsrc == n
54                          otherwise -> True)
55
56          :: D.Edge b
57          -> MapSet
58          -> EdgeCondition b
59  tarIDCondGen le m@(nmatches, _) =
60          (\ge ->
61                  let ltar = D.targetID le
62                      gtar = D.targetID ge
63                      matched = L.find (\(s, t) -> s == ltar) $ S.
                           toList nmatches
64                  in case matched of
65                          Just (_, n) -> gtar == n
66                          otherwise -> True)
67
68  loopCondGen :: D.Edge b -> EdgeCondition b
69  loopCondGen le =
70          (\ge ->
71                  let
72                          lsrc = D.sourceID le
73                          ltar = D.targetID le
74                          gsrc = D.sourceID ge
75                          gtar = D.targetID ge
76                  in if lsrc == ltar
77                          then gsrc == gtar
78                          else True)
79
80  generateEdgeConds
81          :: D.TypedDigraph a b
82          -> D.Edge b
83          -> D.TypedDigraph a b
84          -> MapSet
85          -> [EdgeCondition b]
86  generateEdgeConds l le g m =
87          edgeTypeCondGen le      :
88          srcIDCondGen le m :
89          tarIDCondGen le m :
90          loopCondGen le :
91          []
92
93  processEdge :: [EdgeCondition b] -> D.Edge b -> Bool
94  processEdge cl e =
95          L.foldr (\c acc -> (c e) && acc) True cl
96
97  type NodeCondition a = D.Node a -> Bool
98
99  nodeTypeCondGen :: D.Node a -> NodeCondition a
100 nodeTypeCondGen ln =
101         (\n -> D.typeId ln == D.typeId n)
102
103 danglingCondGen ::
104         D.Rule a b
105         -> D.TypedDigraph a b
106         -> D.Node a
```

```
107         -> NodeCondition a
108 danglingCondGen r g ln =
109         if toBeDeleted r (D.elemId ln)
110                 then (\gn -> not $ D.hasEdge g gn)
111                 else (\gn -> True)
112 
113 delCondGen ::
114         D.Rule a b
115         -> D.Node a
116         -> MapSet
117         -> NodeCondition a
118 delCondGen r ln m =
119         (\gn -> (not $ isMapped gn m) ||
120                 (toBeDeleted r (D.elemId ln) == mappedToDel r m gn))
121 
122 isMapped :: D.Node a -> MapSet -> Bool
123 isMapped gn (nmaps, _) =
124         let found = S.filter (\(_, gnode) -> gnode == D.elemId gn)
            nmaps
125         in not $ S.null found
126 
127 mappedToDel :: D.Rule a b -> MapSet -> D.Node a -> Bool
128 mappedToDel r (nmaps, _) n =
129         let nmap = S.filter (\(lnid, gnid) ->
130                         gnid == nid && toBeDeleted r lnid
131                         ) nmaps
132         in not $ S.null nmap
133         where nid = D.elemId n
134 
135 toBeDeleted :: D.Rule a b -> Int -> Bool
136 toBeDeleted r@(D.Rule _ _ nal _) nid =
137         let naction = L.find (\na ->
138                 case na of
139                         (Just ln, _) -> D.elemId ln == nid
140                         otherwise    -> False
141                 ) nal
142         in case naction of
143                 Just (_, Nothing) -> True
144                 otherwise         -> False
145 
146 generateConds ::
147         D.Rule a b
148         -> D.TypedDigraph a b
149         -> D.Node a
150         -> D.TypedDigraph a b
151         -> MapSet
152         -> [NodeCondition a]
153 generateConds r l ln g m =
154         nodeTypeCondGen ln      :
155         delCondGen r ln m       :
156         danglingCondGen r g ln  :
157         []
158 
159 processNode :: [NodeCondition a] -> D.Node a -> Bool
160 processNode cl n =
161         L.foldr (\c acc -> (c n) && acc) True cl
162 
163 mapGraphs
```

