UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

SIMONE ANDRÉ DA COSTA CAVALHEIRO

Relational Approach of Graph Grammars

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Prof^a. Dr^a. Leila Ribeiro
Advisor

Prof. Dr. Antônio Carlos da Rocha Costa
Coadvisor

Porto Alegre, July 2010

CIP – CATALOGING-IN-PUBLICATION

Cavalheiro, Simone André da Costa

Relational Approach of Graph Grammars / Simone André da Costa Cavalheiro. – Porto Alegre: PPGC da UFRGS, 2010.

137 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2010. Advisor: Leila Ribeiro; Coadvisor: Antônio Carlos da Rocha Costa.

1. Graph grammar. 2. Theorem proving. 3. First-order logic. 4. Formal specification. 5. Formal verification. I. Ribeiro, Leila. II. Costa, Antônio Carlos da Rocha. III. Título.

ACKNOWLEDGEMENTS

I am very grateful to my supervisor, Leila Ribeiro, who accepted me under her supervision as a PhD student, and for her support and guidance during this research. I would like to thank her for the various revisions and suggestions throughout this work. Without her support, this work would not be the same.

I would like to express my thanks to my friend, teacher and co-advisor Antônio Carlos da Rocha Costa. He has given me great incentive since my master's course. I would also like to show my gratitude to my colleagues from the Department of Informatics at UFPel. Their support makes possible the conclusion of this work.

Special thanks to my husband and son, for their love and care, and support in difficult moments. Thanks to all my family that was always looking forward to seeing the success of this work. Thanks also to my friends for the emotional support. Thanks to God, for making me able to carry out this work.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Graph grammars are a formal language well-suited to applications in which states have a complex topology (involving not only many types of elements, but also different types of relations between them) and in which behaviour is essentially data-driven, that is, events are triggered basically by particular configurations of the state. Many reactive systems are examples of this class of applications, such as protocols for distributed and mobile systems, simulation of biological systems, and many others. The verification of graph grammar models through model-checking is currently supported by various approaches. Although model-checking is an important analysis method, it has as disadvantage the need to build the complete state space, which can lead to the state explosion problem. Much progress has been made to deal with this difficulty, and many techniques have increased the size of the systems that may be verified. Other approaches propose to over- and/or under-approximate the state-space, but in this case it is not possible to check arbitrary properties. Besides model checking, theorem proving is another well-established approach for verification. Theorem proving is a technique where both the system and its desired properties are expressed as formulas in some mathematical logic. A logical description defines the system, establishing a set of axioms and inference rules. The process of verification consists of finding a proof of the required property from the axioms or intermediary lemmas of the system. Each verification technique has arguments for and against its use, but we can say that model-checking and theorem proving are complementary. Most of the existing approaches use model checkers to analyse properties of computations, that is, properties over the sequences of steps a system may engage in. Properties about reachable states are handled, if at all possible, only in very restricted ways. In this work, our main aim is to provide a means to prove properties of reachable graphs of graph grammar models using the theorem proving technique. We propose an encoding of (the Single-Pushout approach of) graph grammar specifications into a relational and logical approach which allows the application of the mathematical induction technique to analyse systems with infinite state-spaces. We have defined graph grammars using relational structures and used logical languages to model rule applications. We first consider the case of simple (typed) graphs, and then we extend the approach to the non-trivial case of attributed-graphs and grammars with negative application conditions. Besides that, based on this relational encoding, we establish patterns for the presentation, codification and reuse of property specifications. The pattern has the goal of helping and simplifying the task of stating precise requirements to be verified. Finally, we propose to implement relational definitions of graph grammars in event-B structures, such that it is possible to use the event-B provers to demonstrate properties of a graph grammar.

Keywords: Graph grammar, theorem proving, first-order logic, formal specification, formal verification.

Abordagem Relacional de Gramática de Grafos

RESUMO

Gramática de grafos é uma linguagem formal bastante adequada para sistemas cujos estados possuem uma topologia complexa (que envolvem vários tipos de elementos e diferentes tipos de relações entre eles) e cujo comportamento é essencialmente orientado pelos dados, isto é, eventos são disparados por configurações particulares do estado. Vários sistemas reativos são exemplos desta classe de aplicações, como protocolos para sistemas distribuídos e móveis, simulação de sistemas biológicos, entre outros. A verificação de gramática de grafos através da técnica de verificação de modelos já é utilizada por diversas abordagens. Embora esta técnica constitua um método de análise bastante importante, ela tem como desvantagem a necessidade de construir o espaço de estados completo do sistema, o que pode levar ao problema da explosão de estados. Bastante progresso tem sido feito para lidar com esta dificuldade, e diversas técnicas têm aumentado o tamanho dos sistemas que podem ser verificados. Outras abordagens propõem aproximar o espaço de estados, mas neste caso não é possível a verificação de propriedades arbitrárias. Além da verificação de modelos, a prova de teoremas constitui outra técnica consolidada para verificação formal. Nesta técnica tanto o sistema quanto suas propriedades são expressas em alguma lógica matemática. O processo de prova consiste em encontrar uma prova a partir dos axiomas e lemas intermediários do sistema. Cada técnica tem argumentos pró e contra o seu uso, mas é possível dizer que a verificação de modelos e a prova de teoremas são complementares. A maioria das abordagens utilizam verificadores de modelos para analisar propriedades de computações, isto é, sobre a seqüência de passos de um sistema. Propriedades sobre estados alcançáveis só são verificadas de forma restrita. O objetivo deste trabalho é prover uma abordagem para a prova de propriedades de grafos alcançáveis de uma gramática de grafos através da técnica de prova de teoremas. Propõe-se uma tradução (da abordagem *Single-Pushout*) de gramática de grafos para uma abordagem lógica e relacional, a qual permite a aplicação de indução matemática para análise de sistemas com espaço de estados infinito. Definiu-se gramática de grafos utilizando estruturas relacionais e aplicações de regras com linguagens lógicas. Inicialmente considerou-se o caso de grafos (tipados) simples, e então se estendeu a abordagem para grafos com atributos e gramáticas com condições negativas de aplicação. Além disso, baseado nesta abordagem, foram estabelecidos padrões para a definição, codificação e reuso de especificações de propriedades. O sistema de padrões tem o objetivo de auxiliar e simplificar a tarefa de especificar requisitos de forma precisa. Finalmente, propõe-se implementar definições relacionais de gramática de grafos em estruturas de *event-B*, de forma que seja possível utilizar os provadores disponíveis para *event-B* para demonstrar propriedades de gramática de grafos.

Palavras-chave: Gramática de Grafos, prova de teoremas, lógica de primeira ordem, especificação formal, verificação formal.

1 INTRODUCTION

1.1 Motivation

Hardware and software systems are everywhere: in communication, transportation, financial, administration and in our homes. Everyday they grow in scale and scope, many times having to interact with another complex and independent environments. Together with this increase in complexity, the possibility of subtle errors is intensified and can lead to catastrophic losses (examples are found in (DAVIS, 2005) and (HUTH; RYAN, 2000)). Facing this, techniques to aid the development of reliable and correct systems are becoming more and more needed (DWYER et al., 2007). One way of achieving this goal is through the use of formal methods, which are mathematically-based techniques that can offer rigorous and effective ways to model, design and analyse computer systems (CRAIGEN; GERHART; RALSTON, 1993).

During the past two decades, various case studies and industrial applications (WOODCOCK et al., 2009; ALPUENTE; COOK; JOUBERT, 2009; CLARKE; WING, 1996; HINCHEY; BOWEN, 1995; CRAIGEN; GERHART; RALSTON, 1993) have confirmed the significant importance of the use of formal methods to improve the quality of both hardware and software designs. The description of a system by a formal specification language has shown to provide a solid foundation to guide later development activities and obtain through verification a high confidence that the system satisfies its requirements. Well-formed specifications, validated with respect to critical properties, have supplied a basis for generating correct and efficient source code. Notable examples can be found in the most diverse domains (WOODCOCK et al., 2009; HANEBERG et al., 2007; HOMMERSOM et al., 2007; HINCHEY; BOWEN, 1999; BOWEN; HINCHEY, 1997; HINCHEY; BOWEN, 1995): transportation, telecommunication and information systems, security, protocols and hardware. An impressive and relatively recent example is the traffic management system for line 14 (Tolbiac-Madeleine) of the Paris metro system (BEHM et al., 1999). The system is completely automatic (supporting driverless trains) and had the safety-critical parts formally developed by Matra Transport International using B (ABRIAL, 1996). According to (BEHM et al., 1999), the abstract and concrete model was specified with approximately 100,000 lines of B code and 87,000 lines of ADA code. About 28,000 lemmas were automatically proved by B tool. Errors were found and corrected during the development. After applying a conventional testing process, not a single error was found.

Nevertheless, the employment of such methods is far from trivial: it requires some mathematical expertise, demands quality documentation and increases the time spent in the first stages of the development. Despite a significant number of successful stories, the software engineering community has not been convinced on a large scale of the validity of formal approaches (BOWEN; HINCHEY, 2005, 2006). The most common reasons against the use of formal methods in practice are the extension of development cycle, the need of extensive personnel training, the difficulties in finding suitable abstractions and the mathematical knowledge required. Therefore, though promising, several improvements are needed to turn the use of these methods and their support tools a common practice in software development process.

Experts in formal methods have analysed the situation and examined the issues concerning the use of formal methods in industrial software development (ABRIAL, 2006; ROSSI, 2005; KNIGHT, 1998; MICHAEL; W., 1996; ROSENBLUM, 1996). One pondered possibility is that present formal methods might be incomplete or inadequate for some applications. Heitmeyer (HEITMEYER, 2006) argues that the most popular modeling languages used in industry, like UML (JACOBSON; BOOCH; RUMBAUGH, 1999) and Stateflow (The MathWorks, 2007), lack explicit formal semantics and produce large specifications (including a lot of implementing details). To tackle this situation he proposes to enhance existing formal languages with features such as suitable graphical interfaces, encouraging thus their use by practitioners.

This scenario claims further research to provide suitable specification and verification techniques for the software development community. Our dependence on software systems grows everyday. Clients and users demand that their systems are delivered with a high level of accuracy and trust. Software engineers should have the tools upon which this trust can be built. The present thesis contributes to the development of such tools, furnishing theoretical foundations for the analysis of a range of systems.

1.2 Graph Grammar

Graph grammars (short GG) are a formal language suitable for the specification of a wide range of computational systems (EHRIG et al., 1999). This formalism is specially well-suited to applications in which states have a complex topology (involving not only many types of elements, but also different types of relations between them) and in which behaviour is essentially data-driven, that is, events are triggered basically by particular configurations of the state. Many reactive systems are examples of this class of applications, such as protocols for distributed and mobile systems, simulation of biological systems, etc. Additionally to the complex states and reactive behaviour, concurrency and non-determinism play an essential role in this area of applications: many events may happen concurrently, if they all are enabled, and the choice of occurrence between conflicting events is non-deterministic.

The basic idea of graph grammars is to model the states of a system as graphs and describe the possible state changes as rules (where the left- and right-hand sides are graphs). The operational behaviour of the system is expressed via applications of these rules to graphs depicting the actual states of the system. Rules operate locally on the state-graph, and therefore it is possible that many rules are applied at the same time.

In general, a graph grammar system is composed by a *type graph*, characterizing the types of vertices and edges allowed in the system, an *initial graph*, representing the initial state of the system and a *set of rules*, describing the possible state changes that can occur. A rule has a left-hand side and a right-hand side, which are both graphs, and a partial graph morphism that connect the graphs in some compatible way and determine what should be modified by the rule application. Depending on the conditions imposed by these rules, they may be mutually exclusive or not. In the latter case, one of them will be chosen non-deterministically to be executed. The initial state has the function of restricting the computation and the reachable states allowed in the system. All state graphs are labeled by the type graph via graph morphisms. This allows that some inconsistent states of the system be ruled out by the typing compatibility.

Typically, the semantics of a system described using a graph grammar is a transition system where the states are graphs and the transitions describe the possible rule applications. A rule is applicable in a state if there is a match, that is, an occurrence of the left-hand side of the rule in the state. This formalism has been used in very distinct applications such as image recognition and generation (LLADÓS; SÁNCHEZ, 2003; HUSSEIN; HASSANIEN, 1999; BUNKE, 1991), analysis of fault behaviours (DOTTI; RIBEIRO; SANTOS, 2003), database models (SONG et al., 2004), music composition (WANKMÜLLER, 1986), DNA computing (CERVO; RIBEIRO, 2002) and visual programming languages (ZHANG; ZHANG; ORGUN, 2001), among many others (SAKSENA; WIBLING; JONSSON, 2008; CORRADINI et al., 2006; EHRIG et al., 1999, 1987).

Graph grammars are appealing as specification formalism because they are formal and based on simple, but powerful concepts to describe behaviour. At the same time they have a nice graphical layout that helps even non-theoreticians understand a specification. Due to the declarative style (using rules), concurrency arises naturally in a specification: if rules do not conflict (do not try to update the same portion of the state), they may be applied in parallel (it is not necessary to say explicitly which rules shall occur concurrently). Consequently, graph grammars can be seen as a very suitable formalism to achieve a good and understandable description of concurrent systems.

The verification of concurrent systems is much more complex than sequential ones. Concurrent systems usually consist of several autonomous components that run in parallel and interact with each other (for example, via messages). The interaction between these components affects the behaviour of the whole system, such that is not enough to know that each component works as expected to know that the whole system will present the expected behaviour. For that reason, the analysis of this kind of systems demands the verification of the system as a whole, and this is a difficult task: the high level of parallelism generates a number of possible computations. In such situations, the reasoning is almost unfeasible without adopting formal techniques. Therefore, we can say that the use of formal methods for verification purposes is mandatory for ensuring correctness of concurrent systems.

1.3 Model Checking and Theorem Proving

Model Checking (EDMUND M. CLARKE; GRUMBERG; PELED, 1999) and Theorem Proving (ROBINSON; VORONKOV, 2001) are two well-established approaches used to analyse systems for critical and desired properties. Model checking takes as input a finite model representing a concurrent system and a property to be checked against the system, and then exhaustively performs a state space search deciding if the property holds in that model. The process is automatic, in many cases efficient and can also be used to check partial specifications. In some cases, when properties are violated counterexamples can be produced, providing important debugging information.

Since the number of states of a system is typically exponential in the size of its description, the main disadvantage of model checking is the state explosion problem. Much progress has been made to deal with this difficulty, and many techniques have increased the size of the systems that may be verified: partial ordered reduction (LLUCH-LAFUENTE; EDELKAMP; LEUE, 2002), abstraction (CLARKE et al., 2001), symbolic representation (BIERE et al., 1999; BURCH et al., 1992), among others. However, these approaches generally derive the model as an under- or over-approximation of system's behaviour, which can result in inconclusive error reports or inconclusive verification reports (DWYER et al., 2007).

Theorem proving (CLARKE; WING, 1996) is a technique where both the system and its desired properties are expressed as formulas in some mathematical logic. A formal system defines the logic, establishing a set of axioms and inference rules. The verification process consists in finding a proof of the required property from the axioms or intermediary lemmas of the system. In contrast to model checking, theorem proving can deal directly with infinite state spaces and it relies on techniques such as structural induction to prove over infinite domains. The use of this technique may require interaction with a human; however, by constructing the proof the user often gains very useful perceptions into the system and/or the property being proved.

1.4 Graph Grammar Analysis

In this section we review methods and tools available for the formalism of graph grammars. First we present environments based on graph transformation proposed for the development of software systems, considering in particular the available analysis techniques in each tool. Finally we discuss approaches designed to model check graph grammars.

There are at least two widespread graph transformation languages, AGG (ERMEL; RUDOLF; TAENTZER, 1999) and PROGRES (SCHüRR; WINTER; ZüNDORF, 1999), which offer a declarative and visual programming method for the development of software systems. PROGRES (PROgrammed Graph REwriting Systems) (RANGER; WEINELL, 2008) is an environment for creating, analysing (type checking), compiling and debugging graph transformation specifications.

The AGG (Attributed Graph Grammar) system (THE AGG SYSTEM, 2010; ERMEL; RUDOLF; TAENTZER, 1999), besides simulation, supports validation of attributed graph grammars. Attributed graph grammars extend the basic graph grammar formalism with attributed graphs, giving raise to a language to reason about attributes (data values). Compared with AGG (FUSS et al., 2007), PROGRES provides the highest level of maturity, including a syntax-directed editor, an interpreter, and a code generation mechanism. On the other hand, AGG provides more analysis methods than PROGRES.

An attributed graph grammar can be validated in AGG system through two analysis techniques, namely critical pair analysis (HECKEL; KüSTER; TAENTZER, 2002) and consistency checking (HECKEL; WAGNER, 1995). Critical pair analysis is used to check if a system has a functional behaviour. Functional behaviour is required when the specification has to be functional (i.e., terminating and confluent (HECKEL; KüSTER; TAENTZER, 2002)) in order to ensure the existence of a unique result. For instance, functional behaviour is specially important when we use graph grammars for automated translation of visual models into code or semantic domains (HAUSMANN; HECKEL; TAENTZER, 2002), since the result of the translation must be unique. The functional behaviour is also expected when using graph grammars for parsing visual languages (BOTTONI; TAENTZER; SCHüRR, 2000), since it avoids the overhead of backtracking necessary for the parsing.

The AGG consistency control mechanism is able to check if a given graph satisfies certain consistency conditions specified for a graph grammar. Consistency conditions (HECKEL; WAGNER, 1995) describe basic properties of graphs that have to be preserved by the application of rules. AGG transforms global consistency conditions into post application conditions for individual rules. A so-constructed rule is applicable to a consistent graph if and only if the derived graph is consistent, too. The post application conditions generated from graphical consistency constraints ensure consistency of a graph grammar during rule application. However, graphical consistency constraints just express very basic graph conditions such as the existence or uniqueness of certain nodes or edges. They can not express structural conditions like the existence of paths or circles of arbitrary length or global properties as e.g. connectivity. Also, the translation of consistency constraints into post conditions might cause problems for rules with attribute conditions.

The Tiger project (Transformation-based Generation of modeling Environments) extends (TIGER PROJECT, 2010) the AGG engine by a concrete visual syntax definition for visual model representation. From the definition of the visual language, the Tiger generator generates Java source code. The generated Java code implements an Eclipse visual editor plug-in based on Graphical Editing Framework (CONSORTIUM, 2010; EHRIG et al., 2005). The result is a generated environment for the visual language simulation (RENSINK et al., 2008; TAMáS MéSZáROS; MEZEI, 2008). The focus of Tiger is the visualization power and not the graph transformation speed. Also, since it is an editor generator, where graph rules are translated to palette or context menu entries, no means to control rule application, and to apply more than one rule without user interaction are supported.

Besides that, a simulation environment for a specific class of graph grammars, Object Based Graph Grammars (OBGG), was proposed in (DUARTE et al., 2002). OBGG (DOTTI; RIBEIRO, 2000) incorporate object-based concepts, such as communication through message passing and encapsulation, to describe object-based systems. The proposed framework was used to simulate mobile applications for open environments (RÖDEL et al., 2002), control systems (COPSTEIN; COSTA MÓRA; RIBEIRO, 2000) and others (RIBEIRO; COPSTEIN, 1998). The first step to simulate an OBGG system consists in translating the specification into a simulation model. The environment uses a defined algorithm to translate the specification to a Java program, that actually simulates the behaviour of the specification. This process was shown to be very useful in finding specification errors (for example, missing rules or wrong behaviours) and estimating the communication behaviour (for example, the number of exchanged messages to complete a service) (RIBEIRO; DOTTI; BARDOHL, 2005; KREOWSKI et al., 2005).

However, through simulation, it is not possible to make conclusive assertions about the behaviour of a system. Thus, many methods and tools were proposed to allow the model checking of graph grammars. GROOVE (GRaphs for Object-Oriented Verification) (RENSINK, 2004a) is a tool that generates the space-state of a graph grammar, in the attempt that the resulting transition system can be model checked. But, as emphasized in (KASTENBERG; RENSINK, 2006a), if we consider time-performance, GROOVE can not compete with SPIN (HOLZMANN, 1997a). For that reason, a common strategy that has been applied is the translation of graph grammar models into formal languages that are input languages of established model checkers. In such case, the main steps for model checking of graph grammars specifications are (RIBEIRO; DOTTI; BARDOHL, 2005): i) translate the specification to a verification model that serves as input to a model checker; ii) define properties in some temporal logic; iii) check the properties against the model (model checking); iv) analyse results.

(FOSS; RIBEIRO, 2004) presents a translation of OBGG specifications to π-calculus (MILNER, 1999). Following this method, automatic checkers (for example, HAL (FERRARI et al., 1998) and Mobility Workbench (VICTOR; MOLLER, 1994)) can be used to verify a system. The semantic compatibility of the translation is depicted in (FOSS, 2003). Nevertheless, as described in (RIBEIRO; DOTTI; BARDOHL, 2005), some problems were encountered using this approach: models had to be considerably restricted such that OBGG objects had no internal state and limitations were brought by the use of existing model checkers (specially while supporting the replication operator of π-calculus). As a result, only small examples could be translated.

Another proposal (DOTTI et al., 2003) translated OBGG specifications to PROMELA (PROcess/PROtocol MEta LAnguage), allowing the verification of OBGG models using the SPIN (Simple Promela INterpreter) model checker (HOLZMANN, 1997a). This approach provides a means to verify properties based on events. Verification of properties based on states only works for specifications with a static number of objects. For specifications with dynamic creation of objects, it would be necessary to create dynamically new global variables - feature not supported by the tool. Then, the focus given in this proposal was to prove properties about possible OBGG derivations.