```
164         :: D.Rule a b
165         -> D.MatchType
166         -> D.TypedDigraph a b   -- ^ @l@, the "left side" graph
167         -> (MapSet, D.TypedDigraph a b, [D.Edge b], [D.Node a]) -- ^
                @m@, what already got mapped
168         -> [(MapSet, D.TypedDigraph a b, [D.Edge b], [D.Node a])]
169 mapGraphs _ mt _ ml@((nmap, emap), D.TypedDigraph dg@(D.Digraph gnm
        gem) _, [], []) =
170         case mt of
171         D.Epi -> let
172                 gMappedNodes = S.fold (\(ln, gn) acc -> S.insert gn
                        acc) S.empty nmap
173                 gMappedEdges = S.fold (\(le, ge) acc -> S.insert ge
                        acc) S.empty emap
174                 in
175                 if S.size gMappedNodes == IM.size gnm && S.size
                        gMappedEdges == IM.size gem
176                         then [ml]
177                         else []
178         D.Iso -> if D.null dg
179                         then [ml]
180                         else []
181         otherwise -> [ml]
182 mapGraphs r mt l (m@(nmatch, ematch),
183         g@(D.TypedDigraph dg@(D.Digraph gnm gem) tg),
184         (le:les), lns) =
185         let
186                 conds = generateEdgeConds l le g m
187                 edgeList = D.edges dg
188                 candidates = filter (processEdge conds) $ edgeList
189                 newMapSets = fmap
190                         (\ge ->
191                                 let
192                                         sid = D.sourceID ge
193                                         tid = D.targetID ge
194                                         eid = D.elemId ge
195                                         newLNodeList = L.filter (\n
                                                ->
196                                                 let nid = D.elemId n
197                                                 in (nid /= D.
                                                        sourceID le) &&
                                                        (nid /= D.
                                                        targetID le)
198                                                 ) lns
199                                 in
200                                 ((S.insert (D.sourceID le, sid) $
201                                   S.insert (D.targetID le, tid) $
202                                   nmatch,
203                                   S.insert (D.elemId le, eid) ematch
                                        ),
204                                 if mt == D.Normal || mt == D.Epi
205                                 then g
206                                 else (D.TypedDigraph (D.Digraph (IM.
                                        delete sid $ IM.delete tid gnm)
207
```

```
208                                                    tg),
209                               les,
210                               newLNodeList)
211                       ) candidates
212         in newMapSets >>= mapGraphs r mt l
213 mapGraphs r mt l (m@(nmatch, ematch),
214         g@(D.TypedDigraph dg@(D.Digraph gnm gem) tg),
215         [], (ln:lns)) =
216         let
217                 conds = (generateConds r l ln g m)
218                 candidates = filter (processNode conds) $ D.nodes dg
219                 newMapSets = fmap
220                         (\gn ->
221                                 let gid = D.elemId gn
222                                 in
223                                 ((S.insert (D.elemId ln, gid) nmatch
                                    , ematch),
224                                  if mt == D.Normal || mt == D.Epi
225                                  then g
226                                  else D.TypedDigraph (D.Digraph (IM.
                                    delete gid gnm) gem) tg,
227                                  [],
228                                  lns)
229                         ) candidates
230         in newMapSets >>= mapGraphs r mt l
231
232 matchGraphs :: D.Rule a b -> D.MatchType -> D.TypedDigraph a b -> D.
    TypedDigraph a b -> [MapSet]
233 matchGraphs r mt l@(D.TypedDigraph dl _) g =
234         map (\(m, _, _, _) -> m ) $
235                 mapGraphs r mt l ((S.empty, S.empty), g, D.edges dl,
                     D.nodes dl)
236
237 isSurjective :: D.TypedDigraph a b -> MapSet -> Bool
238 isSurjective (D.TypedDigraph (D.Digraph gnm gem) _) m@(nmaps, emaps)
     =
239         IM.size gnm == S.size nmaps && IM.size gem == S.size emaps
240
241 findIsoMorphisms :: D.TypedDigraph a b -> D.TypedDigraph a b -> [D.
    Morphism]
242 findIsoMorphisms l@(D.TypedDigraph (D.Digraph lnm lem) _) g@(D.
    TypedDigraph (D.Digraph gnm gem) _) =
243         if IM.size lnm /= IM.size gnm ||
244            IM.size lem /= IM.size gem
245         then []
246         else findMatchesR D.emptyRule D.Iso l g
247
248 isIsomorphic :: D.TypedDigraph a b -> D.TypedDigraph a b -> Bool
249 isIsomorphic a b = findIsoMorphisms a b /= []
```

A.6 GraphGrammar.StateSpace

```
1  {-# LANGUAGE DoAndIfThenElse #-}
2  module GraphGrammar.StateSpace ( StateSpace (..)
3                                 , runStateSpace
4                                 ) where
5
6  import Prelude
7
8  import GraphGrammar.Builder.Graph
9  import GraphGrammar.Graph
10 import GraphGrammar.Match
11 import GraphGrammar.Rule
12 import GraphGrammar.Transformation
13 import Assorted.PrettyPrint
14
15 import Data.List
16 import Data.Maybe
17 import Data.IntMap (fromList,toList,elems)
18 import qualified Data.IntMap as IM
19
20 import Control.Monad
21 import Control.Monad.IO.Class
22
23
24 type StateSpace a b = Digraph (TypedDigraph a b) (Rule a b, Morphism
     )
25 type SSBuilder a b m r = GraphBuilder (TypedDigraph a b) (Rule a b,
     Morphism)  m r
26
27 tpg (TypedDigraph _ t) = t
28
29 ns (Digraph n _) = n
30
31 onSnd :: (b -> c) -> (a, b) -> (a, c)
32 onSnd f (x, y) = (x, f y)
33
34 runStateSpace :: (Eq a, Eq b, MonadIO m) => Int
35   -> TypedDigraph a b -> [Rule a b] -> m (StateSpace a b)
36 runStateSpace n g r = buildGraph $ do
37   i <- createNode g 1
38   mkStateSpace n i g r
39
40
41 mkStateSpace :: (Eq a, Eq b, MonadIO m) => Int -> Int
42   -> TypedDigraph a b -> [Rule a b] -> SSBuilder a b m ()
43 mkStateSpace 0  _ _ _ = return ()
44 mkStateSpace n i g r = do
45   forM_ r $ \r' -> do
46     let (Rule _ mt _ _) = r'
47     forM_ (findMatches mt (left r' $ tpg g) g) $ \m -> do
48       let t' = rewrite r' g m
49       if t' == Nothing
50         then return ()
51         else do
52           let (Just t) = t'
53           i' <- putState i t (r', m)
54           if i' == 0
```

```
55              then return ()
56              else mkStateSpace (n - 1) i' t r
57
58 putState :: (Eq a, Eq b, MonadIO m) => Int
59   -> TypedDigraph a b -> (Rule a b, Morphism) -> SSBuilder a b m Int
60 putState i g e = do
61   states <- liftM (map (onSnd payload) . toList . ns) getG
62   let isos = filter snd $ map (onSnd (isIsomorphic g)) states
63   case isos of
64       [] -> do i' <- createNode g 1
65                createEdge e 1 (i, i')
66                return i'
67       [(i', True)] -> do createEdge e 1 (i, i')
68                          return 0
69       _ -> error $ "Undefined case for " ++ show isos
```

A.7 GraphGrammar.Serialized

```
1  module GraphGrammar.Serialized where
2
3  import GraphGrammar.Graph
4  import GraphGrammar.Rule
5
6  data Serialized a b = Serialized [TypedDigraph a b] [Rule a b]
       deriving (Show, Read)
7
8  serialize :: (Show a, Show b, Read a, Read b) => [TypedDigraph a b]
       -> [Rule a b] -> String
9  serialize gs = show . Serialized gs
10
11 unserialize :: (Show a, Show b, Read a, Read b) => String ->
       Serialized a b
12 unserialize = read
```

A.8 GraphGrammar.GML

```
1  module GraphGrammar.GML where
2
3  data N = N Int String deriving (Show)
4  data E = E Int Int String deriving (Show)
5  data Document = Document [N] [E] deriving (Show)
6
7  newDocument :: Document
8  newDocument = Document [] []
9
10 newNode :: (Show l) => Int -> l -> Document -> Document
11 newNode i l (Document ns es) = Document ((N i (show l)):ns) es
12
13 newEdge :: (Show l) => Int -> Int -> l -> Document -> Document
14 newEdge s t l (Document ns es) = Document ns ((E s t (show l)):es)
15
16 strN (N i l) = unlines [ "node [", "id " ++ show i, "label " ++ l, "
       ]" ]
17 strE (E s t l) = unlines [ "edge [", "source " ++ show s, "target "
       ++ show t, "label " ++ l, "]" ]
18
19 writeDocument :: Document -> String -> IO ()
```