Compositional verification, using an assume-guarantee approach, is also provided (DOTTI et al., 2006). This work improved the approach for property specification, enabling the proof of properties about the internal state of involved objects. Moreover, there is an extension of graph grammars (MICHELON; COSTA; RIBEIRO, 2006) that explicitly models time restrictions and allows the automatic verification of properties with the UPPAAL model checker (BEHRMANN; DAVID; LARSEN, 2004). In this case, semantics of real-time systems is defined in terms of Timed Automata (ALUR; DILL, 1994), the input language of UPPAAL. Besides, verification techniques for another kind of graph transformation systems can be found in (RENSINK; SCHMIDT; VARRÓ, 2004).

Although model checking is an important analysis method, it has as disadvantage the need to build the complete state space, which can lead to the state explosion problem. In many cases, verification terminates because of insufficient resources, such as memory. Consequently, the use of this approach can be very time and space consuming, not allowing the verification of properties of many systems.

Several works (MCNEW; KLAVINS, 2006; KORFF, 1991; DIXIT; MOLDOVAN, 1991) have been concerned on reducing the usually enormous number of states and transitions produced by a graph grammar system. Paolo Baldan, Andrea Corradini and Barbara König propose a framework (BALDAN; CORRADINI; KöNIG, 2008; BALDAN; KÖNIG, 2002) for the verification of infinite-state graph transformation systems based on the construction of finite structures approximating their behaviour. Details about such approach are described in Section 1.6.

Each verification technique has arguments for and against its use, but we can say that model-checking and theorem proving are complementary. Most of the existing approaches use model checkers to analyse properties of computations. Properties about reachable states are handled, if at all possible, only in very restricted ways. Currently there are no approaches that allow the use of theorem provers to prove properties that involve infinite states in the context of graph grammars. Our work was developed to provide a means to prove properties about reachable graphs of a (infinite-state graph) grammar using the theorem proving technique.

1.5 Goals and Structure of this Thesis

The main aim of this thesis is to provide a relational approach for graph grammars that allows the application of theorem proving technique to analyse concurrent and reactive systems that involve an infinite number of states.

More specifically, the main contributions of this work are listed below:

1. the description of a relational definition for graph grammars;
2. the extension of the approach to particular classes of graph grammars, named attributed graph grammars and graph grammars with negative application conditions;
3. the establishment of an strategy that can be applied to analyse infinite-state systems specified as graph grammars;
4. the definition of patterns for the presentation, codification and reuse of property specifications.

Besides that, we propose a translation of graph grammar specifications in Event-B structures, such that it is possible to use the theorem provers available for Event-B (for instance, through the Rodin platform) to demonstrate properties of a graph grammar. This translation is based on the relational approach of graph grammars.

The *structure of the thesis* can be divided in three main parts, Foundations, Techniques and Applications, described as follows.

Part I: Foundations. The main aim of this part is to define a representation for graph grammars that allows the use of theorem proving technique to prove properties of systems specified in this formalism. We propose the definition of graph grammars using relational structures, where rule applications are modeled as graph grammar transformations using logical formulas. We have also shown that this approach is equivalent to the Single-Pushout approach or simply SPO-approach (ROZENBERG, 1997) to graph grammars. At last, we have extended the approach to other classes of graph grammars, namely, attributed graph grammars and graph grammars with negative application conditions.

Part II: Techniques. In this part, we define a library of recursive functions and the specification of patterns that can be used to specify properties over reachable states for systems specified in graph grammars. The pattern has the goal of helping and simplifying the task of stating precise requirements to be verified.

Part III: Applications. In the last part, we use Event-B to analyse properties of graph grammars. We translate graph grammar specifications in Event-B structures, such that it is possible to use the Event-B provers to demonstrate properties of a graph grammar. This translation is based on the relational definition of graph grammars.

1.6 Related Works

1.6.1 Other Approaches for Analysing Infinite-State Systems

Nowadays, several software systems involve a range of aspects like dynamic creation of objects and threads, data manipulation and others, which require the reasoning about infinite-state specifications. Many approaches have been focused on this issue. One of them is regular model checking (ABDULLA et al., 2004; BOUAJJANI; HABERMEHL; VOJNAR, 2004; KESTEN et al., 2001), an automata-based approach that encodes sets of states (or configurations) as regular sets of words and transitions as finite state transducers (automata). A crucial problem to be faced in the use of such technique to verify graph grammar models is the lack of expressivity of finite automata to represent arbitrary graphs. Another problem is the state space explosion in automata representations of the sets of configurations (or reachability relations) being examined that could just be minimized with some kind of abstraction or approximation.

Many other approaches deal with infinite-state verification. For instance, Delzanno (DELZANNO, 2000) shows that symbolic model-checking can be used to verify a large class of cache coherence protocols, while Fisher and his colleagues (FISHER; KONEV; LISITSA, 2005) apply temporal reasoning to analyse similar kind of systems. It is important to notice that in general systems analysed using such techniques must be described by simple action-reaction models, in which states must have a non-complex representation.

Paolo Baldan and Barbara König proposed (BALDAN; CORRADINI; KÖNIG, 2008; BALDAN; KÖNIG, 2002) to approximate the behaviour of (infinite-state) graph transformation systems (GTSs) by a chain of finite under-approximations or by a chain of finite over-approximations, at a level k of accuracy of the full unfolding of the system. A GTS is a finite set of graph rules. Then, a graph grammar can be seen as a GTS with an initial state. The unfolding semantics of a graph grammar (RIBEIRO, 1996; BALDAN et al., 2007) defines an operational model of computation that represents all its possible sequential and concurrent derivations (i.e., all its computations). It is generally infinite for non trivial systems. The under-approximations of the behaviour of a graph grammar are obtained by truncating the construction of the unfolding at a finite depth k (the k-truncation). The over-approximations of the behaviour of a graph grammar are achieved by constructing a Petri graph (that is a Petri net with a (hyper)graph structure over places) up to a certain depth k (the k-covering). A covering represents all computations of the original system (but possibly more).

The (under- and over-) approximations converge, in a categorical sense, to the full unfolding (BALDAN; KÖNIG, 2002): "the unfolding of a graph grammar can be expressed as the colimit of the chain of k-truncations or as the limit of the chain of k-coverings". These approximations are used to verify liveness and safety properties of a GTS. Under-approximations for infinite-state systems don't allow performing any computation of the original system in the truncations. Therefore, they are used to verify some liveness properties like "eventually P" for a predicate P. Coverings permit verifying deadlock-freedom and safety properties like "always P" for a predicate P. However, due to the presence of spurious runs, introduced by the abstraction, it is usually not possible to verify properties of the kind "there exists a run" with particular properties (BALDAN; KÖNIG; RENSINK, 2005). These approaches are restricted to GTSs with simple rules: rules can not preserve edges (but can produce and delete edges), delete nodes and consume edges with the same label. Besides, the left-hand side of a rule must be connected.

In (BALDAN; KÖNIG; RENSINK, 2005) the unfolding approach (by over-approximations) is compared with another proposal, the partitioning approach. This last one approximates graphs according to their local structure. For example, the local structure of a node can be defined by the number of incident edges; and the local structure of an edge can be defined as the tuple of the local structure of its extreme nodes. A similarity relation over the elements of a graph is used to partition the graph. This relation is originated from any function that associates to each graph element its local structure. A notion of local structure proposed in (RENSINK, 2004b) gives rise to an abstract function that preserves a fragment of two-variable first-order logic with counting quantifiers. However, rule applications or rule effects are generally not preserved. I.e., if a rule is applicable to a graph, the same rule or its abstraction must not be applicable to the abstraction of the graph. Actually, the partitioning approach is at a stage where there are only preliminary ideas on how to transform the graph abstractions (BALDAN; KÖNIG; RENSINK, 2005). There isn't a theory to perform actual verifications.

Conversely, the unfolding approach (by over-approximations) can be used to execute concrete analysis. In (BALDAN; KÖNIG; KÖNIG, 2003) a monadic second-order logic over graphs to characterize typical graph properties is proposed. It shows an encoding of such graph formulas into quantifier-free formulas over Petri net markings. The work identifies a subclass of formulas F such that the validity of F over a GTS G is implied by the validity of the encoding of F over the Petri net approximation of G. This result allows the use of verification techniques for Petri nets to analyse a given GTS. I.e., the Petri net produced by the approximated unfolding algorithm and the formula itself can be analysed by a model checker or a similar tool. Also, a tool for the analysis of GTSs using this approximation is under development (BALDAN; CORRADINI; KÖNIG, 2008; KöNIG; KOZIOURA, 2008; KÖNIG; KOZIOURA, 2005).

1.6.2 Other Approaches that Adopt a Relational, Logical or Set Theoretical Representation for Graphs and Graph Grammars

The representation of graph grammar that we have proposed was inspired by Bruno Courcelle's research about logic and graphs (COURCELLE, 2000, 1997). Courcelle investigates in various papers (BLUMENSATH; COURCELLE, 2006; COURCELLE, 2004, 1994a) the representation of graphs and hypergraphs by relational structures as well as the expressiveness of its properties by logical languages. In (COURCELLE, 1991) he presents a comparison among various descriptions of graph sets (by characteristic properties expressed in monadic second-order logic, by context-free graph grammars and by forbidden minors) and in (COURCELLE; ENGELFRIET; ROZENBERG, 1993; COURCELLE, 1990) he shows that every set of graphs defined by a single HR (Hyperedge Replacement) or VR (Vertex Replacement) graph grammar has a decidable monadic theory. The description of graph properties and transformation of graphs in monadic second-order logic is proposed at (COURCELLE, 1994b). However, these works are not particularly interested in effectively verifying properties of graph transformation systems.

Other authors have investigated the analysis of GTSs based on relational logic or set theory. Baresi and Spoletini (BARESI; SPOLETINI, 2006) explore the formal language Alloy to find instances and counterexamples for models and GTSs. In fact, with Alloy, they only analyse the system for a finite scope, whose size is user-defined. Strecker (STRECKER, 2008), aiming to verify structural properties of GTSs, proposes a formalization of graph transformations in a set-theoretic model. The approach replaces the match occurrence (i.e., the applicability condition of a rule) by a formula over graph structure, constructed over a fragment of first-order logic. Graphs and graphs transformations are formalized with datatypes, predicates, functions, definitions and transformations in a set-theoretic model. The proposal has been carried out in Isabelle (NIPKOW; PAULSON; WENZEL, 2002) and the focus is given to prove structural properties. His goal is to obtain a language for writing graph transformation programs and reasoning about them. Nevertheless, the language has only two statements, one to apply a rule repeatedly to a graph, and another to apply several rules in a specific order to a graph. Until now, the work just presents a glimpse of how to reason about graph transformations.

1.6.3 Other Approaches for Theorem Proving Concurrent Systems

CSP Prover (ISOBE; ROGGENBACH, 2008a) is an interactive theorem prover for the process algebra CSP (HOARE, 1978) based on the theorem prover Isabelle (PAULSON, 1994). CSP Prover allows the analysis of typical properties of scalable concurrent systems, such as scalability, parametrization, local activity, global result and others. Examples of concurrent systems analysed in CSP Prover are the Uniform Candy Distribution Puzzle (ISOBE; ROGGENBACH, 2008a), a systolic array (ISOBE; ROGGENBACH; GRUNER, 2005) and part of a standard of electronic payment system (ISOBE; ROGGENBACH, 2008b). Other tools for theorem proving CSP have been presented: Tej and Wolff propose another encoding of CSP in Isabelle/HOL, HOL-CSP (TEJ; WOLFF, 1997); Schneider and Dutertre encode CSP traces in PVS (DUTERTRE; SCHNEIDER, 1997).

Based on general purpose theorem provers like Isabelle (PAULSON, 1994), HOL (GORDON; MELHAM, 1993) or PVS (OWRE; RUSHBY; SHANKAR, 1992), many other tools for theorem proving process algebras (BASTEN; HOOMAN, 1999; GROENBOOM et al., 1995; CAMILLERI; INVERARDI; NESI, 1991; ARCHER et al., 1992; GERBER; GUNTER; LEE, 1991) have been presented. In addition, a range of other formal languages designed for concurrent systems have been encoded in proof assistants. The formalization of Petri Nets was specified in HOL (BARROS LUCENA, 1991), Coq (CHOPPY; MAYERO; PETRUCCI, 2008; HAMID, 2008), Isabelle (LEHMANN; LEUSCHEL, 2003), among others.

Circus (WOODCOCK; CAVALCANTI, 2001), another alternative language for the development of reactive systems, is being mechanized in the ProofPower-Z theorem prover (LEMMA1-LTD., 2010). Circus (WOODCOCK; CAVALCANTI, 2002) can be seen as a combination of Z (WOODCOCK; DAVIES, 1996) and CSP with a refinement calculus. A branch of the Circus project (PROJECT, 2010) is devoted to the mechanization of the Circus semantics and on the proof of its refinement laws (ZEYDA; CAVALCANTI, 2009). The basis of this work is the mechanization of the UTP theories (Unifying Theories of Programming) of relations, designs, reactive processes, and CSP (ZEYDA; CAVALCANTI, 2008; OLIVEIRA; CAVALCANTI; WOODCOCK, 2006).

In this thesis, we propose the use of Event-B (ABRIAL, 2007) for the analysis of graph grammar systems. Event-B has been used in the specification and analysis of many systems: interaction protocols of multi-agent systems (JEMNI BEN AYED; SIALA, 2008), bus protocols (FRANCA et al., 2009; CANSELL et al., 2002), file systems (DAMCHOOM; BUTLER; ABRIAL, 2008; DAMCHOOM; BUTLER, 2009), air traffic information system (REZAZADEH; EVANS; BUTLER, 2007), among others.

A graphical front end based on UML for Event-B, UML-B (SNOOK; BUTLER, 2008; SAID; BUTLER; SNOOK, 2009), provides support for object-oriented modelling concepts. The tools available for UML-B include drawing tools and a translator that automatically generates Event-B models. Also, an encoding of a process algebra into the Event-B method can be found in (AIT-AMEUR et al., 2009). In fact, what the authors propose is a (informal) translation of a BNF grammar to a set of Event B models. The translation is illustrated with a specific language describing a classical process algebra.

The main reason for theorem proving graph grammars is that, besides being formal, its visual style, its powerful and expressive way of describing complex states (via graphs) and its rule-based behaviour modelling provide a natural and intuitive means of describing concurrent and reactive systems. These are advantages of graph grammars comparing to other specification methods such as process algebras and Petri Nets. Although Petri Nets is also a visual language, its representation of states by sets of tokens is not well suited for the specification of systems with complex topologies on states.

Besides that, the feature of providing asynchronous communication allows a natural description of reactive systems, but has the drawback that when synchronous communication is needed, the specifier has to explicitly introduce state variables and messages with corresponding rules to simulate a synchronous message passing scheme. The choice of specification method shall always take into account the main characteristics of the application being modelled, and also the features offered by the specification formalism. For inherently synchronous systems, formalisms based on process algebras may be more adequate. For asynchronous systems, graph grammars offer a more natural specification means.

1.7 Thesis Outline

The rest of this text is organized as follows:

- Chapter 2: This chapter introduces the graph grammar specification language according to the SPO-approach (ROZENBERG, 1997). First, we present the main definitions, which are considered to underlie the following work. Next, we illustrate the use of graph grammars specifying the token-ring protocol. This working example is retaken in subsequent chapters to elucidate new definitions.

- Chapter 3: In this chapter we propose a relational approach to graph grammars that allows the application of the mathematical induction technique to analyse systems with infinite state-spaces. We have defined graph grammars using relational structures and used first-order logic to model rule applications. We also check the well-definedness of such definitions. At last, we use our approach to verify properties of the token-ring protocol.

- Chapter 4: This chapter extends the approach to attributed graph grammars. Attributed graph grammars enrich the graph grammar formalism integrating data types into graphs, by allowing assignment of values to vertices and/or edges. We first establish the relational representation of attributed graph grammars and then we modify and extend the token-ring protocol.

- Chapter 5: In this chapter we consider the case of graph grammars with negative application conditions. Negative application conditions restrict the application of a rule by asserting that a specific structure must not be present in a state-graph, before applying the rule. We also show the use of graph grammars with negative application conditions for the specification and verification of the token-ring protocol.

- Chapter 6: This chapter presents specification patterns for properties over reachable states in the approach of graph grammars. The patterns are based on functions that describe typical characteristics or elements of graphs (like the type of a vertex, the set of all edges of some type, the cardinality of vertices, etc.). We show how these functions can be defined in the framework of relational graph grammars.

- Chapter 7: In this chapter we use Event-B to analyse properties of graph grammars. Due to the similarity between Event-B models and graph grammar specifications, specially concerning the rule-based behaviour, we propose to translate graph grammar specifications in Event-B structures, such that it is possible to use the Event-B provers to demonstrate properties of a graph grammar. This translation is based on the relational definition of graph grammars.
- Chapter 8: Finally, we summarise the contributions of this thesis and list possible developments for the work presented here.

2 GRAPH GRAMMARS

In this chapter, we review the basic definitions of graph grammars used in this thesis. It can be seen as a set-theoretical presentation of the algebraic single-pushout approach (see, e.g., (ROZENBERG, 1997; BALDAN; KÖNIG; KÖNIG, 2003)).

Graph grammars generalize Chomsky grammars from strings to graphs: it specifies a system in terms of states, described by graphs, and state changes, described by rules having graphs at the left- and right-hand sides. Graph rules are used to capture the dynamical aspects of the systems. That is, from the initial state of the system (the initial graph), the application of rules successively changes the system state.

2.1 Basic Definitions

Definition 1 (Graph, Graph morphism)**.** *A **graph** $G = (V_G, E_G, src_G, trg_G)$ consists of a set of vertices V_G, a set of edges E_G, a source and a target function $src_G, trg_G : E_G \to V_G$. A **(partial) graph morphism** $g : G \to H$ from a graph G to a graph H is a tuple $g = (g_{Vert}, g_{Edge})$ consisting of two partial functions $g_{Vert} : V_G \to V_H$ and $g_{Edge} : E_G \to E_H$ which are weakly homomorphic, i.e., $g_{Vert} \circ src_G \geq src_H \circ g_{Edge}$ and $g_{Vert} \circ trg_G \geq trg_H \circ g_{Edge}$.*[1] *A morphism g is called total/ injective if both components are total/ injective, respectively.*

The weak commutativity used above means that everything that is preserved (mapped) by the morphism must be compatible, that is, every edge that is mapped by g_{Edge} must be compatible with the mapping of its source and target vertices by g_{Vert}. The term "weak" is used because the compatibility is just required on preserved items, not on all items. A typed graph is a graph equipped with a morphism t^G to a fixed graph of types.

Definition 2 (Typed Graph, Typed Graph Morphism)**.** *A **typed graph** G^T is a tuple $G^T = (G, t^G, T)$, where G and T are graphs and $t^G : G \to T$ is a total graph morphism called **typing morphism**. A **typed graph morphism** between graphs G^T and H^T with type graph T is a morphism $g : G \to H$ such that $t^G \geq t^H \circ g$ (that is, g may only map elements of the same type).*

A rule specifies a possible behaviour of the system. It consists of a left-hand side, describing items that must be present in a state to enable the rule application and a right-hand side, expressing items that will be present after the rule application. We require that rules do not collapse vertices or edges (are injective) and do not delete vertices.

[1]$\geq$ is the usual relation between partial functions meaning "more defined than". Considering $g^{\blacktriangledown}_{Edge} : dom(g_{Edge}) \hookrightarrow E_G$ and $g_{Edge}! : dom(g_{Edge}) \to E_H$ the g_{Edge} domain inclusion and restriction, respectively, we write $g_{Vert} \circ src_G \geq src_H \circ g_{Edge}$ iff $g_{Vert} \circ src_G \circ g^{\blacktriangledown}_{Edge} = src_H \circ g_{Edge}!$, and we write $g_{Vert} \circ trg_G \geq trg_H \circ g_{Edge}$ iff $g_{Vert} \circ trg_G \circ g^{\blacktriangledown}_{Edge} = trg_H \circ g_{Edge}!$.

The imposed restrictions do not represent a several limitation for many practical applications. The purpose of the graph grammars that we have in mind in this thesis is the specification of concurrent and reactive systems. The components of a left rule-graph must represent resources that shall not be identified by a transformation (rule application). And the deletion of a node can be simulated by using extra edges or, depending on the case, by leaving the node isolated (BALDAN; CORRADINI; KÖNIG, 2003). Then, it must not affect the expressiveness of the formalism. Furthermore, it leads us to a more simple theory: if we allowed deletion of nodes, extra conditions (such as the occurrence of dangling edges) should be considered when applying a rule to a state-graph.

Definition 3 (Rule)**.** *Let T be a graph. A* ***rule*** *with respect to T is an injective typed graph morphism $\alpha : L^T \rightarrow R^T$ from a typed graph L^T to a typed graph R^T, such that $\alpha_{Vert} : V_L \rightarrow V_R$ is a total function on the set of vertices.*

A graph grammar is composed of a *type graph*, characterizing the types of vertices and edges allowed in a system, an *initial graph*, representing the initial state of a system and a *set of rules*, describing the possible state changes that can occur in a system.

Definition 4 (Graph Grammar)**.** *A* ***(typed) graph grammar*** *is a tuple $GG = (T, G0, R)$, such that T is a type graph (the type of the grammar), $G0$ is a graph typed over T (the initial graph of the grammar) and R is a set of rules with respect to type T .*

Given a rule α and a state G, we say that this rule is applicable in this state if there is a match m, that is, an image of the left-hand side of the rule in the state. The operational behaviour of a graph grammar is defined in terms of rule applications. In what follows, $A \uplus B$ denotes the disjoint union of sets A and B and $rng(f)$ denotes the range of function f, that is, the image of the domain of f.