```
20 writeDocument (Document ns es) n = let content = ["graph [", "
       directed 1"] ++ map strN ns ++ map strE es ++ ["]"]
21                                    in writeFile n $ unlines content
```

A.9 GraphGrammar.Builder.Graph

```
1  {-# LANGUAGE GADTs #-}
2  module GraphGrammar.Builder.Graph ( GraphBuilderT (..)
3                                    , GraphBuilder (..)
4                                    , buildGraphT
5                                    , buildGraphTFrom
6                                    , buildGraph
7                                    , buildGraphFrom
8                                    , createNode
9                                    , createEdge
10                                   , getG
11                                   , putG
12                                   , getS
13                                   , putS
14                                   ) where
15 import Prelude hiding (null,lookup,elem)
16
17 import Control.Monad.State
18 import Control.Monad.Identity
19
20 import Data.IntMap as IM hiding (empty,map,filter,(\\))
21 import Data.Maybe
22 import Data.List hiding (lookup,null)
23
24 import GraphGrammar.Graph hiding (nodes,edges,node,edge)
25 import GraphGrammar.Transformation
26
27 import Assorted.Maybe
28
29 {-| A graph builder that carries extra state (s). -}
30
31 {- 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
32    m is the embadded monad and r is the return type. -}
33 type GraphBuilderT a b s m r = StateT (Digraph a b, s) m r
34
35 type GraphBuilder a b m r = GraphBuilderT a b () m r
36
37 {-| Builds from an empty graph -}
38 buildGraphT :: (Monad m) => s -> GraphBuilderT a b s m r -> m (
       Digraph a b)
39 buildGraphT = buildGraphTFrom empty
40
41 {-| Builds from an specified graph -}
42 buildGraphTFrom :: (Monad m) => Digraph a b -> s -> GraphBuilderT a
       b s m r -> m (Digraph a b)
43 buildGraphTFrom g s b =  do { (a, (g', s')) <- runStateT b (g, s);
       return g' }
44
45 buildGraphFrom :: Monad m => Digraph a b -> GraphBuilder a b m r ->
       m (Digraph a b)
46 buildGraphFrom g = buildGraphTFrom g ()
47
```

```
48  buildGraph :: Monad m => GraphBuilder a b m r -> m (Digraph a b)
49  buildGraph = buildGraphFrom empty
50
51  nodes (Digraph g _) = g
52  edges (Digraph _ e) = e
53
54  nextId m = if IM.null m then 0 else 1 + fst (findMax m)
55  nextNodeId = nextId . nodes
56  nextEdgeId = nextId . edges
57
58  getG :: Monad m => GraphBuilderT a b s m (Digraph a b)
59  getG = do { s <- get; return $ fst s }
60
61  putG :: Monad m => Digraph a b -> GraphBuilderT a b s m ()
62  putG g = do s <- get
63              put (g, snd s)
64
65  getS :: Monad m => GraphBuilderT a b s m s
66  getS = do { s <- get; return $ snd s }
67
68  putS :: Monad m => s -> GraphBuilderT a b s m ()
69  putS s = do s' <- get
70              put (fst s', s)
71
72  createNode :: Monad m => a -> Int -> GraphBuilderT a b s m Int
73  createNode p t = do g <- getG
74                      let k = nextId $ nodes g
75                      g' <- flip addNode g $ Node k t p
76                      putG g'
77                      return k
78
79  createEdge :: Monad m => b -> Int -> (Int, Int) -> GraphBuilderT a b
        s m Int
80  createEdge p t c = do g <- getG
81                        let k = nextId $ edges g
82                        g' <- flip addEdge g $ Edge k c t p
83                        putG g'
84                        return k
```

A.10 Logic.Modal

```
1  module Logic.Modal ( module Logic.Modal.Graph
2                     ) where
3
4  import Logic.Modal.Graph
```

A.11 Logic.Modal.Graph

```
1  {-# LANGUAGE TypeFamilies #-}
2  module Logic.Modal.Graph where
3
4  import Data.List
5
6  data Node a = Node Int [a] deriving (Show, Read, Eq)
7  data Edge a = Edge Int Int Int [a] deriving (Show, Read, Eq)
8  data Graph a = Graph [Node a] [Edge a] deriving (Show, Read, Eq)
9
```

```
10 class Element e where
11         type Payload e :: *
12         elementId :: e -> Int
13         values    :: e -> [Payload e]
14
15 instance Element (Node a) where
16         type Payload (Node a) = a
17         elementId (Node i _) = i
18         values    (Node _ v) = v
19
20 instance Element (Edge a) where
21         type Payload (Edge a) = a
22         elementId (Edge i _ _ _) = i
23         values    (Edge _ _ _ v) = v
24
25 src :: Edge a -> Int
26 src (Edge _ s _ _) = s
27
28 tgt :: Edge a -> Int
29 tgt (Edge _ _ t _) = t
30
31 nexts :: Graph a -> Int -> [Int]
32 nexts (Graph _ es) nId = map tgt $ filter (\x -> src x == nId) es
33
34 follows :: Graph a -> Int -> Int -> Bool
35 follows g a b = a `elem` nexts g b
36
37 prevs :: Graph a -> Int -> [Int]
38 prevs g@(Graph ns _) i = let ids = map elementId ns
39                          in filter (\i' -> i `elem` nexts g i') ids
40
41 precedes :: Graph a -> Int -> Int -> Bool
42 precedes g a b = a `elem` prevs g b
43
44 findById :: Element a => [a] -> Int -> [a]
45 findById es i = filter (\x -> elementId x == i) es
46
47 node :: Graph a -> Int -> Node a
48 node (Graph ns _) = head . findById ns
49
50 nodes (Graph ns _) = ns
51
52 edge :: Graph a -> Int -> Edge a
53 edge (Graph _ es) = head . findById es
54
55 edges (Graph _ es) = es
56
57 pathsStartingAt :: Graph a -> Int -> [[Int]]
58 pathsStartingAt g i = let
59         ns = nexts g i
60 in if null ns then
61   return [i]
62 else do
63   n <- ns
64   ns' <- pathsStartingAt g n
65   return $ if n `elem` ns' then ns' else i:ns'
66
67 successors :: Graph a -> Int -> [Int]
```

```
68 successors g s = foldl (union) [] $ pathsStartingAt g s
69
70 elementIds :: Element a => [a] -> [Int]
71 elementIds = map elementId
72
73 nodeIds = elementIds . nodes
74 edgeids = elementIds . edges
```

A.12 Logic.CTL

```
1  module Logic.CTL ( check
2                  , module Logic.CTL.Parser
3                  , module Logic.CTL.Base
4                  , module Logic.CTL.Semantics
5                  , module Logic.Modal
6                      ) where
7
8  import Logic.CTL.Parser
9  import Logic.CTL.Base
10 import Logic.CTL.Semantics
11 import Logic.Modal
12
13 check :: Graph String -> CTL -> Int -> Bool
14 check g f s0 = s0 `elem` (map elementId $ ctlSat g f)
```

A.13 Logic.CTL.Base

```
1  module Logic.CTL.Base where
2
3  data CTL = Literal Bool
4           | Atom String
5           | Not CTL
6           | And CTL CTL
7           | Or CTL CTL
8           | Implies CTL CTL
9           | AllNext CTL
10          | SomeNext CTL
11          | AllFuture CTL
12          | SomeFuture CTL
13          | AllGlobal CTL
14          | SomeGlobal CTL
15          | AllUntil CTL CTL
16          | SomeUntil CTL CTL
17          deriving (Show, Eq, Read)
```

A.14 Logic.CTL.Parser

```
1 module Logic.CTL.Parser (parseCTL) where
2
3 import Text.ParserCombinators.Parsec
4
5 import Data.List
6
7 import Logic.CTL.Base
8
9 parseCTL :: Parser CTL
```