Definition 5 (Match, Rule Application)**.** *Given a rule $\alpha : L^T \rightarrow R^T$ with respect to a type graph T, a* ***match*** *of a rule α in a typed graph G^T is a total typed graph morphism $m : L^T \rightarrow G^T$ which is injective on edges. A* ***rule application*** *$G^T \overset{(\alpha,m)}{\Longrightarrow} H^T$, or the application of α to a typed graph G^T at match m, generates a typed graph $H^T = (H, t^H, T)$, with $H = (V_H, E_H, src_H, trg_H)$, as follows:*

Vertices of H:

$$V_H = V_G \uplus (V_R - \alpha_{Vert}(V_L))$$

Edges of H:

$$E_H = (E_G - m_{Edge}(E_L)) \uplus E_R$$

Source and target functions of H:

$$src_H(e) = \begin{cases} src_G(e) & \textit{if } e \in (E_G - m_{Edge}(E_L)) \\ \overline{m}(src_R(e)) & \textit{if } e \in E_R \end{cases}$$

$$trg_H(e) = \begin{cases} trg_G(e) & \textit{if } e \in (E_G - m_{Edge}(E_L)) \\ \overline{m}(trg_R(e)) & \textit{if } e \in E_R \end{cases}$$

where $\overline{m} : V_R \rightarrow V_H$ is defined by

$$\overline{m}(v) = \begin{cases} m_{Vert}(v0) & \textit{if } v \in rng(\alpha_{Vert}) \textit{ and } v = \alpha_{Vert}(v0) \\ v & \textit{otherwise} \end{cases}$$

Typing morphism: *The morphism $t^H = (t^H_{Vert}, t^H_{Edge})$ from H to T is*

$$t^H_{Vert}(v) = \begin{cases} t^G_{Vert}(v) & \text{if } v \in V_G \\ t^R_{Vert}(v) & \text{if } v \in (V_R - \alpha_{Vert}(V_L)) \end{cases}$$

$$t^H_{Edge}(e) = \begin{cases} t^G_{Edge}(e) & \text{if } e \in (E_G - m_{Edge}(E_L)) \\ t^R_{Edge}(e) & \text{if } e \in E_R \end{cases}$$

Intuitively, the application of α to G at the match m first removes from G the image of the edges in L. Then, graph G is extended by adding the new nodes in R (i.e., the nodes in $V_R - \alpha_{Vert}(V_L)$) and the edges of R. This construction can be described by a pushout in a suitable category of typed graphs (LöWE, 1993).

2.2 Working Example: The Token Ring Protocol

We illustrate the use of graph grammars specifying the token-ring protocol. This protocol is used to control the access of various stations to a shared transmission medium in a ring topology network (TANENBAUM, 2002). According to the protocol, a special bit pattern, called the token, is transmitted from station to station in only one direction. When a station wants to send some content through the network, it waits for the token, holds it, and sends the message (data frame) to the ring. The message circulates the ring and all stations may copy its contents. When the message completes the cycle, it is received by the originating station, which then removes the message from the ring and sends the token to the next station, restarting the cycle. If only one token exists, only one station may be transmitting at a given time. Here we will model a token-ring protocol in an environment in which new stations may be added at any time.

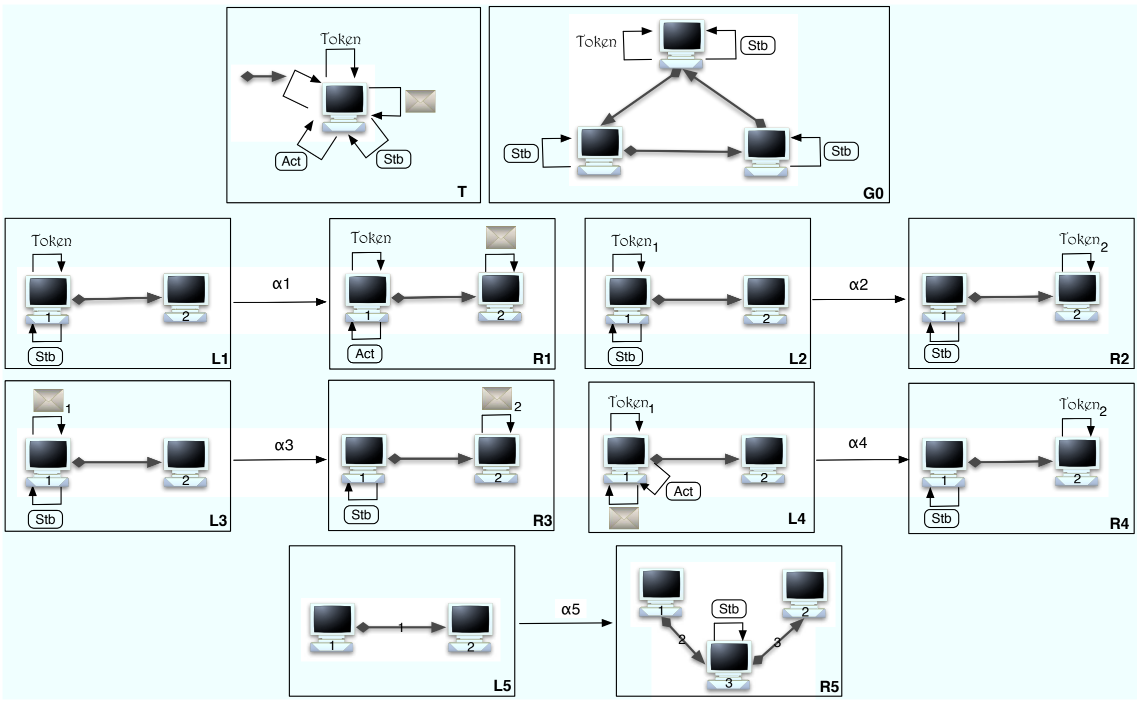
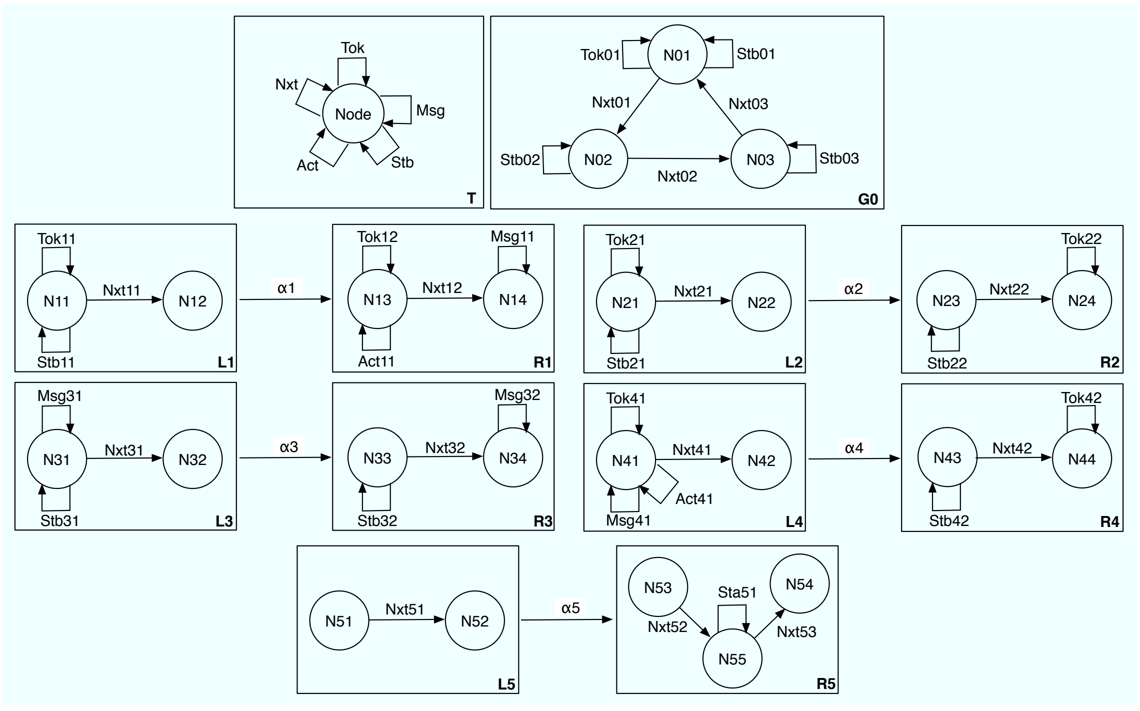


Figure 2.1: Token Ring Graph Grammar

Figure 2.1 illustrates the graph grammar for the example. The type graph T defines a single type of node (Node) , and five types of edges (Message), Token (Token),



$$\alpha 1\begin{cases}\alpha 1_{Vert}(\mathsf{N11}) = \mathsf{N13}\\ \alpha 1_{Vert}(\mathsf{N12}) = \mathsf{N14}\\ \alpha 1_{Edge}(\mathsf{Tok11}) = \mathsf{Tok12}\\ \alpha 1_{Edge}(\mathsf{Nxt11}) = \mathsf{Nxt12}\end{cases} \qquad \alpha 2\begin{cases}\alpha 2_{Vert}(\mathsf{N21}) = \mathsf{N23}\\ \alpha 2_{Vert}(\mathsf{N22}) = \mathsf{N24}\\ \alpha 2_{Edge}(\mathsf{Stb21}) = \mathsf{Stb22}\\ \alpha 2_{Edge}(\mathsf{Nxt21}) = \mathsf{Nxt22}\end{cases}$$

$$\alpha 3\begin{cases}\alpha 3_{Vert}(\mathsf{N31}) = \mathsf{N33}\\ \alpha 3_{Vert}(\mathsf{N32}) = \mathsf{N34}\\ \alpha 3_{Edge}(\mathsf{Stb31}) = \mathsf{Stb32}\\ \alpha 3_{Edge}(\mathsf{Nxt31}) = \mathsf{Nxt32}\end{cases} \qquad \alpha 4\begin{cases}\alpha 4_{Vert}(\mathsf{N41}) = \mathsf{N43}\\ \alpha 4_{Vert}(\mathsf{N42}) = \mathsf{N44}\\ \alpha 4_{Edge}(\mathsf{Nxt41}) = \mathsf{Nxt42}\end{cases}$$

$$\alpha 5\begin{cases}\alpha 5_{Vert}(\mathsf{N51}) = \mathsf{N53}\\ \alpha 5_{Vert}(\mathsf{N52}) = \mathsf{N54}\end{cases}$$

Figure 2.2: Alternative Definition of the Token Ring GG

↔ (Next), Act (Active Station) and Stb (Standby Station). 🖥 represents a network station and ✉ defines a frame of data. The stations are connected by edges of type ↔ . The Token represents a special signal which enables the station to start the transmission. Every station is either an active station (Act), meaning that the station is transmitting a message on the network, or a standby station (Stb). There can be only one active station on a ring at a time. The initial graph $\mathsf{G0}$ defines a ring with three nodes. Initially the Token is at a specific station and no station is transmitting information on the network (all stations have a Stb edge).

The behaviour of the protocol is modeled by the rules. In the graphical representation, usually the morphism is not explicitly represented; we assume that items of a graph are mapped to items with same names. A standby node with a token edge may retain this edge and send a message, becoming an active station (rule $\alpha 1$), or pass the token to the next node (rule $\alpha 2$). When a message is received by a standby node, rule $\alpha 3$ can be applied and the message is passed to the next node. If the receiving node is an active station, then rule $\alpha 4$ can be applied, removing the message from the ring and sending the token to the next station. Rule $\alpha 5$ is applied to insert a new station into the ring. This model has an infinite state-space and generates an infinite number of possible computations.

Although the graphical representation shown in Figure 2.1 is natural, to obtain a relational representation of a graph grammar we will assume, without loss of generality, that all items (vertices or edges) that appear in graph grammar have different names. Thus, we need to explicitly show the morphisms when defining the rules of a grammar. In our example, a grammar that is (behaviourally) equivalent to the one shown in Figure 2.1 is depicted in Figure 2.2 (morphisms are shown below the graphical representation). Note that, in the definition of morphism $\alpha 1$, the edge Stb11 of $L1$ is not mapped, this means that it is deleted by this rule; edge Msg11 is not in the image of $\alpha 1$, and therefore is created by this rule.

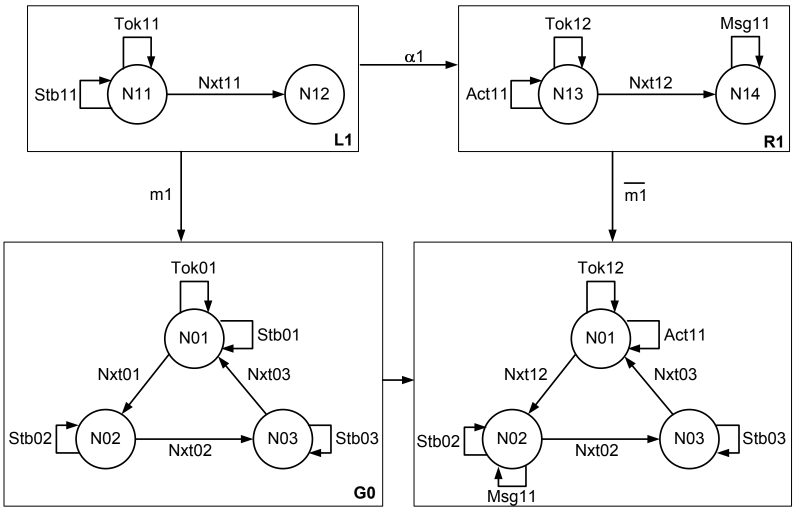


Figure 2.3: Application of the Rule $\alpha 1$ to Initial Graph

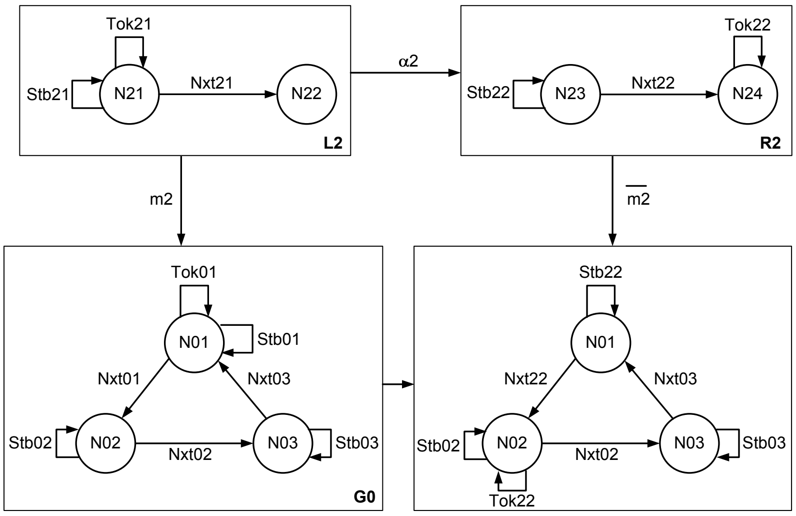


Figure 2.4: Application of the Rule $\alpha 2$ to Initial Graph

Examples of rule applications are presented in Figure 2.3 and in Figure 2.4. In Figure 2.3 rule $\alpha 1$ is applied to the initial graph $G0$, modeling the situation where station $N01$ sends a message through the network. In Figure 2.4 rule $\alpha 2$ is applied to the initial graph $G0$. In this case station $N01$ remains in standby and passes the token to the next station.

Both rules $\alpha 1$ and $\alpha 2$ compete to update the same portion of the state. In this case, one of the rules is (non-deterministically) chosen to be applied, representing the fact that a station may decide to hold the token and send a message, or to forward the token.

3 RELATIONAL REPRESENTATION OF GRAPH GRAMMARS

Aiming to define a theory that allows the formulation of properties and the development of proofs for systems specified as graph grammars, we propose a representation of graph grammars by relational structures (i.e., by structures with relations only). Our approach is equivalent to the SPO-approach (ROZENBERG, 1997), and our choice for such encoding relies on the possibility of using a theorem prover to semi-automate the proofs.

3.1 Relational Structures

A relational structure (COURCELLE, 1997) is a tuple formed by a set and by a family of relations over this set.

Definition 6 (Relational Structures). *Let $\mathcal{R}$ be a finite set of relation symbols, where each $R \in \mathcal{R}$ has an associated positive integer called its arity, denoted by $\rho(R)$. An $\mathcal{R}$-**structure** is a tuple $S = \langle D_S, (R_S)_{R\in\mathcal{R}} \rangle$ such that D_S is a possibly empty set called the domain of S and each R_S is a $\rho(R)$-ary relation on D_S, i.e., a subset of $D_S^{\rho(R)}$. $R(d_1, \dots, d_n)$ holds in S if and only if $(d_1, \dots, d_n) \in R_S$, where $d_1, \dots, d_n \in D_S$. The class of $\mathcal{R}$-structures is denoted by $STR(\mathcal{R})$.*

We start by defining a relational structure to model graphs, and establishing a relational representation for graph morphisms, typed graphs and rules, which will later be used to build the relational structure associated to a graph grammar. A relational structure representing a graph G is a tuple composed of a set, the domain of the structure, representing all vertices and edges of G and by two finite relations: a unary relation, i.e. a set $vert_G$, defining the set of vertices of G and a ternary relation inc_G representing the incidence relation between vertices and edges of G.

Definition 7 (Relational Structure Representing a Graph). *Let $\mathcal{R}_{gr} = \{vert, inc\}$ be a set of relations, where $vert$ is unary and inc is ternary. Given a graph $G = (V_G, E_G, src_G, trg_G)$, a **relational structure representing** G is a $\mathcal{R}_{gr}$-structure $|G| = \langle D_G, (R_G)_{R\in\mathcal{R}_{gr}} \rangle$, where:*

- *$D_G = V_G \cup E_G$ is the union of sets of vertices and edges of G.*
- *$vert_G = V_G$, i.e. $vert_G(x)$ iff $x \in V_G$;*
- *$inc_G \subseteq E_G \times V_G \times V_G$, with $inc_G(x, y, z)$ iff $x \in E_G \wedge src_G(x) = y \wedge trg_G(x) = z$;*

Example 1. *The typed graph $G0$ depicted in Figure 2.2 can be defined by the relational structure $|G0| = \langle D_{G0}, \{vert_{G0}, inc_{G0}\}\rangle$, where*

Domain: $D_{G0} = V_{G0} \cup E_{G0}$ *with*

$$V_{G0} = \{\mathsf{N01}, \mathsf{N02}, \mathsf{N03}\}$$
$$E_{G0} = \{\mathsf{Tok01}, \mathsf{Stb01}, \mathsf{Stb02}, \mathsf{Stb03}, \mathsf{Nxt01}, \mathsf{Nxt02}, \mathsf{Nxt03}\}$$

Relations:

$$vert_{G0} = \{\mathsf{N01}, \mathsf{N02}, \mathsf{N03}\}$$
$$inc_{G0} = \{(\mathsf{Tok01},\ \mathsf{N01},\ \mathsf{N01}), (\mathsf{Stb01},\ \mathsf{N01},\ \mathsf{N01}), (\mathsf{Nxt01},\ \mathsf{N01},\ \mathsf{N02}), (\mathsf{Stb02}, \mathsf{N02}, \mathsf{N02}), (\mathsf{Nxt02},\ \mathsf{N02},\ \mathsf{N03}), (\mathsf{Stb03}, \mathsf{N03}, \mathsf{N03}), (\mathsf{Nxt03}, \mathsf{N03}, \mathsf{N01})\}.$$

Proposition 1. *The relational structure $|G|$ is well-defined.*

Proof. By definition, the relational structure $|G|$ has the same set of vertices of G. The ternary relation inc_G specifies the set of directed edges. Each edge x of G is related, by inc_G, to (and only to) two vertices: its source and target vertices. Nothing else belongs to inc_G. Then, $|G|$ defines graph G. $\square$

The relational representation of a graph morphism g from a graph G to a graph H is obtained through two binary relations: one to relate vertices (g_V) and other to relate edges (g_E). Since these relations just map vertices and edges names, we have to impose some restrictions to ensure that they represent a morphism. The *type consistency conditions* state that if two vertices are related by g_V then the first one must be a vertex of G and the second one a vertex of H, and if two edges are related by g_E, then the first one must be an edge of G and the second one an edge of H. The *(morphism) commutativity condition* assures that the mapping of edges preserves the mapping of source and target vertices.

Definition 8 (Relational Graph Morphism). *Let $|G| = \langle V_G \cup E_G, \{vert_G, inc_G\}\rangle$ and $|H| = \langle V_H \cup E_H, \{vert_H, inc_H\}\rangle$ be $\mathcal{R}_{gr}$-structures representing graphs. A **relational graph morphism** $|g|$ **from** $|G|$ **to** $|H|$ is defined by a set $|g| = \{g_V, g_E\}$ of binary relations where:*

- $g_V \subseteq V_G \times V_H$ *is a partial function that relates vertices of $|G|$ to vertices of $|H|$;*
- $g_E \subseteq E_G \times E_H$ *is a partial function that relates edges of $|G|$ to edges of $|H|$;*

such that the following conditions are satisfied:

- ***Type Consistency Conditions.*** $\forall x, x'$,
  $[g_V(x, x')] \Rightarrow vert_G(x) \wedge vert_H(x')$*; and*
  $[g_E(x, x')] \Rightarrow \exists y, y', z, z'[inc_G(x, y, z) \wedge inc_H(x', y', z')]$*;*
- ***Morphism Commutativity Condition.*** $\forall x, y, z, x', y', z'$,
  $[g_E(x, x') \wedge inc_G(x, y, z) \wedge inc_H(x', y', z') \Rightarrow\ g_V(y, y')\ \wedge\ g_V(z, z')]$.

g is called total/injective if relations g_V and g_E are total/injective functions, respectively.

Proposition 2. *A relational graph morphism $g = \{g_V, g_E\}$ from $|G|$ to $|H|$ is a well-defined graph morphism from graph G to graph H.*

Proof. We have g_V and g_E denoting partial functions. According to the type consistency conditions, they relate vertices and edges of G and H, respectively. Moreover, due to the morphism commutativity condition, every edge that is related by g_E must be compatible with the relations established by g_V. In other words, if an edge x is related by g_E to an edge x', then its source and target vertices must be related by g_V, i.e., the weak commutativity holds. $\square$

A typing morphism is a graph morphism that has the role of typing all elements of a graph G over a graph T. Thus, its relational definition is the same as graph morphisms, with the restriction that both relations must represent total functions.

Definition 9 (Relational Typing Morphism)**.** *Let $|G|$ and $|T|$ be $\mathcal{R}_{gr}$-structures representing graphs. A **relational typing morphism from** $|G|$ **over** $|T|$ is defined by a total relational graph morphism $|t^G| = \{t_V^G, t_E^G\}$ from $|G|$ to $|T|$.*

Example 2. *The relational typing morphism from $|G0|$ over $|T|$ (see Figure 2.2) is defined by $|t^{G0}| = \{t_V^{G0}, t_E^{G0}\}$, with $t_V^{G0} = \{(\mathsf{N01}, \mathsf{Node}), (\mathsf{N02}, \mathsf{Node}), (\mathsf{N03}, \mathsf{Node})\}$ and $t_E^{G0} = \{(\mathsf{Tok01}, \mathsf{Tok}), (\mathsf{Stb01}, \mathsf{Stb}), (\mathsf{Stb02}, \mathsf{Stb}), (\mathsf{Stb03}, \mathsf{Stb}), (\mathsf{Nxt01}, \mathsf{Nxt}), (\mathsf{Nxt02}, \mathsf{Nxt}), (\mathsf{Nxt03}, \mathsf{Nxt})\}$.*

Proposition 3. *A relational typing morphism is a well-defined typing morphism.*

Proof. Following Proposition 2 a relational typing morphism is a well-defined graph morphism. Since both relations in a relational typing morphism must be total functions, it is a well-defined typing morphism. $\square$

The relational representation of a typed graph $G^T = (G, t^G, T)$ is defined by two $\mathcal{R}_{gr}$-structures representing G and T and by a relational typing morphism, which defines exactly the typing morphism t^G.

Definition 10 (Relational Representation of a Typed Graph)**.** *Given a typed graph $G^T = (G, t^G, T)$ with $t^G = (t^G_{Vert}, t^G_{Edge})$, **a relational representation of** G^T is given by a tuple $|G^T| = \langle |G|, |t^G|, |T| \rangle$ where:*

- *$|G|$ and $|T|$ are $\mathcal{R}_{gr}$-structures representing G and T respectively;*
- *$|t^G| = \{t^G_{Vert}, t^G_{Edge}\}$ is a relational typing morphism from $|G|$ over $|T|$.*

Proposition 4. *The relational representation of a typed graph is well-defined.*

Proof. By Proposition 1 the relational representation of graphs is well-defined and by Proposition 3 the relational typing morphism is well-defined. Since the definition of the relational typing morphism guarantees that it represents the same typing morphism given, then the relational representation $|G^T|$ defines the same typed graph G^T. $\square$

A relational graph morphism is also the basis of the relational definition of a relational typed graph morphism from a graph G to a graph H. Since both graphs are typed over the same graph T, a *(typed morphism) compatibility condition* assures that the mappings of vertices and edges preserve types.

Definition 11 (Relational (Typed) Graph Morphism)**.** *Let $|G|$, $|H|$ and $|T|$ be $\mathcal{R}_{gr}$ -structures representing graphs and $|t^G| = \{t_V^G, t_E^G\}$ and $|t_H| = \{t_V^H, t_E^H\}$ be relational typing morphisms from $|G|$ and $|H|$ over $|T|$, respectively. A **relational (typed) graph morphism from** $|G^T|$ **to** $|H^T|$ is defined by a relational graph morphism $|g| = \{g_V, g_E\}$ from $|G|$ to $|H|$, such that the typed morphism compatibility condition is satisfied:*

- ***(Typed Morphism) Compatibility Condition.*** $\forall x, x', y,$
  $[g_V(x,x') \wedge t_V^G(x,y) \Rightarrow t_V^H(x',y)]$*; and*
  $[g_E(x,x') \wedge t_E^G(x,y) \Rightarrow t_E^H(x',y)]$.

Proposition 5. *The relational representation of a typed graph morphism is well-defined.*

Proof. Following Proposition 2, a relational graph morphism is a well-defined graph morphism. The (typed morphism) compatibility condition guarantees that the relational typed graph morphism only maps elements of the same type. □

Given a rule $\alpha : L^T \to R^T$, its relational representation is given by the relational representation of typed graphs L^T and R^T, together with a relational typed morphism which must define the same morphism given. Note that, since a rule does not delete vertices, the function α_{Vert} must be total.

Definition 12 (Relational Representation of a Rule)**.** *Given a rule* $\alpha : L^T \to R^T$, $\alpha = (\alpha_{Vert}, \alpha_{Edge})$, ***a relational representation of*** α *is given by a tuple* $\langle |L^T|, |\alpha|, |R^T| \rangle$ *where:*

- $|L^T| = \langle |L|, |t^L|, |T| \rangle$ *and* $|R^T| = \langle |R|, |t^R|, |T| \rangle$ *are relational representations of typed graphs* L^T *and* R^T*, respectively;*
- $|\alpha| = \{\alpha_{Vert}, \alpha_{Edge}\}$ *is a relational typed graph morphism from* $|L^T|$ *to* $|R^T|$.

Example 3. *The relational typed graph morphism of rule* $\alpha 1$ *illustrated in Figure 2.2 is defined by* $|\alpha_1| = \{\alpha_{1_V}, \alpha_{1_E}\}$*, where* $\alpha_{1_V} = \{(\mathsf{N11}, \mathsf{N13}), (\mathsf{N12}, \mathsf{N14})\}$ *and* $\alpha_{1_E} = \{(\mathsf{Tok11}, \mathsf{Tok12}), (\mathsf{Nxt11}, \mathsf{Nxt12})\}$*. The relational typing morphisms from* $|L1|$ *and* $|R1|$ *over* $|T|$ *are respectively given by* $t_V^{L1} = \{(\mathsf{N11}, \mathsf{Node}), (\mathsf{N12}, \mathsf{Node})\}$, $t_E^{L1} = \{(\mathsf{Tok11}, \mathsf{Tok}), (\mathsf{Stb11}, \mathsf{Stb}), (\mathsf{Nxt11}, \mathsf{Nxt})\}$ *and* $t_V^{R1} = \{(\mathsf{N13}, \mathsf{Node}), (\mathsf{N14}, \mathsf{Node})\}$, $t_E^{R1} = \{(\mathsf{Tok12}, \mathsf{Tok}), (\mathsf{Act11}, \mathsf{Act}), (\mathsf{Nxt12}, \mathsf{Nxt}), (\mathsf{Msg11}, \mathsf{Msg})\}$.

Proposition 6. *A relational representation of a rule is well-defined.*

Proof. By Proposition 4 the relational representation of typed graphs is well defined and by Proposition 5 the relational representation of a typed graph morphism is well-defined. Also, the definition of the relational typed graph morphism guarantees that it represents the same morphism given. Then, the relational graph morphism is injective with the component that relates vertices total. □

Given a graph grammar $GG = (T, G0, R)$, we define a relational structure $|GG|$ associated to it as a tuple composed of a set and a collection of relations. The set describes the domain of the structure. The relations define the type graph, the initial graph and the rules. The type graph is defined by relations of a $\mathcal{R}_{gr}$-structure representing T. The initial graph $G0$, and the left- and right-hand sides of rules are specified by relations of $\mathcal{R}_{gr}$-structures representing graphs, which are typed over T by relational typing morphisms. Relational typed graph morphisms map the graphs of left-hand side and right-hand side of rules.

Definition 13 (Relational Structure Associated to a Graph Grammar)**.** *Let* $\mathcal{R}_{GG} = \{vert_T,\ inc_T,\ vert_{G0},\ inc_{G0},\ t_V^{G0},\ t_E^{G0}, (vert_{Li},\ inc_{Li},\ t_V^{Li},\ t_E^{Li}, vert_{Ri}, inc_{Ri},\ t_V^{Ri},\ t_E^{Ri},\ \alpha_{i_V},\ \alpha_{i_E})_{i \in \{1,\dots,n\}}\}$ *be a set of relation symbols. Given a graph grammar* $GG = (T, G0, R)$ *where* R *has cardinality* n, ***the*** $\mathcal{R}_{\mathbf{GG}}$***-structure associated to*** GG*, denoted by* $|GG|$*, is the tuple* $\langle D_{GG}, (r)_{r \in \mathcal{R}_{GG}} \rangle$[1] *where*

[1]In order to simplify the reading we omit the subscript GG in relations.

- *$D_{GG} = V_{GG} \cup E_{GG}$ is the set of vertices and edges of the graph grammar, where: $V_{GG} \subseteq V_T \cup V_{G0} \cup (V_{Li} \cup V_{Ri})_{i\in\{1,\ldots,n\}}$ and $E_{GG} \subseteq E_T \cup E_{G0} \cup (E_{Li} \cup E_{Ri})_{i\in\{1,\ldots,n\}}$ with $V_T \cap V_{G0} \cap (V_{Li} \cap V_{Ri})_{i\in\{1,\ldots,n\}} = \varnothing$ and $E_T \cap E_{G0} \cap (E_{Li} \cap E_{Ri})_{i\in\{1,\ldots,n\}} = \varnothing$;*
- *$vert_T$ and inc_T model the **type graph**. They are the relations of a $\mathcal{R}_{gr}$-structure $|T| = \langle V_T \cup E_T, \{vert_T, inc_T\}\rangle$ representing graph T.*
- *$vert_{G0}$, inc_{G0}, t_V^{G0} and t_E^{G0} model the **initial graph typed over** T, i.e., they are the relations that compose the relational representation of $G0^T$.*
- *Each collection $(vert_{Li},\ inc_{Li},\ t_V^{Li},\ t_E^{Li},\ vert_{Ri}, inc_{Ri},\ t_V^{Ri},\ t_E^{Ri},\ \alpha_{i_V},\ \alpha_{i_E})$ defines a **rule**:*
  - *$vert_{Li},\ inc_{Li},\ t_V^{Li}$ and t_E^{Li} model the **left-hand side** of the rule, i. e., they are the relations of the relational representation of Li^T.*
  - *$vert_{Ri}, inc_{Ri},\ t_V^{Ri}$ and t_E^{Ri} model the **right-hand side** of the rule, i. e., they are the relations of the relational representation of Ri^T.*
  - *α_{i_V} and α_{i_E} are relations of $|\alpha_i|$, which defines a **relational typed graph morphism** from $|Li^T|$ to $|Ri^T|$, such that the tuple $\langle |Li^T|, |\alpha_i|, |Ri^T|\rangle$ is a relational representation of rule $\alpha_i : Li^T \to Ri^T$.*

Example 4. *The relational structure that represents the graph grammar of the example described in Section 2.2 is:*

$$\begin{aligned}|GG| = \langle V_{GG} \cup E_{GG}, \{&vert_T,\ inc_T,\ vert_{G0},\ inc_{G0},\ t_{G0_V},\ t_{G0_E},\\ &vert_{L1},\ inc_{L1},\ t_V^{L1},\ t_E^{L1},\ vert_{R1},\ inc_{R1},\ t_V^{R1},\ t_E^{R1},\ \alpha_{1_V},\ \alpha_{1_E},\\ &vert_{L2},\ inc_{L2},\ t_V^{L2},\ t_E^{L2},\ vert_{R2},\ inc_{R2},\ t_V^{R2},\ t_E^{R2},\ \alpha_{2_V},\ \alpha_{2_E},\\ &vert_{L3},\ inc_{L3},\ t_V^{L3},\ t_E^{L3},\ vert_{R3},\ inc_{R3},\ t_V^{R3},\ t_E^{R3},\ \alpha_{3_V},\ \alpha_{3_E},\\ &vert_{L4},\ inc_{L4},\ t_V^{L4},\ t_E^{L4},\ vert_{R4},\ inc_{R4},\ t_V^{R4},\ t_E^{R4},\ \alpha_{4_V},\ \alpha_{4_E},\\ &vert_{L5},\ inc_{L5},\ t_V^{L5},\ t_E^{L5},\ vert_{R5},\ inc_{R5},\ t_V^{R5},\ t_E^{R5},\ \alpha_{5_V},\ \alpha_{5_E}\}\rangle,\end{aligned}$$

where:

(Domain) Vertex names, $V_{GG} = \{$ Node, N01, ..., N03, N11, ..., N14, N21, ..., N24, N31, ..., N34, N41, ..., N44, N51, ..., N55$\}$;

Edges names, $E_{GG} = \{$ Nxt, Nxt01, Nxt02, Nxt03, Nxt11, Nxt12, Nxt21, Nxt22, Nxt31, Nxt32, Nxt41, Nxt42, Nxt51, Nxt52, Nxt53, Tok, Tok01, Tok11, Tok12, Tok21, Tok22, Tok41, Tok42, Msg, Msg11, Msg31, Msg32, Msg41, Act, Act11, Act41, Stb, Stb01, Stb02, Stb03, Stb11, Stb21, Stb22, Stb31, Stb32, Stb42, Stb51 $\}$.

(Type Graph T) Vertices, $vert_T = \{$ Node $\}$;

Edges, $inc_T = \{$ (Nxt, Node, Node), (Tok, Node, Node), (Msg, Node, Node), (Stb, Node, Node), (Act, Node, Node) $\}$.

(Initial Graph $G0$) Vertices, $vert_{G0} = \{$ N01, N02, N03 $\}$;

Edges, $inc_{G0} = \{$ (Tok01, N01, N01), (Stb01, N01, N01), (Nxt01, N01, N02), (Stb02, N02, N02),

		$(\mathsf{Nxt02}, \mathsf{N02}, \mathsf{N03}), (\mathsf{Stb03}, \mathsf{N03}, \mathsf{N03}), (\mathsf{Nxt03}, \mathsf{N03}, \mathsf{N01})\ \}$;
Typing vertices,	t_V^{G0}	$= \{\ (\mathsf{N01}, \mathsf{Node}), (\mathsf{N02}, \mathsf{Node}), (\mathsf{N03}, \mathsf{Node})\ \}$;
Typing edges,	t_E^{G0}	$= \{\ (\mathsf{Tok01}, \mathsf{Tok}), (\mathsf{Stb01}, \mathsf{Stb}), (\mathsf{Nxt01}, \mathsf{Nxt}), (\mathsf{Stb02}, \mathsf{Stb}), (\mathsf{Nxt02}, \mathsf{Nxt}), (\mathsf{Stb03}, \mathsf{Stb}), (\mathsf{Nxt03}, \mathsf{Nxt})\ \}$.

(Rule 1) ***Left Graph*** $L1$:

Vertices,	$vert_{L1}$	$= \{\ \mathsf{N11}, \mathsf{N12}\ \}$;
Edges,	inc_{L1}	$= \{\ (\mathsf{Tok11}, \mathsf{N11}, \mathsf{N11}), (\mathsf{Stb11}, \mathsf{N11}, \mathsf{N11}), (\mathsf{Nxt11}, \mathsf{N11}, \mathsf{N12})\ \}$;
Typing vertices,	t_V^{L1}	$= \{\ (\mathsf{N11}, \mathsf{Node}), (\mathsf{N12}, \mathsf{Node})\ \}$;
Typing edges,	t_E^{L1}	$= \{\ (\mathsf{Tok11}, \mathsf{Tok}), (\mathsf{Stb11}, \mathsf{Stb}), (\mathsf{Nxt11}, \mathsf{Nxt})\ \}$.

Right Graph $R1$:

Vertices,	$vert_{R1}$	$= \{\ \mathsf{N13}, \mathsf{N14}\ \}$;
Edges,	inc_{R1}	$= \{\ (\mathsf{Tok12}, \mathsf{N13}, \mathsf{N13}), (\mathsf{Act11}, \mathsf{N13}, \mathsf{N13}), (\mathsf{Nxt12}, \mathsf{N13}, \mathsf{N14}), (\mathsf{Msg11}, \mathsf{N14}, \mathsf{N14})\ \}$;
Typing vertices,	t_V^{R1}	$= \{\ (\mathsf{N13}, \mathsf{Node}), (\mathsf{N14}, \mathsf{Node})\ \}$;
Typing edges,	t_E^{R1}	$= \{\ (\mathsf{Tok12}, \mathsf{Tok}), (\mathsf{Act11}, \mathsf{Act}), (\mathsf{Nxt12}, \mathsf{Nxt}), (\mathsf{Msg11}, \mathsf{Msg})\ \}$.

Relational Rule $\alpha1$:

Mapping vertices,	$\alpha 1_V$	$= \{\ (\mathsf{N11}, \mathsf{N13}), (\mathsf{N12}, \mathsf{N14})\ \}$;
Mapping edges,	$\alpha 1_E$	$= \{\ (\mathsf{Tok11}, \mathsf{Tok12}), (\mathsf{Nxt11}, \mathsf{Nxt12})\ \}$.

(Rules 2 to 5 are analogous)

Proposition 7. *The relational structure $|GG|$ is well-defined.*

Proof. Follows immediately from Propositions 1, 4 and 6. □

3.2 Rule Applications as First-Order Definable Transductions

In this section, inspired by the definition of monadic second-order definable transduction, introduced in (COURCELLE, 1997), we show how to define rule applications as graph grammar transformations. This approach will allow a graph grammar theory to be defined, which will be later used to verify properties of distributed and reactive systems.

A monadic second-order definable transduction (COURCELLE, 1997) replaces for graphs the notion of finite automaton used for transformations of words or trees. It is defined through a tuple $(\varphi, \psi, (\theta_q)_{q \in \mathcal{Q}})$ of monadic second-order formulas (GUREVICH, 1985) that specifies a $\mathcal{Q}$-structure $T = \langle D_T, (R_T)_{R \in \mathcal{Q}} \rangle$ based on an $\mathcal{R}$-structure $S = \langle D_S, (R_S)_{R \in \mathcal{R}} \rangle$. The first formula of the tuple, φ, establishes a condition to be satisfied in order to make the transduction possible. The following formula ψ defines the domain of the relation T. Finally, for each relation $q \in \mathcal{Q}$, a formula θ defines the elements of the T domain that belong to the relation. In the original definition, it is possible to make k copies of the original structure S before redefining the relations q, to obtain the new structure T. Next, we present the definition of first-order definable transductions (via first-order formulas) without copies of the original structure, which is enough to represent rule applications as graph-grammar transformations.

Definition 14 (First-Order Definable Transduction)**.** *Let $\mathcal{R}$ and $\mathcal{Q}$ be two finite ranked sets of relation symbols. Let $\mathcal{W}$ be a finite set of variables (parameters) and $FO(\mathcal{R}, \mathcal{W})$ be the set of first-order formulas over $\mathcal{R}$, with free variables in $\mathcal{W}$. A $(\mathcal{Q}, \mathcal{R})$-**definition scheme** is a tuple $\Delta = (\varphi, \psi, (\theta_q)_{q \in \mathcal{Q}})$, where $\varphi \in FO(\mathcal{R}, \mathcal{W})$, $\psi \in FO(\mathcal{R}, \mathcal{W} \cup \{x_1\})$ and $\theta_q \in FO(\mathcal{R}, \mathcal{W} \cup \{x_1, \ldots, x_{\rho(q)}\})$.*

*These formulas are intended to define a structure T in $STR(\mathcal{Q})$ from a structure S in $STR(\mathcal{R})$ in the following way: let $S \in STR(\mathcal{R})$ and γ be a $\mathcal{W}$-assignment in S, **a $\mathcal{Q}$-structure** T with domain $D_T \subseteq D_S$ **is defined in** (S, γ) **by** Δ if:*

1. *$(S, \gamma) \models \varphi$. Formula φ establishes a condition to be fulfilled so that the translation is possible. I.e., T is defined only if φ holds true in S for some γ.*

2. *$D_T = \{d \in D_S \mid (S, \gamma, d) \models \psi\}$. Assuming that 1. is satisfied, formula ψ defines the domain of T as the set of elements in the S domain that satisfy ψ for γ.*

3. *for each $q \in \mathcal{Q}$, $q_T = \{(d_1, \ldots, d_t) \in D_T^t \mid (S, \gamma, d_1, \ldots, d_t) \models \theta_q\}$, where $t = \rho(q)$. Formulas θ_q define the relation q_T for each $q \in \mathcal{Q}$.*

*Since T is associated in a unique way with S, γ and Δ whenever it is defined (whenever $(S, \gamma) \models \varphi$) we can use the functional notation $def_\Delta(S, \gamma)$ for T. A **transduction defined by** Δ is the relation $def_\Delta := \{(S, T) \mid T = def_\Delta(S, \gamma)$ for some $\mathcal{W}$-assignment γ in $S\} \subseteq STR(\mathcal{R}) \times STR(\mathcal{Q})$. $f \subseteq STR(\mathcal{R}) \times STR(\mathcal{Q})$ **is a FO-definable transduction**, if it is equal to def_Δ, for some $(\mathcal{Q}, \mathcal{R})$-definition scheme Δ. In the case where $\mathcal{W} = \varnothing$ we say that f is definable without parameters.*

A rule application may be described by a FO-definable transduction on relational structures associated to graph grammars. The result of the transduction over a graph grammar is another graph grammar whose initial state corresponds to the result of the application of a rule α_i at a match m to the initial state of the original grammar. The other components of the grammar remain unchanged (i.e., the resulting grammar has the same type graph and rules of the original one). In order to define rule application as a FO-definable transduction, we first introduce the relational representation of a match.