```
10 parseCTL = do p <- pctl1
11               eof
12               return p
13
14 pctl1 = do spaces
15            try parseAU <|> try parseEU <|> try implies
16
17 implies = do p <- pctl2
18              try (impliesCont p) <|> return p
19
20 impliesCont p = do spaces
21                    string "->"
22                    spaces
23                    q <- pctl1
24                    return $ Implies p q
25
26 parseAU = do char 'A'
27              spaces
28              char '['
29              spaces
30              p <- pctl1
31              spaces
32              char 'U'
33              spaces
34              q <- pctl1
35              spaces
36              char ']'
37              return $ AllUntil p q
38
39 parseEU = do char 'E'
40              spaces
41              char '['
42              spaces
43              p <- pctl1
44              spaces
45              char 'U'
46              spaces
47              q <- pctl1
48              spaces
49              char ']'
50              return $ SomeUntil p q
51
52 pctl2 = do p <- pctl3
53            try (pAndOr p) <|> return p
54
55 pAndOr p = do spaces
56               op <- binaryOP
57               spaces
58               q <- pctl2
59               return $ op p q
60
61 binaryOP = do try (do string "&&" ; return And) <|> try (do string "
       ||" ; return Or)
62
63 pctl3 = do try pNot <|> try pTempOp <|> try parens <|> try ident
64
65 pNot = do char '~'
66           spaces
```

```
67            p <- pctl3
68            return $ Not p
69
70  pTempOp = do op <- tempOp
71               spaces
72               p <- pctl3
73               return $ op p
74
75  tempOp = try ax <|> try af <|> try ag <|> try ex <|> try ef <|> try
        eg
76
77  parens = do char '('
78              p <- pctl1
79              spaces
80              char ')'
81              return p
82
83  ax = do { string "AX"; return AllNext }
84  ag = do { string "AG"; return AllGlobal }
85  af = do { string "AF"; return AllFuture }
86  ex = do { string "EX"; return SomeNext }
87  eg = do { string "EG"; return SomeGlobal }
88  ef = do { string "EF"; return SomeFuture }
89
90  ident = do
91    c <- letter
92    cs <- many (letter <|> digit)
93    let tIdent = c:cs
94
95    if tIdent == "true" then
96      return true
97    else if tIdent == "false" then
98      return false
99    else if any (==tIdent) reservedWords then
100     unexpected (tIdent ++ " is a reserved word")
101   else
102     return $ Atom tIdent
103
104 reservedWords = [ "AX", "AF", "AG", "EX", "EF", "EG" ]
105 true  = Literal True
106 false = Literal False
```

A.15 Logic.CTL.Semantics

```
1  module Logic.CTL.Semantics where
2
3  import Data.List
4
5  import Logic.CTL.Base
6  import Logic.Modal
7
8  ctlSat :: Graph String -> CTL -> [Node String]
9  ctlSat g (Literal True)    = nodes g
10 ctlSat g (Literal False)   = []
11 ctlSat g (Atom v)          = filter (\x -> v `elem` values x) $ nodes
       g
12 ctlSat g (Not p)           = nodes g \\ ctlSat g p
13 ctlSat g (And p q)         = ctlSat g p `intersect` ctlSat g q
```

```
14 ctlSat g (Or p q)          = ctlSat g p `union` ctlSat g q
15 ctlSat g (Implies p q)     = ctlSat g $ Or q $ Not p
16 ctlSat g (AllNext p)       = ctlSat g $ Not $ SomeNext $ Not p
17 ctlSat g (AllFuture p)     = afSat  g p
18 ctlSat g (AllGlobal p)     = ctlSat g $ Not $ SomeFuture $ Not p
19 ctlSat g (AllUntil p q)    = ctlSat g $ Not (Or (SomeUntil (Not q) (
       And (Not p) (Not q))) (SomeGlobal (Not q)))
20 ctlSat g (SomeNext p)      = exSat  g p
21 ctlSat g (SomeFuture p)    = ctlSat g $ SomeUntil (Literal True) p
22 ctlSat g (SomeGlobal p)    = ctlSat g $ Not $ AllFuture $ Not p
23 ctlSat g (SomeUntil p q)   = euSat  g p q
24
25 afSat :: Graph String -> CTL -> [Node String]
26 afSat g p = let x = elementIds $ nodes g
27                 y = elementIds $ ctlSat g p
28             in nodesById g $ recurse x y
29     where recurse x y
30               | x == y    = y
31               | otherwise = let x' = y
32                                 y' = y `union` preA g y
33                             in recurse x' y'
34
35 exSat :: Graph String -> CTL -> [Node String]
36 exSat g p = let x = elementIds $ nodes g
37                 y = preE g x
38             in nodesById g y
39
40 euSat g p q = let w = elementIds $ ctlSat g p
41                   x = elementIds $ nodes g
42                   y = elementIds $ ctlSat g q
43               in nodesById g $ recurse w x y
44     where recurse w x y
45               | x == y    = y
46               | otherwise = let x' = y
47                                 y' = y `union` (w `intersect` preE g
                                       y)
48                             in recurse w x' y'
49
50 preA :: Graph a -> [Int] -> [Int]
51 preA g y = let ant = foldl union [] $ map (prevs g) y
52                condition s = nexts g s `subsetOf` y
53            in filter condition ant
54
55 preE :: Graph a -> [Int] -> [Int]
56 preE g ids = foldl union [] $ map (prevs g) ids
57
58 subsetOf :: Eq a => [a] -> [a] -> Bool
59 subsetOf as bs = all (`elem` bs) as
60
61 nodesById g = map (node g)
```

A.16 Translation

```
1 module Translation where
2
3 import Data.List
4
5 import qualified GraphGrammar as Grammar
```

```
6  import qualified Logic.Modal  as Model
7
8  translate :: Grammar.StateSpace String String -> Model.Graph String
9  translate g = let gns = Grammar.nodes g
10                   ges = Grammar.edges g
11                   edgesConnected n = filter (\x -> Grammar.elemId n
                        == Grammar.sourceID x) ges
12                   ruleName (Grammar.Edge _ _ _ (Grammar.Rule n _ _ _
                        , _)) = n
13                   atoms = nub . map ruleName . edgesConnected
14                   nodes = map (\n@(Grammar.Node i _ _) -> Model.Node
                         i $ atoms n) gns
15                   edges = map (\(Grammar.Edge i (s, t) _ _) -> Model
                        .Edge i s t []) ges
16
17               in Model.Graph nodes edges
```

A.17 Main

```
1  module Main where
2
3  import System.Console.GetOpt
4  import System.Environment
5
6  import Control.Monad
7  import Control.Monad.IO.Class
8
9  import Text.ParserCombinators.Parsec
10
11 import GraphGrammar
12 import Logic.CTL
13 import Translation
14 import GML
15
16 main = do
17   (options, strs) <- parseOptions
18   stateSpace <- runSerializedGrammar (stepsToStop options) (head
         strs)
19   let model = translate stateSpace
20
21   forM_ (tail strs) $ \f -> do
22     let f' = parseFormula f
23
24     if check model f' $ initialState options
25       then putStrLn $ concat ["Formula \"", f, "\" holds for state "
           , show $ initialState options]
26       else putStrLn $ concat ["Formula \"", f, "\" fails to hold on
           state ", show $ initialState options]
27
28     if graphicalOutput options
29       then writeStateSpace stateSpace "state_space.gml" >>
30         writeStates stateSpace >>
31         writeModel model "model.gml"
32       else return ()
33
34
35 data SystemOptions = SystemOptions { stepsToStop     :: Int
36                                    , graphicalOutput :: Bool
```

```
37                                   , initialState    :: Int
38                                   }
39
40  parseOptions :: MonadIO m => m (SystemOptions, [String])
41  parseOptions = do args <- liftIO getArgs
42                    case getOpt Permute options args of
43                            (o,(h:t),[]) -> return (foldl (flip id)
                                 defaultOptions o, (h:t))
44                            (_,_,es)     -> fail $ concat es ++
                                 usageInfo header options
45      where header = "Usage: main [args] file checks"
46
47  options = [ Option ['s'] ["stop-after"]    (ReqArg (\d o -> o {
        stepsToStop = read d }) "n")  "stops after n iterations"
48            , Option ['d'] ["draw"]          (NoArg  (\o   -> o {
                  graphicalOutput = True }))    "writes the state space
                  as a series of images"
49            , Option ['i'] ["initial-state"] (ReqArg (\d o -> o {
                  initialState = read d }) "n") "evaluates the formulas
                  for state n"
50            ]
51
52  defaultOptions = SystemOptions { stepsToStop     = 10000
53                                 , graphicalOutput = False
54                                 , initialState    = 0
55                                 }
56
57  parseFormula :: String -> CTL
58  parseFormula s = case parse parseCTL "" s of
59                           Right ctl -> ctl
60                           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