Definition 15 (Relational Representation of a Match)**.** *Given a match $m : L^T \to G^T$, $m = (m_{Vert}, m_{Edge})$, **a relational representation of match** m is given by a tuple $\langle |L^T|, |m|, |G^T| \rangle$ where:*

- *$|L^T| = \langle |L|, |t^L|, |T| \rangle$ and $|G^T| = \langle |G|, |t^G|, |T| \rangle$ are relational representations of typed graphs L^T and G^T, respectively;*

- *$|m| = \{m_{Vert}, m_{Edge}\}$ is a relational typed graph morphism from $|L^T|$ to $|G^T|$.*

Example 5. *The relational graph morphism $|m1| = \{m1_V, m1_E\}$ from $|L_1|$ to $|G0|$ (see Figure 2.3), where $m1_V = \{(\mathsf{N11}, \mathsf{N01}), (\mathsf{N12}, \mathsf{N02})\}$ and $m1_E = \{(\mathsf{Nxt11}, \mathsf{Nxt01}), (\mathsf{Tok11}, \mathsf{Tok01}), (\mathsf{Stb11}, \mathsf{Stb01})\}$ represents a relational match of rule $|\alpha_1|$ in $|G0|$. Both relations represent total functions and $m1_E$ is injective. Besides, the mapping respect the typed morphism compatibility condition. We can also notice, for instance, that the pair $(\mathsf{Nxt11}, \mathsf{Nxt01})$ preserves types, i.e, both clauses $t_E^{L1}(\mathsf{Nxt11}, \mathsf{Nxt})$ and $t_E^{G0}(\mathsf{Nxt01}, \mathsf{Nxt})$ hold (see Initial Graph and Left Graph definitions in Example 4).*

Proposition 8. *A relational representation of a match is well-defined.*

Proof. By Proposition 4 the relational representation of typed graphs is well defined and by Proposition 5 the relational representation of a typed graph morphism is well-defined. The definition of the relational typed graph morphism guarantees that it represents the same morphism given. Then, the relational graph morphism is total, with the component that relates edges injective. □

Now, a rule application is represented by a definable transduction (i.e., by a tuple of first-order formulas) that defines a $\mathcal{R}_{GG}$-structure $|GG|'$ (i.e., a graph grammar) based on another $\mathcal{R}_{GG}$-structure $|GG|$. Before applying the transduction, we must first fix a relational representation of a rule αi and a relational representation of a match m of αi in $G0^T$. Then, the $\mathcal{R}_{GG}$-definition scheme $\Delta = (\varphi, \psi, (\theta_q)_{q \in \mathcal{R}_{GG}})$ defines the relational structure $|GG|'$ from $|GG|$, which corresponds to the same grammar, excepted that $|G0|'$ (initial state of $|GG|'$) represents the result of the application of $|\alpha i|$ at match $|m|$ in $|G0|$. In Δ, φ ensures that $|m|$ effectively defines a match, ψ defines the domain of the resulting grammar (the same of original grammar) and each formula θ_q, $q \in \mathcal{R}_{GG}$, defines the elements that will be present in relations $q_{GG'}$, $q \in \mathcal{R}_{GG}$ of the resulting grammar. In fact, the collection (θ_q) defines the structure associated to graph grammar $|GG|'$. Since the type graph and the rules remain unchanged, the formulas that define these components are constructed in the obvious way (they are defined by relations of the original grammar). Formulas $\theta_{vert_{G0}}$, $\theta_{inc_{G0}}$, $\theta_{t_{G0_V}}$, $\theta_{t_{G0_E}}$ that define the resulting graph of the rule application are specified according to Definition 5. Table 3.1 presents the intuitive meaning and the notation used in θ specifications.

Definition 16 (Rule Application as FO-Definable Transduction)**.** *Let $GG = (T, G0, R)$ be a graph grammar such that the sets of edges and vertices of graphs T, $G0$, Li and Ri are disjoint, and let $|GG|$ be the relational structure associated to GG. Given a rule $\alpha_i : Li \to Ri$ of GG and a corresponding match $m : Li \to G0$, with the relational representations respectively given by $|\alpha_i| = \{\alpha_{i_V}, \alpha_{i_E}\}$ from $|Li|$ to $|Ri|$ and $|m| = \{m_V, m_E\}$ of $|Li|$ in $|G0|$, $\Delta = (\varphi, \psi, (\theta_q)_{q \in \mathcal{R}_{GG}})$, with $\mathcal{W} = \varnothing$, **defines a transduction that maps a graph grammar** $|GG|$ **to a graph grammar** $|GG|'$, such that $|G0|'$ (initial state of $|GG|'$) corresponds to the result of the application of rule $|\alpha_i|$ at match $|m|$ in $|G0|$ (initial state of $|GG|$), where:*

φ *expresses that $|m| = \{m_V, m_E\}$ defines a total relational typed graph morphism, with m_E injective. So, it must guarantee that the following conditions are satisfied.*

- *$|m|$ is a total relational graph morphism:*
  - *$m_V \subseteq V_{Li} \times V_{G0}$ is a total function:*

$$\forall x\Big(\Big(vert_{Li}(x)\Big) \Rightarrow \exists! x'\Big(m_V(x, x') \wedge vert_{G0}(x')\Big)\Big)$$

  - *$m_E \subseteq E_{Li} \times E_{G0}$ is a total function:*

$$\forall x, y, z\Big(\Big(inc_{Li}(x, y, z)\Big) \Rightarrow \exists! x', y', z'\Big(m_E(x, x') \wedge inc_{G0}(x', y', z')\Big)\Big)$$

  - *$\{m_V, m_E\}$ satisfies the Type Consistency and the Morphism Commutativity Conditions.*

- *$|m|$ is a relational typed graph morphism with m_E injective:*
  - *m_E is injective:*

$$\forall x, y \left(\Big(m_E(x,y) \Big) \Rightarrow \nexists x' \Big(m_E(x',y) \Big) \right)$$

  - *$\{m_V, m_E\}$ satisfy the Typed Morphism Compatibility Condition.*

ψ is the Boolean constant true (same domain).

θ_{vert_T}, θ_{inc_T} are, respectively, the formulas $vert_T(x)$ and $inc_T(x,y,z)$ (same type graph).

$\theta_{vert_{G0}}$ is the formula $vert_{G0}(x) \vee nvert_{Ri}(x)$ (see next table).

$\theta_{inc_{G0}}(x,y,z)$ is the formula $ninc_{G0}(x,y,z) \vee ninc_{Ri}(x,y,z)$.

$\theta_{t_V^{G0}}(x,t)$ is the formula $nvert_{G0}(x,t) \vee \Big[nvert_{Ri}(x) \wedge t_V^{Ri}(x,t) \Big]$.

$\theta_{t_E^{G0}}(x,t)$ is the formula $nt_E^{G0}(x,t) \vee t_E^{Ri}(x,t)$.

$\theta_{vert_{Li}}$, $\theta_{inc_{Li}}$, $\theta_{t_V^{Li}}$, $\theta_{t_E^{Li}}$, $\theta_{vert_{Ri}}$, $\theta_{inc_{Ri}}$, $\theta_{t_V^{Ri}}$, $\theta_{t_E^{Ri}}$, $\theta_{\alpha_{i_V}}$, $\theta_{\alpha_{i_E}}$ are respectively the formulas $vert_{Li}(x)$, $inc_{Li}(x,y,z)$, $t_V^{Li}(x,y)$, $t_E^{Li}(x,y)$, $vert_{Ri}(x)$, $inc_{Ri}(x,y,z)$, $t_V^{Ri}(x,y)$, $t_E^{Ri}(x,y)$, $\alpha_{i_V}(x,y)$ and $\alpha_{i_E}(x,y)$, for $i = 1 .. n$ (same rules).

Example 6. *The graph grammar that results of the application of rule $|\alpha_1|$ at match $|m1|$ in $|G0|$ ($|GG|$ initial state), has its initial graph defined by the relations (see Figure 2.3):*

$vert_{G0_{|GG|'}}$ = { N01, N02, N03 };
$inc_{G0_{|GG|'}}$ = { (Stb02, N02, N02), (Nxt02, N02, N03), (Stb03, N03, N03), (Nxt03, N03, N01), (Tok12, N01, N01), (Act11, N01, N01), (Nxt12, N01, N02), (Msg11, N02, N02) }
$t^{G0}_{V_{|GG|'}}$ = { (N01, Node), (N02, Node), (N03, Node) };
$t^{G0}_{E_{|GG|'}}$ = { (Stb02, Stb), (Nxt02, Nxt), (Stb03, Stb), (Nxt03, Nxt) , (Tok12, Tok), (Act11, Act), (Nxt12, Nxt), (Msg11, Msg) }.

The elements of these relations are those of $|GG|'$ domain (same domain as $|GG|$) that satisfy the formulas, $\theta_{vert_{G0}}, \theta_{inc_{G0}}, \theta_{t_V^{G0}}, \theta_{t_E^{G0}}$, respectively.

Proposition 9. *The rule application as a FO-definable transduction is well-defined.*

Proof. Let $|GG|'$ be the result of the transduction applied to graph grammar $|GG|$ corresponding to the application of relational rule $|\alpha_i|$ at relational match $|m|$. Considering that the given rule $|\alpha_i| = \{\alpha_{i_V}, \alpha_{i_E}\}$ and the given match $|m| = \{m_V, m_E\}$ are the relational representations of $\alpha_i : Li^T \to Ri^T$ and $m : Li^T \to G0^T$, respectively, and considering $H^T = (H, t^H, T)$ to be the typed graph obtained by the application of α_i to graph $G0^T$ at match m (according to Definition 5) we have to show that[2]:

1. $vert'_T$ and inc'_T are the relations of a $\mathcal{R}_{gr}$-structure $|T|' = \langle V'_T \cup E'_T, \{vert'_T, inc'_T\}\rangle$ representing graph $T = (V_T, E_T, src_T, trg_T)$.

[2]Each relation r of $|GG|'$ will be denoted by r' to avoid confusion with the relations of $|GG|$ (denoted by the unprimed names).

Table 3.1: Formulas used in Definition 16

Notation	Intuitive Meaning	Formula
$vert_G(x)$	x is a vertex of graph G in GG.	$vert_G(x)$
$inc_G(x,y,z)$	x is an edge of graph G with source vertex y and target vertex z in GG.	$inc_G(x,y,z)$
$t_V^G(x,y)$	x is a vertex of graph G of type y in GG.	$t_V^G(x,y)$
$t_E^G(x,y)$	x is an edge of graph G of type y in GG.	$t_E^G(x,y)$
$\alpha_{i_V}(x,y)$	x is a vertex of graph Li mapped to vertex y of Ri by rule α_i in GG.	$\alpha_{i_V}(x,y)$
$\alpha_{i_E}(x,y)$	x is an edge of graph Li mapped to edge y of Ri by rule α_i in GG.	$\alpha_{i_E}(x,y)$
$nvert_{Ri}(x)$	x is a vertex of graph Ri that is not image of the rule α_i in GG.	$vert_{Ri}(x) \wedge \nexists y\Big(\alpha_{i_V}(y,x)\Big)$
$ninc_{G0}(x,y,z)$	x is an edge of graph $G0$ with source y and target z in GG that is not image of the match.	$inc_{G0}(x,y,z) \wedge \nexists w\Big(m_E(w,x)\Big)$
$ninc_{Ri}(x,y,z)$	x is an edge of graph Ri with source and target vertices given by binary relation $\overline{n}$.	$\exists r,s\Big[inc_{Ri}(x,r,s)\wedge\overline{n}(r,y)\wedge\overline{n}(s,z)\Big]$
$\overline{n}(r,y)$	Vertex r is related to some different vertex y if it is image of the rule applied to some vertex v. In this case r is related with the image of the match applied to v. Vertex r is related to itself if it is not image of the rule.	$\begin{cases}\exists v\Big(\alpha_{i_V}(v,r)\wedge m_V(v,y)\Big) & \text{if } r \neq y\\ \nexists v\ \alpha_{i_V}(v,r) & \text{if } r = y\end{cases}$
$nvert_{G0}(x,t)$	x is a vertex of graph $G0$ of type t in GG.	$vert_{G0}(x) \wedge t_V^{G0}(x,t)$
$nt_E^{G0}(x,t)$	x is an edge of graph $G0$ of type t in GG that is not image of the match.	$\exists y,z\Big(inc_{G0}(x,y,z)\Big)\wedge$ $\wedge \nexists w\Big(m_E(w,x)\Big) \wedge t_E^{G0}(x,t)$

- $x \in vert'_T$ *iff* $x \in V_T$: By θ_{vert_T} definition, $x \in vert'_T$ *iff* $x \in vert_T$. Since $vert_T$ is the relation of a $\mathcal{R}_{gr}$-structure representing T, then (following Definition 7) $x \in vert_T$ *iff* $x \in V_T$.
- $(x,y,z) \in inc'_T$ *iff* $x \in E_T \wedge src_T(x) = y \wedge trg_T(x) = z$: By θ_{inc_T} definition, $(x,y,z) \in inc'_T$ *iff* $(x,y,z) \in inc_T$. Since inc_T is the relation of a $\mathcal{R}_{gr}$-structure representing T, then (following Definition 7) $(x,y,z) \in inc_T$ *iff* $x \in E_T \wedge src_T(x) = y \wedge trg_T(x) = z$.

2. $vert'_{G0}$ and inc'_{G0} are the relations of a $\mathcal{R}_{gr}$-structure $|G0|' = \langle V'_{G0} \cup E'_{G0}, \{vert'_{G0}, inc'_{G0}\}\rangle$ representing graph $H = (V_H, E_H, src_H, trg_H)$.

   - $x \in vert'_{G0}$ *iff* $x \in V_H$: By $\theta_{vert_{G0}}$ definition, $x \in vert'_{G0}$ *iff* $x \in vert_{G0}$ or $(x \in vert_{Ri} \wedge \nexists y, (y,x) \in \alpha_{i_V})$.
     – Let $x \in vert_{G0}$. Since $|G0|$ is a relational representation of $G0$, we have

$x \in V_{G0}$. Therefore, by Definition 5, $x \in V_H$.

- Let $x \in vert_{Ri}$ such that $\nexists y, (y, x) \in \alpha_{i_V}$. Since $|Ri|$ is a relational representation of Ri, $x \in V_{Ri}$. Also, as $|\alpha_i|$ is a relational representation of $\alpha_i = (\alpha_{i_{Vert}}, \alpha_{i_{Edge}})$, by Definition 12, $\nexists y, \alpha_{i_{Vert}}(y) = x$. Consequently, $x \in (V_{Ri} - \alpha_{i_{Vert}}(V_{Li}))$, and by Definition 5, $x \in V_H$.

The proof in the other direction (only if case) is analogous.

- $(x, y, z) \in inc'_{G0}$*iff* $x \in E_H \wedge src_H(x) = y \wedge trg_H(x) = z$: By $\theta_{inc_{G0}}$ definition, $(x, y, z) \in inc'_{G0}$*iff* $\Big((x, y, z) \in inc_{G0} \wedge \nexists w, (w, x) \in m_E\Big)$ or $\exists r, s\Big((x, r, s) \in inc_{Ri} \wedge (r, y) \in \overline{n} \wedge (s, z) \in \overline{n}\Big)$.
  - Let $(x, y, z) \in inc_{G0}$, such that $\nexists w, (w, x) \in m_E$. Since $|G0|$ is the relational representation of $G0$, we have $x \in E_{G0} \wedge src_{G0}(x) = y \wedge trg_{G0}(x) = z$. Also, as $|m|$ is a relational representation of $m = (m_{Vert}, m_{Edge})$, by Definition 15, $\nexists w, m_{Edge}(w) = x$. As a result, $x \in E_{G0} - m_{Edge}(E_{Li})$, i.e. by Definition 5, $x \in E_H$. In this case, the source and target vertices of x in H are the same of $G0$, i.e., y and z respectively.
  - Let $(x, r, s) \in inc_{Ri}$, where $\overline{n}(r, y)$ and $\overline{n}(s, z)$ hold. Considering that $|Ri|$ is a relational representation of Ri we have $x \in E_{Ri} \wedge src_{Ri}(x) = r \wedge trg_{Ri}(x) = s$. Consequently, by Definition 5, $x \in E_H$. As $\overline{n}(r, y)$ holds, we have two alternatives:
    * $\exists v, \Big(\alpha_{i_V}(v, r) \wedge m_V(v, y)\Big)$ with $r \neq y$. In this case, since $|\alpha_i|$ is a relational representation of rule α_i and $|m|$ is a relational representation of m, we have $\alpha_{i_{Vert}}(v) = r$ and $m_{Vert}(v) = y$. Hence, by Definition 5, $src_H(x) = \overline{m}(src_{Ri}(x)) = \overline{m}(r) = m_{Vert}(v)\big($ since $r \in rng(\alpha_{i_{Vert}})$ and $r = \alpha_{i_{Vert}}(v)\big) = y$.
    * $\nexists v, \ \alpha_{i_V}(v, r)$ with $r = y$. Thus, $\nexists v$ such that $\alpha_{i_{Vert}}(v) = r$. By Definition 5, in this case, $src_H(x) = \overline{m}(src_{Ri}(x)) = \overline{m}(r) = r = y$.

  Following a similar argument, if $\overline{n}(s, z)$ holds, we can conclude in both alternatives that $trg_H(x) = z$.

The only if proof is similar.

3. $t_V^{G0'}$ and $t_E^{G0'}$ are from the set $|t^{G0}|'$ such that the tuple $\langle |G0|', |t^{G0}|', |T|' \rangle$ is a relational representation of the typed graph $H^T = (H, t^H, T)$.

- $(x, t) \in t_V^{G0'}$ *iff* $t_{Vert}^H(x) = t$: By $\theta_{t_V^{G0}}$ definition, $(x, t) \in t_V^{G0'}$ *iff* $\Big(x \in vert_{G0} \ \wedge \ (x, t) \in t_V^{G0}\Big)$ or $\Big(x \in vert_{Ri} \ \wedge \ \nexists y, \ (y, x) \in \alpha_{i_V} \ \wedge$ $\wedge \ (x, t) \in t_V^{Ri}\Big)$.
  - Let $x \in vert_{G0}$ and $(x, t) \in t_V^{G0}$. Since $\langle |G0|, |t^{G0}|, |T| \rangle$, with $|t^{G0}| = \{t_V^{G0}, t_E^{G0}\}$, is a relational representation of the typed graph $G0^T$, we have $x \in V_{G0}$ and $t_{Vert}^{G0}(x) = t$. Then, by Definition 5 $t_{Vert}^H(x) = t_{Vert}^{G0}(x) = t$.
  - Let $x \in vert_{Ri}$ and $(x, t) \in t_V^{Ri}$, such that $\nexists y, \ (y, x) \in \alpha_{i_V}$. Since $\langle |Ri|, |t^{Ri}|, |T| \rangle$, with $|t^{Ri}| = \{t_V^{Ri}, t_E^{Ri}\}$, is a relational representation of the typed graph Ri^T and $|\alpha_i|$ is a relational representation of α_i, we have $x \in V_{Ri}, t_{Vert}^{Ri}(x) = t$ and $\nexists y, \alpha_{i_{Vert}}(y) = x$. Then, $x \in (V_{Ri} - \alpha_{i_{Vert}}(V_{Li}))$ and by Definition 5, $t_{Vert}^H(x) = t_{Vert}^{Ri}(x) = t$.

The only if proof is similar.

- $(x,t) \in t_E^{G0'}$ *iff* $t_{Edge}^H(x) = t$: By $\theta_{t_E^{G0}}$ definition, $(x,t) \in t_E^{G0'}$ *iff* $\left(\exists y,z,(x,y,z) \in inc_{G0} \ \wedge \ \nexists w,(w,x) \in m_E \wedge \ (x,t) \in t_E^{G0}\right)$ or $(x,t) \in t_E^{Ri}$.
  - Let $(x,t) \in t_E^{G0}$, such that $\exists y,z,(x,y,z) \in inc_{G0}$ and $\nexists w,(w,x) \in m_E$. Since $|G0|$, $|m|$ and $\langle |G0|, |t^{G0}|, |T| \rangle$ are relational representations of $G0$, m and $G0^T$, respectively, we have $x \in E_{G0}$, $\nexists w, m_{Edge}(w) = x$ and $t_{Edge}^{G0}(x) = t$. I.e., $x \in (E_{G0} - m_{Edge}(E_{Li}))$. Then, by Definition 5, $t_{Edge}^H(x) = t_{Edge}^{G0}(x) = t$.
  - Let $(x,t) \in t_E^{Ri}$. Since $\langle |Ri|, |t^{Ri}|, |T| \rangle$, with $|t^{Ri}| = \{t_V^{Ri}, t_E^{Ri}\}$, is a relational representation of the typed graph Ri^T, we have $x \in E_{Ri}$ and $t_{Edge}^{Ri}(x) = t$. Thus, by Definition 5, $t_{Edge}^H(x) = t_{Edge}^{Ri}(x) = t$.

The only if proof is similar.

$\square$

3.3 Verifying Properties

In this section, we lay the foundation for the creation of a graph grammar theory, which may be used to formulate properties and develop proofs. This proposal of formalization was inspired by the standard procedure of Isabelle (NIPKOW; PAULSON; WENZEL, 2002) to the development of proofs: working with Isabelle means creating theories. Nevertheless, the definitions here proposed must guide the analysis of graph grammar systems in any other proof assistant.

The relational definition of a graph grammar establishes a set of axioms to be used in the proof process. That is, given a relational structure $|GG| = \langle D_{GG}, (R)_{R \in \mathcal{R}_{GG}} \rangle$, each relation R of $|GG|$ defines an axiom: $R(x_1, \ldots, x_n) \equiv true$ iff $(x_1, \ldots, x_n) \in R$. The theory defines a data type named reachable graph and a standard library of functions.

Definition 17 (Reachable Graph Data Type)**.** *The* ***data type reachable graph*** *(reach_gr) of a graph grammar is defined with two constructors, one for the initial graph $G0$ and another one for the operator $ap(\alpha i, m)$ that applies the rule αi at match m to a reachable graph.*

$$\textbf{\textit{datatype}}\ gg\ \textit{reach\_gr} = G0$$
$$\mid ap(\alpha i, m)\ \text{“}gg\ \textit{reach\_gr}\text{”}$$

$G0$ is defined by relations $vert_{G0},\ inc_{G0},\ t_V^{G0},\ t_E^{G0}$ of $|GG|$. Relations of the resulting graph of a rule application are defined according to the transduction defined in Section 3.2.

The **standard library** provides a collection of (recursive) functions that can be used to state and prove desirable properties. Properties about reachable states may be proven by induction, since this data type is recursively defined.

For instance, we define two functions: one to determine the types of edges of a reachable graph and another to indicate if a reachable graph has a ring topology. Let $|GG|$ be the relational structure associated to a graph grammar.[3]

[3]Again, in what follows, we omit the subscript GG in relations, assuming that it is clear from context which grammar is under consideration.

In the following we assume a fixed given grammar $GG = (T, G0, R)$.

[Library function tip: Types of Edges of a Reachable Graph] The types of edges of a reachable graph are recursively defined by:

$$tip_E\ G0 = \{(x,t) \mid t_E^{G0}(x,t)\} \tag{3.1a}$$

$$tip_E\ ap(\alpha i, m)\ G = \{(x,t) \mid t_E^{Ri}(x,t) \vee [(x,t) \in tip_E\ G \wedge \nexists w\ m_{E_{\alpha i}}(w,x)]\} \tag{3.1b}$$

That is, if we consider the initial graph (3.1a), typing is given by the relation t_E^{G0} of the relational structure. If we consider a graph obtained from applying rule αi at match $m = \{m_{V_{\alpha i}}, m_{E_{\alpha i}}\}$ to graph G (3.1b), the type of an edge is either the type of edges of the right-hand side of the rule or a type of edge of graph G (in the latter case, the edge can not be image of the match). ■

[Library function $Ring$: Ring Topology in a Reachable Graph] Initially, we define the transitive closure of edges of type t in a graph G, denoted by $TC^t_{inc_G}$, by:

$$\forall a,x,y,z \Big([inc_G(a,x,y) \wedge t_E^G(a,t) \rightarrow (x,y) \in TC^t_{inc_G}] \wedge [(x,y) \in TC^t_{inc_G} \wedge (y,z) \in TC^t_{inc_G} \rightarrow (x,z) \in TC^t_{inc_G}] \Big)$$

Then, the function that indicates if a reachable graph has a ring topology of edges of type t is defined by:

$$\text{Ring}_t\ G0 \equiv \forall x\ [vert_{G0}(x) \rightarrow (x,x) \in TC^t_{inc_{G0}}] \wedge \tag{3.2a}$$

$$\wedge\ \forall a,b,x,y,z\ [inc_{G0}(a,x,y) \wedge t_E^{G0}(a,t) \wedge inc_{G0}(b,x,z) \wedge \wedge\, t_E^{G0}(b,t) \rightarrow a = b] \wedge \tag{3.2b}$$

$$\wedge\ \forall x,z\ [vert_{G0}(x) \wedge vert_{G0}(z) \rightarrow (x,z) \in TC^t_{inc_{G0}}] \tag{3.2c}$$

$$\text{Ring}_t\ ap(\alpha i, m)\ G \equiv \text{Ring}_t\ G \wedge \tag{3.2d}$$

$$\wedge\ \forall a,x,y,z,w\ [inc_{Li}(a,x,y) \wedge t_E^{Li}(a,t) \wedge \alpha_{i_V}(x,z) \wedge \wedge\, \alpha_{i_V}(y,w) \rightarrow (z,w) \in TC^t_{inc_{Ri}}] \wedge \tag{3.2e}$$

$$\wedge\ \forall a,b,x,y,z\ [inc_{Ri}(a,x,y) \wedge t_E^{Ri}(a,t) \wedge inc_{Ri}(b,x,z) \wedge \wedge\, t_E^{Ri}(b,t) \rightarrow a = b] \tag{3.2f}$$

That is, $G0$ has a ring topology if the following conditions are satisfied:

(3.2a) There is a cycle, i.e., every vertex of $G0$ has a path with origin and destination in itself;

(3.2b) There is no bifurcation of edges of type t in $G0$, i.e., if there are two edges of type t with origin at the same vertex, these edges are equal. This property guarantees that the paths of edges of type t in $G0$ are unique;

(3.2c) The graph is connected, i.e., from every vertex in $G0$ there is a path to all other vertices.

And, to have a graph with a ring topology resulting from the application of a rule $\alpha_i = \{\alpha_{i_V}, \alpha_{i_E}\}$ to a reachable graph, it must be guaranteed that:

(3.2d) The reachable graph before applying αi has a ring structure;

(3.2e) For every edge a of type t going from x to y in Li there is a corresponding path in Ri starting at the image α_{i_V} of x and ending at the image α_{i_V} of y;

(3.2f) There is no bifurcation of edges of type t in Ri. This guarantees that the paths of edges of type t in Ri are unique.

■

Other functions could also be included in the library, such as functions to define types of vertices of a reachable graph, cardinality of edges, cardinality of vertices and many others. Having established the theory, we describe the **proof strategy** used to prove properties for a system specified in graph grammar. First, we must define the relational structure associated to the grammar (according to Definition 13). The relations of this structure define axioms that are used in the proofs. Then we may state a goal to be proven using logic formulas. Considering that the property states some desirable characteristic of reachable graphs, the proof must be performed in the following way: first (base case), the property is verified for the initial graph ($G0$) and then, at the inductive step, the property is verified for every rule of the grammar applicable to a reachable graph G (i.e., for $ap(\alpha i, m)\ G$), considering that the property is valid for G. This process may be semi-automated: it may proceed until a separate property or lemma is required, then we must establish the property or prove the lemma, and then the proof of the original goal can continue.

Now, we give two **examples** of proofs of properties for the Token Ring protocol: one about types of edges and another about the structure of reachable graphs.

Property 1. *Any reachable graph has exactly one edge of the type* Tok.

According to the definition of tip_E, previously established in the library, the property to be proven can be stated by the formula:

$$\exists! x\ [(x, \mathsf{Tok}) \in \mathrm{tip_E\ reach\_gr}].$$

Proof.

Basis: Here, the property is verified for the initial graph $G0$.

$$\exists! x\ [(x, \mathsf{Tok}) \in \mathrm{tip_E\ G0}] \overset{(3.1a)}{\equiv} \exists!\mathrm{x}\ [\mathrm{t_E^{G0}}(\mathrm{x}, \mathsf{Tok})] \equiv \mathrm{true}.$$

The last equivalences may be verified automatically. Since the relational structure that defines the grammar has a single pair with the second component Tok belonging to the relation t_E^{G0} (see Example 2), the logical expression must be evaluated to $true$.

Hypothesis: For any reachable graph G $\exists! x[(x, \mathsf{Tok}) \in \mathrm{tip_E G}]$

Inductive Step: Assuming the hypothesis, the proof reduces to five cases, depending on the rule that is applicable:

Rule $\alpha 1$: $\exists! x\ [(x, \mathsf{Tok}) \in \mathrm{tip_E\ ap}(\alpha 1, \mathrm{m})\ \mathrm{G})] \overset{(3.1b)}{\equiv}$

$\exists! x\ [t_E^{R1}(x, \mathsf{Tok}) \vee [(\mathrm{x}, \mathsf{Tok}) \in \mathrm{tip_E\ G} \wedge \nexists \mathrm{w\ m_{E_{\alpha 1}}(w, x)}]].$

Now it is necessary to inform if the edge x of type Tok of the reachable graph is an image of the match or not, when rule αi is applied. This can be done by stating:

$$\forall x\ (x, \mathsf{Tok}) \in \mathrm{tip_E\ G},\ \ \exists \mathrm{w\ m_{E_{\alpha i}}(w, x)} \Leftrightarrow \exists \mathrm{w\ t_E^{Li}}(\mathrm{w}, \mathsf{Tok}) \tag{3.3}$$

According to (3.3), the edge of type Tok of the reachable graph will be an image of the match if and only if the left-hand side of the applied rule contains an edge of the type Tok. Then:

$$\exists ! x\ [t_E^{R1}(x, \mathsf{Tok}) \vee [(\mathrm{x}, \mathsf{Tok}) \in \mathrm{tip_E\ G} \wedge \nexists \mathrm{w\ m_{E_{\alpha 1}}(w, x)}]] \overset{(3.3)}{\equiv}$$
$$\exists ! x\ [t_E^{R1}(x, \mathsf{Tok}) \vee [(\mathrm{x}, \mathsf{Tok}) \in \mathrm{tip_E\ G} \wedge \nexists \mathrm{w\ t_{L1_E}}(\mathrm{w}, \mathsf{Tok})]] \equiv \mathrm{true}.$$

There is a (single) pair at the relation t_E^{R1} that has the second component Tok (see Example 3). Besides it is assumed by hypothesis that $(x, \mathsf{Tok}) \in \mathrm{tip_E}$ G. Since expression $\nexists w\ m_{E_{\alpha 1}}(w, x)$ is evaluated to false (there is a pair in relation t_E^{L1} that has the second component Tok), the complete formula may be automatically evaluated to true.

Rules $\alpha 2$ **to** $\alpha 5$**:** The proofs for rules $\alpha 2$, $\alpha 3$, $\alpha 4$ and $\alpha 5$ are analogous. It is important to notice that, since the property that informs if an edge of type Tok is the image of a match has already been stated, the verification for these rules may proceed automatically.

$\square$

Property 2. *Any reachable graph has a ring topology of edges of type* Nxt.

Considering that the transitive closure of edges and the function that identifies a ring topology are previously defined in the library, the property to be proven can be stated as:

$$\mathrm{Ring}_{\mathsf{Nxt}}\ reach\_gr \equiv true.$$

Proof.

Basis: We instantiate the equations (3.2a), (3.2b) and (3.2c) of the $Ring$ definition with $G0$ and Nxt

$$\mathrm{Ring}_{\mathsf{Nxt}}\ G0 \overset{\text{def.}}{\equiv} \forall x\ [vert_{G0}(x) \rightarrow (x, x) \in TC^{\mathsf{Nxt}}_{inc_{G0}}]\ \wedge \tag{eqn 3.2a}$$
$$\wedge \forall a, b, x, y, z\ [inc_{G0}(a, x, y) \wedge t_E^{G0}(a, \mathsf{Nxt}) \wedge \mathrm{inc_{G0}(b, x, z)} \wedge$$
$$\wedge t_E^{G0}(b, \mathsf{Nxt}) \rightarrow \mathrm{a = b}]\ \wedge \tag{eqn 3.2b}$$
$$\wedge \forall x, z\ [vert_{G0}(x) \wedge vert_{G0}(z) \rightarrow (x, z) \in TC^{\mathsf{Nxt}}_{inc_{G0}}] \equiv \tag{eqn 3.2c}$$
$$\equiv true$$

Considering that the result of the operation $TC^{\mathsf{Nxt}}_{inc_{G0}}$ is the set $\{(\mathsf{N01}, \mathsf{N02}), (\mathsf{N02}, \mathsf{N03}), (\mathsf{N03}, \mathsf{N01}), (\mathsf{N01}, \mathsf{N03}), (\mathsf{N02}, \mathsf{N01}), (\mathsf{N03}, \mathsf{N02}), (\mathsf{N01}, \mathsf{N01}), (\mathsf{N02}, \mathsf{N02}), (\mathsf{N03}, \mathsf{N03})\}$, (eqn 3.2a) and (eqn 3.2c) are satisfied. (eqn 3.2b) is also satisfied because there are no two edges of the type Nxt in $G0$ starting at the same vertex (see Examples 1 and 2).

Hypothesis: For any reachable graph G $\mathrm{Ring}_{\mathsf{Nxt}}\ G \equiv true \Rightarrow$

Inductive Step: Again, here we have to prove for all rules $\alpha 1$ to $\alpha 5$. We show the proof for the first rule, the others are analogous.

$$\mathrm{Ring}_{\mathsf{Nxt}}\ ap(\alpha 1, m)\ G \overset{\text{def.}}{\equiv} \mathrm{Ring}_{\mathsf{Nxt}}\ G\ \wedge \tag{eqn 3.2d}$$

$$\wedge\forall a, x, y, z, w\ [inc_{L1}(a, x, y) \wedge t_E^{L1}(a, \mathsf{Nxt}) \wedge \alpha_{1_\mathrm{V}}(\mathrm{x}, \mathrm{z}) \wedge$$

$$\wedge\alpha_{1_V}(y, w) \rightarrow (z, w) \in TC^{\mathsf{Nxt}}_{inc_{R1}}]\ \wedge \tag{eqn 3.2e}$$

$$\wedge\forall a, b, x, y, z\ [inc_{R1}(a, x, y) \wedge t_E^{R1}(a, \mathsf{Nxt}) \wedge \mathrm{inc_{R1}}(\mathrm{b}, \mathrm{x}, \mathrm{z})\wedge$$

$$\wedge t_E^{R1}(b, \mathsf{Nxt}) \rightarrow \mathrm{a} = \mathrm{b}] \equiv \tag{eqn 3.2f}$$

$$\equiv true$$

This property may be verified automatically: (eqn 3.2d) is valid by the induction hypothesis; (eqn 3.2e) is valid by the result of the operation $TC^{\mathsf{Nxt}}_{inc_{R1}}$; and (eqn 3.2f) is valid because there are no two edges of type Nxt starting at the same node in $R1$ (see Example 3). □

4 DEALING WITH ATTRIBUTED GRAPHS

An attributed graph has two components: a graphical part (a graph) and a data part. These components are linked by attribution functions or edges (depending on the approach). The data part allows the use of variables and terms in the rules (as attributes), giving the specifier a better level of abstraction with respect to grammars using only non-attributed graphs. Figure 4.1 shows an example of rule application using attributed graphs. In rule $r : L \rightarrow R$, instead of using concrete values, one typically uses variables and terms. Equations restrict the situations in which the rule may be applied. To be able to apply such a rule, we must find, besides the graph homomorphism from L to G, an assignment of values to the variables of the rule that satisfy all equations. If such an assignment is found (like asg in the figure), the rule can be applied and the resulting graph H is obtained as previously defined (as in Def. 5) with the values of attributes of the vertices changing as defined in the rules.

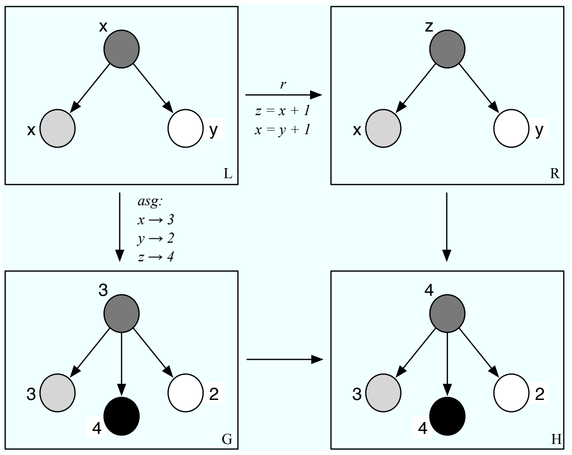


Figure 4.1: Rule Application using Attributed Graphs

In (EHRIG et al., 2006), an attributed graph is a graph in which some vertices are actually data values, and some edges are attribution edges, that is, all data values are considered as vertices and there are special edges connecting graphical vertices to these data vertices. This approach has a very nice theory but, for (automated) verification purposes, it is not directly useful because typically data types involve infinite sets of values, and thus each graph will be an infinite structure (because data values are vertices).

A different approach was presented by GROOVE in (KASTENBERG, 2006), in which the data values were modeled as term graphs. In this approach, rewriting takes place at two different levels: normal graph rewriting for the graphical part and term graph rewriting for the data part of the attributed graph. However, modeling data types as terms has some disadvantages: many data types can be more naturally expressed as “textual" terms; resolution for many of the most used equational systems (like natural numbers, booleans, strings, lists, ...) is already efficiently implemented, whereas there are some limitations for term graph rewriting. Moreover, this technique presented for GROOVE is for finite state graph transformation systems.

A new approach to perform verification of attributed GTS was presented in (KÖNIG; KOZIOURA, 2008). This approach is based on (LöWE; KORFF; WAGNER, 1993), in which there is an attribution function mapping elements from the graphical part to the data part of the graph. Here the data part is not seen as vertices or edges of an attributed part, but rather as a set of values. The disadvantage is that it is not possible to change the value of the attribute of a vertex without deleting this vertex (because a simple change of attribute would not be compatible with the original attribution function, and this compatibility is a requirement for the definition of morphisms). In (KÖNIG; KOZIOURA, 2008) this drawback does not play a role since only edges are attributed, and all edges belonging to the left-hand side of a rule must be deleted.

Our approach is inspired by both (EHRIG et al., 2006) and (KÖNIG; KOZIOURA, 2008). On the one hand, we will have some special kind of edges of the graph that will be called *attribute edges* or simply *attributes* and will be used actually to describe attribution of vertices. But on the other hand, we will have a function assigning a data value to each of these attribute edges. This way, we can model that the attribute $a1$ of a vertex changes (by deleting the attribute edge corresponding to $a1$ and creating a new one with the new value) in a framework in which graphs are not infinite (because data values must not be part of the graph).

For example, the left-hand side of the attributed rule shown before would be actually described by the graph depicted in Figure 4.2: dashed loop edges are placed onto the vertices that will get attributed, and the attribute values are actually connected to these edges.

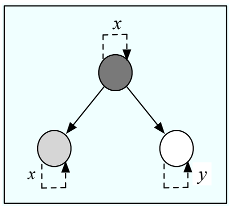


Figure 4.2: Attributed Graph

4.1 Attributed Graph Grammars

We use algebraic specifications to define data types, and algebras to describe the values that can be used as attributes. Appendix A provides basic definitions of algebraic specifications (these concepts will also be informally introduced as necessary).

A *signature* $SIG = (S, OP)$ consists of a set S of sorts and a set OP of constant and operations symbols. Given a set of variables X (of sorts in S), the *set of terms* over SIG is denoted by $T_{OP}(X)$ (this is defined inductively by stating that all variables and constants are terms, and then all possible applications of operation symbols in OP to existing terms are also terms). An *equation* is a pair of terms $(t1, t2)$, and is usually denoted by $t1 = t2$. A *specification* is a pair $SPEC = (SIG, Eqns)$ consisting of a signature and a set of equations over this signature. An *algebra* for specification $SPEC$, or $SPEC$-algebra, consists of one set for each sort symbol of SIG, called *carrier set*, and one function for each operation symbol of SIG such that all equations in $Eqns$ are satisfied (satisfaction of one equation is checked by substituting all variables in the equation by values of corresponding carrier sets and verifying whether the equality holds, for all possible substitutions). Given two $SPEC$-algebras, a homomorphism between them is a set of functions mapping corresponding carrier sets that are compatible with all functions of the algebras. The set obtained by the disjoint union of all carrier sets of algebra A is denoted by $\mathcal{U}(A)$.

In the following, let $loop(G)$ denote the subset of edges of a graph that are loops, that is, edges that have the same source and target vertices. In a graph, some of its loop edges will be considered as special edges: they will be used to connect a vertex to an attribute value.

Definition 18 (Attributed Graph)**.** *Given a specification $SPEC$, an **attributed graph** is a tuple $AG = (G, A, attr_G)$ where $G = (V_G, E_G, src_G, trg_G)$ is a graph, A is a $SPEC$-algebra, and*

$$attr_G : AttrE_G \to \mathcal{U}(A)$$

*is a total function, with $AttrE_G \subseteq loop(G)$. Edges belonging to $AttrE_G$ are called **attribute edges**.*

*A **(partial) attributed graph morphism** g between attributed graphs AG and AH is a pair $g = (g_{Graph}, g_{Alg})$ consisting of a graph morphism $g_{Graph} = (g_{Vert}, g_{Edge})$ and an algebra homomorphism g_{Alg} between the corresponding components that are compatible with the attribution, i.e.*

$$\forall e \in AttrE_G\ [g_{Alg}(attr_G(e)) = attr_H(g_{Edge}(e))]$$

An attributed graph morphism g is called total/ injective if all components are total/ injective, respectively.

The role of the type graph is to define the types of vertices and edges of instance graphs. It is thus adequate that the part of the type graph describing data elements consists of names of types. Therefore, we require that the algebra of the type graph is a final one, that is, an algebra in which all carrier sets are singletons. In practice, we will use the name of the corresponding sort as the only element in a carrier set interpreting it. With respect to the attribute edges, there may be many different kinds of attribute edges for the same vertex, and this is described by the existence of many of such edges in the type graph. The two requirements that we impose on a typed attributed graph are (i) *attribute uniqueness*: there may be at most one attributed edge of each kind connected to the same vertex (that is, at most one value for this attribute is associated to each vertex), and (ii) *attribute completeness*: in an attributed graph, all attributes of each vertex must be defined (that is, once an attribute edge exists in the type graph, there must be a corresponding value in any instance graph). These requirements make sense in practice, since when a list of attributes is defined for a vertex, typically one wants that all vertices of each graph will have values for those attributes (completeness), and these values are unique (uniqueness).

For example, Figure 4.3 shows a type graph T in which we can see three types of attributes, two natural numbers and one boolean. Graph G is typed over T (the morphism is given by the dashed arrows). To have a cleaner graphical representation, we will draw a typed attributed graph as shown in Figure 4.4. Here we named the attribute edges to make clear which is which in an instance graph. The morphism on the algebra component is not shown, but it is obvious: The algebra of T will have as carrier sets $T_{Nat} = \{Nat\}$ and $T_{Bool} = \{Bool\}$, and the algebra for G will have $G_{Nat} = \{0, 1, 2, 3, 4, 5, \ldots\}$ and $G_{Bool} = \{true, false\}$. In this case, there is only one possible way to map between the algebras of G and T, that is to map all natural numbers to the element Nat and $true$ and $false$ to $Bool$.

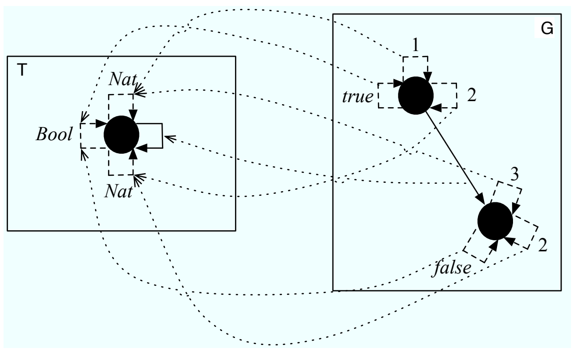


Figure 4.3: Typed Attributed Graph

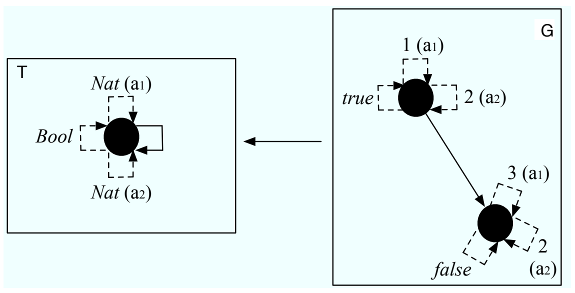


Figure 4.4: Typed Attributed Graph Graphical Notation

Definition 19 (Attributed Type Graph, Typed Attributed Graphs)**.** *Given a specification $SPEC$, an **attributed type graph** is an attributed graph $AT = (T, A, attr_T)$ in which all carrier sets of A are singletons.*

*A **typed attributed graph** is a tuple $AG^{AT} = (AG, t^{AG}, AT)$, where AG is an attributed graph, AT is an attributed type graph and $t^{AG} : AG \to AT$ is a total attributed graph morphism called **attributed typing morphism** such that*

- ***Attribute Uniqueness Condition.*** $\forall e1, e2 \in AttrE_G$ $[src_G(e1) = src_G(e2) \Rightarrow t^{AG}_{Edge}(e1) \neq t^{AG}_{Edge}(e2)]$
- ***Attribute Completeness Condition.*** $\forall e \in AttrE_T$ $[\exists e' \in AttrE_G[t^{AG}_{Edge}(e') = e]]$

We denote by $attrV$ the partial function that associates values to the vertices of an attributed graph. This function is defined by $attrV : V_G \times AttrE_T \to \mathcal{U}(A)$, for all $(v, at) \in V_G \times AttrE_T$

$$attrV(v, at) = \begin{cases} attr_G(e) & \textit{if } \exists e \in AttrE_G\ [src_G(e) = v \wedge attr_T(t^{AG}_{Edge}(e)) = at] \\ undefined & \textit{otherwise} \end{cases}$$

*A **typed attributed graph morphism** between graphs AG^{AT} and AH^{AT} with attributed type graph AT is an attributed graph morphism g between AG and AH such that $t^{AG} \geq t^{AH} \circ g$ (that is, g may only map between elements of the same type).*

Note: The function $attrV$ is well-defined because if there is an attribute edge at some vertex, it will be the only one of its kind (due to the restriction imposed on typed attributed graphs).

Since in the following we will be dealing only with typed attributed graphs, we will omit the word “typed".

Rules specify patterns of behaviour of a system. Therefore, it is natural that variables and expressions (terms) are used for the data part of the graph. We will restrict possible attributes in left- and right-hand sides to be variables, and the possible relations between these variables will be expressed by equations associated to each rule. When applying a rule, all its equations will be required to be satisfied by the chosen assignment of values to variables. The following definition is a slight modification of the usual descriptions of rules using attributed graphs. Usually, a quotient term algebra satisfying all equations of the specification plus the rule equations is used as attribute algebra. This gives rise to a simple and elegant definition. However, since here our aim is to find a finite representation of attributed graph grammars in terms of relational structures, this standard definition is not suitable (in a quotient term algebra, each element of a carrier set is an equivalence class of terms, and this set is typically infinite for many useful data types). Therefore, we just use terms as attributes, that is, we use the term algebra over the signature of the specification as attribute algebra (in the definition below, we equivalently use the term algebra over a specification without equations). In such an algebra, each carrier set consists of all terms that can be constructed using the operations defined for the corresponding sort, functions just represent the syntactical construction of terms (for example for a term t and algebra operation op^A corresponding to an operator op in the signature, we would have $op^A(t) = op(t)$). Consequently, all terms are considered to represent different values in a term algebra, since they are syntactically different. The satisfaction of the equations will be dealt with in the match construction, that is, in the application of a rule.

Definition 20 (Attributed Rule). *Given a specification $SPEC = (SIG, Eqns)$. A rule over $SPEC$ with type AT is a tuple $attRule = (r, X, ruleEqns)$ where*

- *X is a set of variables over the sorts of $SPEC$;*
- *$r : (L, T_{OP}(X), attr_L)^{AT} \to (R, T_{OP}(X), attr_R)^{AT}$ is an injective attributed graph morphism over the specification $(SIG, \varnothing)$ in which $r_{Vert} : V_L \to V_R$ is a total function on the set of vertices, the algebra component is the identity on the term algebra $T_{OP}(X)$, and all attributes used in the left- and right hand sides are variables, i. e. $\bigcup_{e \in AttrE_L} attr_L(e) \cup \bigcup_{e \in AttrE_R} attr_R(e) \subseteq X$.*
- *$ruleEqns$ is a set of equations using terms of $T_{OP}(X)$ such that*
  - *in all equations $t1 = t2 \in ruleEqns$, $t1 \in X$ and $t2$ involves only variables that are attributes of L;*

– *all variables x used in R are either in L or there is an equation $x = t2$ in $ruleEqns$.*

An attributed graph grammar is composed of an *attributed type graph*, an *initial graph* and a *set of rules*.

Definition 21 (Attributed Graph Grammar)**.** *Given a specification $SPEC$ and a $SPEC$-algebra A, a **(typed) attributed graph grammar** is a tuple $AGG = (AT, AG0, R)$, such that AT (the type of the grammar) is an attributed type graph over $SPEC$, $AG0$ (the initial graph of the grammar) is an attributed graph typed over AT using algebra A, and R is a set of rules over $SPEC$ with type AT .*

To define a match, we have to relate, additionally to the graph morphism, the variables of the left-hand side of the rule to the actual values of attributes in the graph in which the rule shall be applied. Additionally, the match construction must assure that all equations of the specification and the rule equations are satisfied by the chosen assignment of variables to values. This will be achieved by first, lifting the rule to a corresponding one having a quotient term algebra as attribute algebra. This is a standard construction in algebraic specification. Then, the actual match will include an algebra homomorphism from this quotient term algebra to the actual algebra used in the graph to which the rule is being applied. The existence of this homomorphism guarantees that all necessary equations are satisfied.

Definition 22 (Attributed Match)**.** *Let a specification $SPEC = (SIG, Eqns)$, a rule over $SPEC$ $attRule = (r, X, ruleEqns)$, $r : AL^{AT} \to AR^{AT}$, with $AL = (L, T_{OP}(X), attr_L)$, and a $SPEC$ attributed graph AG^{AT} be given. An **attributed match** $m : \overline{AL^{AT}} \to AG^{AT}$ is a total attributed graph morphism $m = (m_{Graph}, m_{Alg})$ such that m_{Edge} is injective, $\overline{AL^{AT}} = (L, T_{eq}(X), \overline{attr_L})$, where $T_{eq}(X)$ is the algebra obtained by constructing the quotient term algebra of the specification $(SIG, Eqns \cup ruleEqns)$ using the set of variables X, and, for all term $t \in T_{OP}(X)$, $\overline{attr_L}(t) = [attr_L(t)]$.*

Practically, given a set of variables X and an algebra A, if we define an evaluation function $eval : X \to \mathcal{U}(A)$, there is a unique way to construct the algebra homomorphism (in case it exists for this assignment). First, we check whether all equations in $Eqns \cup ruleEqns$ are satisfied by this assignment. If not, this assignment of values to variables can not lead to an algebra homomorphism, and thus no match can exist using this $eval$ function. Otherwise, we build the extension of $eval$ to (equivalence classes of) terms, that will be denoted by $\overline{eval} : T_{eq}(X) \to \mathcal{U}(A)$. This is the homomorphism we are looking for.

Definition 23 (Rule Application)**.** *Given a specification $SPEC$, a rule over $SPEC$ with type AT $attRule = (r, X, ruleEqns)$ with $r : (L, T_{OP}(X), attr_L)^{AT} \to (R, T_{OP}(X), attr_R)^{AT}$, and a match $m : (L, T_{eq}(X), \overline{attr_L})^{AT} \to (G, A_G, attr_G)^{AT}$ the **application** of rule $attRule$ at match m results in the typed attributed graph AH^{AT}, with $AH = (H, A_H, attr_H)$, where*

- *H is the resulting graph of applying rule $L \to R$ to graph G (as in Def. 5);*
- *$A_H = A_G$;*

- $\forall e \in AttrE_H$

$$attr_H(e) = \begin{cases} attr_G(e) & \text{if } e \in E_G - m_{Edge}(E_L) \\ m_{Alg}(\overline{attr_R}(e)) & \text{if } e \in E_R \end{cases}$$

- *the typing morphism t^{AH} is defined as in Def. 5 for vertices and edges, and t^{AH}_{Alg} is defined as follows:*

$$\forall a \in rng(attr_H),\ t^{AH}_{Alg}(a) = \begin{cases} t^{AG}_{Alg}(a) & \text{if } attr_G(e) = a \wedge e \in E_G - m_{Edge}(E_L) \\ t^{AR}_{Alg}(w) & \text{if } w \in rng(attr_R) \wedge m_{Alg}(w) = a \end{cases}$$

Proposition 10. *Rule application is well-defined (i.e. graph AH is actually an attributed graph).*

Proof. Following Definition 5, H is a well-defined graph and by Definition 18, $A_H = A_G$ is a SPEC-algebra. Since $attr_G$, $attr_R$ and m_{Alg} define total functions, $attr_H$ is defined for all loop edges of H, i.e., $attr_H$ is a total function between $AttrE_H$ and $\mathcal{U}(A)$. Then, $AH = (H, A_H, attr_H)$ is a well-defined attributed graph. The attribute completeness condition is satisfied because, each vertex of H must be either in G or in R (or in both): in any case, since R and G are attributed graphs, all attributes of this vertex must be present and will be copied to H by construction (Def. 5). Attribute uniqueness is due to the fact that L, R and G have at most one attribute of each kind and that the match is total: in this case, either this value of this attribute in the resulting graph H will be given by G (if r preserves this attribute) or by R (if r changes the value of this attribute). Moreover, since the algebra component of the typing morphism is the identity and the other components are compatible with typing (due to Def. 5), AH^{AT} is a well-defined typed attributed graph. □

4.2 Relational Structures Representing Attributed Graph Grammars

In this section we describe the representation of attributed graph grammars by relational structures. The following definitions are proposed assuming a fixed specification SPEC and a fixed algebra A over SPEC. The relational structure representing an attributed graph is essentially Def. 7, including data values in the domain and adding one relation to represent the attribution. Note that only the used data values were included in the domain, not the whole algebra.

Definition 24 (Relational Structure Representing an Attributed Graph)**.** *Let $\mathcal{R}_{agr} = \{vert, inc, attr\}$ be a set of relations, where $vert$ is unary, inc is ternary and $attr$ is binary. Given an attributed graph $AG = (G, A, attr_G)$, a **relational structure representing** AG is a $\mathcal{R}_{agr}$-structure $|AG| = \langle D_{AG}, (R_{AG})_{R \in \mathcal{R}_{agr}} \rangle$, where:*

- $D_{AG} = V_G \cup E_G \cup rng(attr_G)$
- *$vert_{AG}$ and inc_{AG} are the relations defined in Def. 7 (relational structure representing a graph);*
- *$attr_{AG} \subseteq E_G \times rng(attr_G)$ with $(e, a) \in attr_{AG} \iff attr_G(e) = a$*

Proposition 11. *The relational structure $|AG|$ is well-defined.*

Proof. By Proposition 1, $\langle V_G \cup E_G, vert_{AG}, inc_{AG}\rangle$ is a well-defined graph. Since the binary relation $attr_{AG}$ is defined according to $attr_G$, it specifies a value for each attribute edge. Then, $|AG|$ is well-defined. □

The definition of relational morphisms between attributed graphs only adds a relationship between the data values, and requires basically the same conditions as in Def. 8.

Definition 25 (Relational Attributed Graph Morphism)**.** *Let* $|AG| = \langle V_G \cup E_G \cup rng(attr_G), \{vert_{AG}, inc_{AG}, attr_{AG}\}\rangle$ *and* $|AH| = \langle V_H \cup E_H \cup rng(attr_H), \{vert_{AH}, inc_{AH}, attr_{AH}\}\rangle$ *be* $\mathcal{R}_{agr}$*-structures representing attributed graphs. A* ***relational attributed graph morphism*** g ***from*** $|AG|$ ***to*** $|AH|$ *is defined by a set* $g = \{g_V, g_E, g_A\}$ *of binary relations where:*

- g_V *and* g_E *form a relational representation of a graph morphism between the underlying graphs (Def. 8);*
- $g_A \subseteq rng(attr_G) \times rng(attr_H)$ *is a partial function that relates attributes of* $|AG|$ *to attributes of* $|AH|$

such that the following conditions are satisfied:

- ***Attribute Consistency Condition.*** $\forall a, a'$, $[g_A(a, a')] \Rightarrow \exists e, e'[attr_{AG}(e, a) \wedge attr_{AH}(e', a')\ ]$*;*
- ***Attributed Morphism Commutativity Condition.*** $\forall e, a, e', a'$, $[g_A(a, a') \wedge attr_{AG}(e, a) \wedge attr_{AH}(e', a') \Rightarrow\ g_E(e, e')]$

g *is called total/injective if relations* g_V*,* g_E *and* g_A *are total/injective functions, respectively.*

Proposition 12. *A relational attributed graph morphism* $g = \{g_V, g_E, g_A\}$ *from* $|AG|$ *to* $|AH|$ *is well-defined.*

Proof. By Proposition 2, $\{g_V, g_E\}$ is a well-defined graph morphism. g_A is a partial function that, according to the attribute consistency condition, relates attributes of $|AG|$ to attributes of $|AH|$. Moreover, due to the attributed morphism commutativity condition, the relations established by g_A must be compatible with the relations established by g_E. □

The relational representation of typed attributed graphs replaces the relational representations of typed graphs and graph morphism of Def. 10 by relational representation of attributed typed graphs and attributed graph morphism, respectively.

Definition 26 (Relational Representation of a Typed Attributed Graph)**.** *Given a typed attributed graph* $AG^{AT} = (AG, t^{AG}, AT)$ *with* $t^{AG} = (t^{AG}_{Vert}, t^{AG}_{Edge}, t^{AG}_{Alg})$*,* ***a relational representation of*** AG^{AT} *is given by a tuple* $|AG^{AT}| = \langle |AG|, |t^{AG}|, |AT|\rangle$ *where:*

- $|AG|$ *and* $|AT|$ *are* $\mathcal{R}_{agr}$*-structures representing* AG *and* AT*, respectively;*
- $|t^{AG}| = \{t^{AG}_{Vert}, t^{AG}_{Edge}, t^{AG}_{A}\}$ *is a total relational attributed graph morphism from* $|AG|$ *over* $|AT|$*, with* t^{AG}_{A} *corresponding to* t^{AG}_{Alg} *restricted to the elements that are in* $rng(attr_G)$ *and* $rng(attr_T)$*;*

Proposition 13. *The relational representation of a typed attributed graph is well-defined.*

Proof. By Proposition 11, the relational representation of attributed graphs is well-defined and by Proposition 12, the relational representation of an attributed graph morphism is well-defined. Besides, $|AG|$ defines the same set of edges of AG (by Def. 22) and the relational attributed graph morphism between the relational attributed graphs represents the same morphism given. Then, the attribute uniqueness and completeness conditions are still valid. □

The definition of relational morphisms between attributed graphs basically extends the (typed morphism) compatibility condition of Def. 11 with the relationship between data values included in the graph morphism.

Definition 27 (Relational (Typed) Attributed Graph Morphism). *Let $|AG|$, $|AH|$ and $|AT|$ be $\mathcal{R}_{agr}$-structures representing attributed graphs, where $|AT|$ is the relational representation of an attributed type graph, and let $|t^{AG}| = \{t_V^{AG}, t_E^{AG}, t_A^{AG}\}$ and $|t^{AH}| = \{t_V^{AH}, t_E^{AH}, t_A^{AH}\}$ be total relational attributed graph morphisms from $|AG|$ and $|AH|$ to $|AT|$, respectively. A* ***relational attributed (typed) graph morphism from*** *$|AG^T|$* ***to*** *$|AH^T|$ is defined by a relational attributed graph morphism $|g| = \{g_V, g_E, g_A\}$ from $|AG|$ to $|AH|$, such that the attributed typed morphism compatibility condition is satisfied:*

- ***(Attributed Typed Morphism) Compatibility Condition.*** $\forall x, x', y,$
  $[g_V(x, x') \wedge t_V^{AG}(x, y) \Rightarrow t_V^{AH}(x', y)]$;
  $[g_E(x, x') \wedge t_E^{AG}(x, y) \Rightarrow t_E^{AH}(x', y)]$; *and*
  $[g_A(x, x') \wedge t_A^{AG}(x, y) \Rightarrow t_A^{AH}(x', y)]$.

Proposition 14. *The relational representation of a typed attributed graph morphism is well-defined.*

Proof. Following Proposition 12, a relational graph morphism is well-defined. The (typed morphism) compatibility condition guarantees that the relational attributed typed graph morphism only maps elements of the same type. □

The relational representation of an attributed rule is given by a relational typed attributed graph morphism between typed attributed graphs together with two relations: a unary relation to represent the set of variables over $SPEC$ and a binary relation to model the set of equations.

Definition 28 (Relational Representation of an Attributed Rule). *Given a rule $attRule = (r, X, ruleEqns)$ over $SPEC$ with type AT, such that $r = ((r_{Vert}, r_{Edge}), r_{Alg}))$, $r : AL^{AT} \to AR^{AT}$, with $AL = (L, T_{OP}(X), attr_L)$ and $AR = (R, T_{OP}(X), attr_R)$,* ***a relational representation of*** *$attRule$ is given by a tuple $|attRule| = \langle |AL^{AT}|, |r|, |AR^{AT}|, var, |ruleEqns| \rangle$ where:*

- *$|AL^{AT}|$ and $|AR^{AT}|$ are relational representations of typed attributed graphs AL^{AT} and AR^{AT}, respectively;*

- *$|r| = \{r_{Vert}, r_{Edge}, r_A\}$ is a relational typed attributed graph morphism from $|AL^{AT}|$ to $|AR^{AT}|$, where r_A corresponds to r_{Alg} restricted to the elements that are in $rng(attr_L)$ and $rng(attr_R)$;*

- *$var \subseteq X$, with $x \in var \iff x \in X$;*

- $|ruleEqns| \subseteq T_{OP}(X) \times T_{OP}(X)$, *with* $(t1,t2) \in |ruleEqns| \iff (t1,t2) \in ruleEqns$.

Proposition 15. *A relational representation of an attributed rule is well-defined.*

Proof. According to Proposition 13 the relational representation of typed attributed graphs is well defined and according to Proposition 14 the relational representation of a typed attributed graph morphism is well-defined. The definition of the relational typed attributed graph morphism guarantees that it represents the morphism given. Then, the morphism is injective and the component that relates vertices is total. Besides, var is a set of variables over the sorts of $SPEC$ and $|ruleEqns|$ defines the same set $ruleEqns$ (and thus, satisfies the same conditions as $ruleEqns$). □

The definition of relational structure associated to an attributed graph grammar is analogous to the case without attributes, we just have to add the components that correspond to the values of attributes and map these attributes. Remind that we assume a fixed specification $SPEC$ and an algebra A over $SPEC$.

Definition 29 (Relational Structure Associated to an Attributed Graph Grammar)**.** *Let* $\mathcal{R}_{AGG} = \{vert_{AT},\ inc_{AT}, attr_{AT},\ vert_{AG0},\ inc_{AG0}, attr_{AG0},\ t_V^{AG0},\ t_E^{AG0}, t_A^{AG0}, (vert_{ALi},\ inc_{ALi}, attr_{ALi},\ t_V^{ALi},\ t_E^{ALi}, t_A^{ALi}, vert_{ARi}, inc_{ARi}, attr_{ARi},\ t_V^{ARi},\ t_E^{ARi},\ t_A^{ARi},\ r_{i_V},\ r_{i_E},\ r_{i_A},\ var_i, |ruleEqns|_i)_{i\in\{1,\dots,n\}}\}$ *be a set of relation symbols. Given a specification* $SPEC$*, a corresponding algebra* A*, and an attributed graph grammar* $AGG = (AT, AG0, R)$ *over* $SPEC$ *and* A*, where* R *has cardinality* n*,* ***the*** $\mathcal{R}_{\mathbf{AGG}}$***-structure associated to*** AGG*, denoted by* $|AGG|$*, is the tuple* $\langle D_{AGG}, (r)_{r\in\mathcal{R}_{AGG}}\rangle$ *where*

- $D_{AGG} = V_{AGG} \cup E_{AGG} \cup A_{AGG}$ *is the set of vertices, edges and attribute values of the graph grammar, where:* $V_{AGG} \cap E_{AGG} \cap A_{AGG} = \varnothing$, $V_{AGG} = V_T \cup V_{G0} \cup (V_{Li} \cup V_{Ri})_{i\in\{1,\dots,n\}}$, $E_{AGG} = E_T \cup E_{G0} \cup (E_{Li} \cup E_{Ri})_{i\in\{1,\dots,n\}}$ *and* $A_{AGG} = rng(attr_T) \cup rng(attr_{G0}) \cup (rng(attr_{Li}) \cup rng(attr_{Ri}))_{i\in\{1,\dots,n\}}$

- $vert_{AT}$ *,* inc_{AT} *and* $attr_{AT}$ *model the* ***attributed type graph****.*

- $vert_{AG0}$*,* inc_{AG0}*,* $attr_{AG0}$*,* t_V^{AG0}*,* t_E^{AG0} *and* t_A^{AG0} *model the* ***initial graph typed over*** AT*, i.e., they are the relations that compose the relational representation of* $AG0^{AT}$*.*

- *Each collection* $(vert_{ALi},\ inc_{ALi}, attr_{ALi},\ t_V^{ALi},\ t_E^{ALi},\ t_A^{ALi},\ vert_{ARi},\ inc_{ARi},\ attr_{ARi},\ t_V^{ARi},\ t_E^{ARi},\ t_A^{ARi}, r_{i_V},\ r_{i_E}, r_{i_A}, var_i, |ruleEqns|_i)$ *defines a* ***rule****.*

Proposition 16. *The relational structure* $|AGG|$ *is well-defined.*

Proof. Follows immediately from Propositions 11, 13 and 15. □

The definition of the attributed match is also analogous to the one without attributes. However here the match should also include the mapping between the corresponding algebras. Since an assignment of values to the variables involved in the rule uniquely determines the corresponding algebra homomorphism, we will restrict the mapping to these variables in the relational representation of an attributed match. Note that X may contain variables that are not in L, and therefore the image of this assignment may not be completely in the graph to which the rule is being applied. That is why the relational representation of an attributed match has 4 components: the relational representations of the left-hand side of a rule, the graph to which the rule shall be applied, and the match morphism; together with a relation representing the complete assignment of values to variables described by the match.

Definition 30 (Relational Representation of an Attributed Match)**.** *Given a specification $SPEC$, a rule over $SPEC$ $attRule = (r, X, ruleEqns)$ with $r : AL^{AT} \to AR^{AT}$, and a match $m = ((m_{Ver}, m_{Edge}), m_{Alg})$ from $\overline{AL^{AT}}$ to AG^{AT}, with $AG = (G, A_G, attr_G)$,* ***a relational representation of*** *m is given by a tuple $\langle asg, |\overline{AL^{AT}}|, |m|, |AG^{AT}|\rangle$ where:*

- *$asg \subseteq X \times \mathcal{U}(A_G)$ is a relation that corresponds to the algebra homomorphism m_{Alg}, restricted to the variables in X;*
- *$|AL^{AT}|$ and $|AG^{AT}|$ are relational representations of typed attributed graphs AL^{AT} and AG^{AT}, respectively;*
- *$|m| = \{m_{Vert}, m_{Edge}, asg_L\}$ is a relational typed attributed graph morphism from $|AL^{AT}|$ to $|AG^{AT}|$ where*
  - *asg_L is a restriction of asg to the variables appearing in L*

Proposition 17. *A relational representation of an attributed match is well-defined.*

Proof. According to Proposition 13 the relational representation of typed attributed graphs is well defined and according to Proposition 14 the relational representation of a typed attributed graph morphism is well-defined. The definition of the relational typed attributed graph morphism guarantees that it represents the morphism given. Then, the morphism is total, the component that relates edges is injective and asg satisfies all equations in $ruleEqns$. □

For the definition of rule application as a transduction, everything of Def. 16 remains the same. We have just to extend the condition of rule application and add formulas, which will respectively specify the attribution function of graphs, the data values component of typing morphisms and the set of variables and equations of rules. That is, some formulas must be added in the definition of Δ. Table 4.1 describes the intuitive meaning and the notation used in the following definition.

Definition 31 (Rule Application as Definable Transduction for Attributed Graph Grammars)**.** *Let $AGG = (AT, AG0, R)$ be an attributed graph grammar over a specification SPEC and an algebra A, such that the sets of edges and vertices of graphs AT, $AG0$, ALi and ARi are disjoint, and let $|AGG|$ be the relational structure associated to AGG. Given a rule $attRule = (\alpha i, X, ruleEqns)$, $\alpha i : ALi^{AT} \to ARi^{AT}$, of AGG and a corresponding match $m = ((m_{Ver}, m_{Edge}), m_{Alg})$ from ALi^{AT} to $AG0^{AT}$, with the relational representations respectively given by $|attRule| = \langle |ALi^{AT}|, |\alpha i|, |ARi^{AT}|, var, |ruleEqns|\rangle$ and $\langle asg, |ALi^{AT}|, |m|, |AG0^{AT}|\rangle$, $\Delta = (\varphi, \psi, (\theta_q)_{q \in \mathcal{R}_{AGG}})$, with $\mathcal{W} = \varnothing$,* ***defines a transduction that maps an attributed graph grammar*** *$|AGG|$* ***to an attributed graph grammar*** *$|AGG|'$, such that $|AG0|'$ (initial state of $|AGG|'$) corresponds to the result of the application of rule $|attRule|$ at match $|m|$ in $|AG0|$ (initial state of $|AGG|$), where:*

φ *expresses that $|m| = \{m_V, m_E, asg_L\}$ defines a total relational typed attributed graph morphism, with m_E injective (as in Def.16) and that asg satisfies all equations in $|ruleEqns|$.*

ψ, $\theta_{vert_{AT}}$, $\theta_{inc_{AT}}$, $\theta_{vert_{AG0}}$, $\theta_{inc_{AG0}}$, $\theta_{t_V^{AG0}}$, $\theta_{t_E^{AG0}}$, $\theta_{vert_{ALi}}$, $\theta_{inc_{ALi}}$, $\theta_{t_V^{ALi}}$, $\theta_{t_E^{ALi}}$, $\theta_{vert_{ARi}}$, $\theta_{inc_{ARi}}$, $\theta_{t_V^{ARi}}$, $\theta_{t_E^{ARi}}$, $\theta_{\alpha_{i_V}}$, $\theta_{\alpha_{i_E}}$ *are the same formulas specified in Def. 16.*

$\theta_{attr_{AT}}$ *is the formula $attr_{AT}(x, y)$.*

$\theta_{attr_{AG0}}(x, y)$ *is the formula* $nattr_{AG0}(x, y) \vee nattr_{ARi}(x, y)$.

$\theta_{t_A^{AG0}}(x, t)$ *is the formula* $nt_A^{AG0}(x, t) \vee nt_A^{ARi}(x, t)$.

$\theta_{attr_{ALi}}$*,* $\theta_{t_A^{ALi}}$*,* $\theta_{attr_{ARi}}$*,* $\theta_{t_A^{ARi}}$*,* $\theta_{\alpha_{i_A}}$*,* θ_{var_i}*,* $\theta_{|ruleEqns|_i}$ *are respectively the formulas* $attr_{ALi}(x, y)$*,* $t_A^{ALi}(x, y)$*,* $attr_{ARi}(x, y)$*,* $t_A^{ARi}(x, y)$*,* $\alpha_{i_A}(x)$ $var_i(x)$ *and* $|ruleEqns|_i(x, y)$*, for* $i = 1 .. n$.

Table 4.1: Formulas used in θ specifications

Notation	**Intuitive Meaning**	**Formula**
$attr_G(x, y)$	x is an attribute edge of graph G with value y.	$attr_G(x, y)$
$t_A^G(x, y)$	x is a value of graph G of type y.	$t_A^G(x, y)$
$var_i(x)$	x is a variable over the sorts of $SPEC$.	$var_i(x)$
$\|ruleEqns\|_i(x, y)$	$x = y$ is an equation over $SPEC$.	$\|ruleEqns\|_i(x, y)$
$nattr_{AG0}(x, y)$	x is an attribute edge of graph $AG0$ with value y that is not image of the match.	$attr_{AG0}(x, y) \wedge \nexists w\Big(m_E(w, x)\Big)$
$nattr_{ARi}(x, y)$	x is an attribute edge of graph ARi with value w, that is assigned, by the match component asg, to y	$\exists w\Big[attr_{ARi}(x, w) \wedge asg(w, y)\Big]$
$nt_A^{AG0}(x, t)$	x is a value of graph $AG0$ of type t of an attribute edge that is not image of the match.	$\exists y\Big(attr_{AG0}(y, x)\Big)\wedge$ $\wedge\nexists w\Big(m_E(w, y)\Big) \wedge t_A^{AG0}(x, t)$
$nt_A^{ARi}(x, t)$	x is a value assigned by asg to the value of an attribute edge of type t of graph ARi .	$\exists y, w\Big[attr_{ARi}(y, w) \wedge asg(w, x) \wedge$ $t_A^{ARi}(w, t)\Big]$

The well-definedness of the rule application as a definable transduction is still valid for the attributed version. The proof is analogous to the proof of Proposition 9. We have just to include the relations that define the attributed version of the graphs and morphisms.

Proposition 18. *The rule application as a definable transduction for attributed graph grammars is well-defined.*

Proof. Assume a fixed specification SPEC and a fixed algebra A. Let $|AGG|'$ be the result of the transduction applied to attributed graph grammar $|AGG|$ over SPEC corresponding to the application of relational rule $|attRule|$ at relational match $\langle asg, |\overline{ALi^{AT}}|, |m|, |AG0^{AT}|\rangle$, with $|m| = \{m_V, m_E, asg_L\}$. Considering that the given rule $|attRule| = \langle|ALi^{AT}|, |\alpha i|, |ARi^{AT}|, var, |ruleEqns|\rangle$, with $|\alpha_i| = \{\alpha_{i_V}, \alpha_{i_E}, \alpha_{i_A}\}$ and the relational match specified above are the relational representations of $attRule = (\alpha_i, X, ruleEqns)$ over SPEC, with $\alpha_i : ALi^{AT} \to ARi^{AT}$, and $m : \overline{ALi^{AT}} \to AG0^{AT}$, respectively, and considering AH^{AT} with $AH = (H, A_H, attr_H)$ to be the typed attributed graph obtained by the application of $attRule$ to graph $AG0^{AT}$ at match m (according to Definition 23) we have to show that[1]:

[1]Each relation r of $|AGG|'$ will be denoted by r' to avoid confusion with the relations of $|AGG|$ (denoted by the unprimed names).

1. $vert'_{AT}$, inc'_{AT} and $attr'_{AT}$ are the relations of a $\mathcal{R}_{agr}$-structure $|AT|' = \langle V'_T \cup E'_T \cup rng(attr_T), \{vert'_{AT}, inc'_{AT}, attr'_{AT}\}\rangle$ representing graph $AT = (T, A, attr_T)$, with $T = (V_T, E_T, src_T, trg_T)$.

   - $x \in vert'_{AT}$ *iff* $x \in V_T$, and $(x, y, z) \in inc'_{AT}$ *iff* $x \in E_T \wedge src_T(x) = y \wedge trg_T(x) = z$: Both are assured by Proposition 9.
   - $(x, y) \in attr'_{AT}$ *iff* $(x, y) \in E_T \times rng(attr_T) \wedge attr_T(x) = y$: By $\theta_{attr_{AT}}$ definition, $x \in attr'_{AT}$ *iff* $(x, y) \in attr_{AT}$. Since $attr_{AT}$ is the relation of a $\mathcal{R}_{agr}$-structure representing AT, then (following Definition 24) $(x, y) \in attr_{AT}$ *iff* $(x, y) \in E_T \times rng(attr_T) \wedge attr_T(x) = y$.

2. $vert'_{AG0}$, inc'_{AG0} and $attr'_{AG0}$ are the relations of a $\mathcal{R}_{agr}$-structure $|AG0|' = \langle V'_{G0} \cup E'_{G0} \cup rng(attr'_{G0}), \{vert'_{AG0}, inc'_{AG0}, attr'_{AG0}\}\rangle$ representing graph $AH = (H, A, attr_H)$, with $H = (V_H, E_H, src_H, trg_H)$.

   - $x \in vert'_{AG0}$ *iff* $x \in V_H$, and $(x, y, z) \in inc'_{AG0}$*iff* $x \in E_H \wedge src_H(x) = y \wedge trg_H(x) = z$: Follows from Proposition 9.
   - $(x, y) \in attr'_{AG0}$ *iff* $(x, y) \in E_H \times rng(attr_H) \wedge attr_H(x) = y$: By $\theta_{attr_{AG0}}$ definition, $x \in attr'_{AG0}$ *iff* $\Big((x, y) \in attr_{AG0} \wedge \nexists w, (w, x) \in m_E\Big)$ or $\exists w\Big((x, w) \in attr_{ARi} \wedge (w, y) \in asg\Big)$
     - Let $(x, y) \in attr_{AG0}$, such that $\nexists w, (w, x) \in m_E$. Since $|AG0|$ is the relational representation of $AG0$, we have $x \in E_{G0} \wedge y \in rng(attr_{G0}) \wedge attr_{G0}(x) = y$. Also, as $|m|$ is a relational representation of $m = ((m_{Vert}, m_{Edge}), m_{Alg})$, by Definition 30, $\nexists w, m_{Edge}(w) = x$. As a result, $x \in E_{G0} - m_{Edge}(E_{ALi})$, i.e. by Definition 23, $x \in E_H$. In this case, the attribute of x in H is the same of $G0$, i.e., $attr_H(x) = y$ and $y \in rng(attr_H)$.
     - Let $(x, w) \in attr_{ARi}$, with $(w, y) \in asg$. Considering that $|ARi|$ is a relational representation of Ri we have $x \in E_{Ri} \wedge w \in rng(attr_{Ri}) \wedge attr_{Ri}(x) = w$. Consequently, by Definition 23, $x \in E_H$. In this case, the attribute of x in H is given by the result of m_{Alg} applied to the attribute of x, i.e., the result of $m_{Alg}(w)$. Since asg corresponds to the algebra homomorphism m_{Alg} restricted to the variables in X, we have $m_{Alg}(w) = y$. Then, $attr_H(x) = m_{Alg}(w) = y$ and $y \in rng(attr_H)$.

   The proof in the other direction (only if case) is analogous.

3. $t_V^{AG0'}$, $t_E^{AG0'}$ and $t_A^{AG0'}$ are from the set $|t^{AG0}|'$ such that the tuple $\langle |AG0|', |t^{AG0}|', |AT|'\rangle$ is a relational representation of the typed attributed graph $AH^{AT} = (AH, t^{AH}, AT)$.

   - $(x, t) \in t_V^{AG0'}$ *iff* $t^H_{Vert}(x) = t$, and $(x, t) \in t_E^{AG0'}$ *iff* $t^H_{Edge}(x) = t$: Follows from Proposition 9.
   - $(x, t) \in t_A^{AG0'}$ *iff* $t^H_{Alg}(x) = t$: By $\theta_{t_A^{AG0}}$ definition, $(x, t) \in t_A^{AG0'}$ *iff* $\Big(\exists y, (y, x) \in attr_{AG0} \wedge \nexists w, (w, y) \in m_E \wedge (x, t) \in t_A^{AG0}\Big)$ or $\Big(\exists y, w, \big((y, w) \in attr_{ARi} \wedge (w, x) \in asg \wedge (w, t) \in t_A^{ARi}\big)\Big)$.
     - Let $(x, t) \in t_A^{AG0}$, such that $\exists y, (y, x) \in attr_{AG0}$ and $\nexists w, (w, y) \in m_E$. Since $|AG0|$, $|m|$ and $\langle |AG0|, |t^{AG0}|, |AT|\rangle$ are relational representations

of $AG0$, m and $AG0^{AT}$, respectively, we have $y \in E_{G0}$, $\nexists w, m_{Edge}(w) = y$, $attr_{G0}(y) = x$ and $t^{AG0}_{Alg}(x) = t$. I.e., $x \in (E_{G0} - m_{Edge}(E_{Li}))$. Then, by Definition 23, $t^{H}_{Alg}(x) = t^{AG0}_{Alg}(x) = t$.

– Let $(y, w) \in attr_{ARi}$, $(w, t) \in t^{ARi}_{A}$ and $(w, x) \in asg$. Since $\langle |ARi|, |t^{ARi}|, |AT| \rangle$ is the relational representation of the attributed typed graph ARi^{AT}, we have $y \in E_{Ri}$, $w \in rng(attr_{Ri})$ and $t^{ARi}_{Alg}(w) = t$. Also, considering that asg corresponds to the algebra homomorphism m_{Alg} restricted to X, $m_{Alg}(w) = x$. Thus, by Definition 23, $t^{H}_{Alg}(x) = t^{ARi}_{Alg}(w) = t$.

The only if proof is similar.

$\square$

4.3 Token Ring Example with Attributed Graphs

In this subsection we modify and extend the token-ring protocol. The basic idea of the protocol remains the same: all stations are connected in a ring and each station can receive transmissions only from its immediate neighbor. Permission to transmit is granted by a token that circulates around the ring. Now, we include a buffer to store received messages in each station, and each station has its own buffer size. For this example, we will use the types **Nat** for natural numbers and **Status**, that can be either active or standby. These data types can be described by the algebraic specification **TRing**$=(SIG_{\textbf{TRing}}, Eqns)$: the signature is shown in Figure 4.5, we omitted the equations (they are the usual ones for the corresponding functions on natural numbers, there are no equations for sort $Status$).

TRing : **sorts** Status, Nat
opns
active : $\rightarrow$ Status
standby : $\rightarrow$ Status
0 : $\rightarrow$ Nat
succ : Nat $\rightarrow$ Nat
+ : Nat $\times$ Nat $\rightarrow$ Nat
- : Nat $\times$ Nat $\rightarrow$ Nat
mod : Nat $\times$ Nat $\rightarrow$ Nat

Figure 4.5: Signature $SIG_{\textbf{TRing}}$

Models for algebraic specification are algebras, and they are constructed by assigning a set to each sort name (called carrier set) and a function to each operation symbol. Moreover, functions shall be compatible with the equations of the specification. In our approach, we will use three different models for each specification: a final model (to define the type graph), a term-algebra (to be used in rules), and a concrete “value" algebra (that is used to attribute the initial and all reachable graphs). For the token ring example, these algebras are shown in Figures 4.6, 4.7 and 4.8, respectively.

All these algebras are possible interpretations of the symbols in **TRing**. The carrier sets define which elements may be used as attribute values. There is a unique homomorphism from any algebra to F^{TRing} (because there is only one possible way in which we can map elements of the corresponding carrier sets). Moreover, if we fix an assignment from X to values in A^{TRing}, there is also only one possible way in which we can map $T^{TRing}(X)$ to A^{TRing}.

$F_{Status} = \{Status\}$

$F_{Nat} = \{Nat\}$

$active^F :\to F_{Status}$	$active^F() = Status$
$standby^F :\to F_{Status}$	$standby^F() = Status$
$0^F :\to F_{Nat}$	$0^F() = Nat$
$succ^F : F_{Nat} \to F_{Nat}$	$\forall n \in F_{Nat} : succ^F(n) = Nat$
$+^F : F_{Nat} \times F_{Nat} \to F_{Nat}$	$\forall n1, n2 \in F_{Nat} : +^F(n1, n2) = Nat$
$-^F : F_{Nat} \times F_{Nat} \to F_{Nat}$	$\forall n1, n2 \in F_{Nat} : -^F(n1, n2) = Nat$
$mod^F : F_{Nat} \times F_{Nat} \to F_{Nat}$	$\forall n1, n2 \in F_{Nat} : mod^F(n1, n2) = Nat$

Figure 4.6: Final Algebra $F^{TRing} = (F_{Status}, F_{Nat}, active^F, standby^F, 0^F, succ^F, +^F, -^F, mod^F)$

$X = (X_{Status}, X_{Nat})$ with $X_{Status} = \{x, y\}$ and $X_{Nat} = \{n, m, p\}$

$T_{Status} = \{active, standby, x, y\}$

$T_{Nat} = \{0, n, m, p, succ(0), succ(n), succ(m), succ(p), succ(succ(0)), succ(succ(n)), succ(0) + n, n + m, ...\}$

$active^T :\to T_{Status}$	$active^T() = active$
$standby^T :\to T_{Status}$	$standby^T() = standby$
$0^T :\to T_{Nat}$	$0^T() = 0$
$succ^T : T_{Nat} \to T_{Nat}$	$\forall n \in T_{Nat} : succ^T(n) = succ(n)$
$+^T : T_{Nat} \times T_{Nat} \to T_{Nat}$	$\forall n1, n2 \in T_{Nat} : +^T(n1, n2) = n1 + n2$
$-^T : T_{Nat} \times T_{Nat} \to T_{Nat}$	$\forall n1, n2 \in T_{Nat} : -^T(n1, n2) = n1 - n2$
$mod^T : T_{Nat} \times T_{Nat} \to T_{Nat}$	$\forall n1, n2 \in T_{Nat} : mod^T(n1, n2) = n1\, mod\, n2$

Figure 4.7: Term Algebra $T^{TRing}(X) = (T_{Status}, T_{Nat}, active^T, standby^T, 0^T, succ^T, +^T, -^T, mod^T)$

$A_{Status} = \{act, stb\}$	
$A_{Nat} = \{0, 1, 2, 3, 4, 5, 6, ...\}$	
$active^A :\rightarrow A_{Status}$	$active^A() = act$
$standby^A :\rightarrow A_{Status}$	$standby^A() = stb$
$0^A :\rightarrow A_{Nat}$	$0^A() = 0$
$succ^A : A_{Nat} \rightarrow A_{Nat}$	usual successor function for naturals
$+^A : A_{Nat} \times A_{Nat} \rightarrow A_{Nat}$	usual sum function for naturals
$-^A : A_{Nat} \times A_{Nat} \rightarrow A_{Nat}$	usual subtraction function, with $n1 - n2 = 0$, if $n1 < n2$
$mod^A : A_{Nat} \times A_{Nat} \rightarrow A_{Nat}$	usual modulo function for naturals

Figure 4.8: Value Algebra $A^{TRing} = (A_{Status}, A_{Nat}, active^A, standby^A, 0^A, succ^A, +^A, -^A, mod^A)$

Now that the data part is defined, we can construct the grammar that describes the behaviour of the modified token ring. The type graph and the initial graph are depicted in Figure 4.9. The rules are illustrated in Figure 4.10. In the graphical representation of rules, we only draw the attribute (edges) that are modified by the rule (however, formally, all attributes are part of each graph). Also, for convenience, we used same variable names in different rules (but this specification can be translated to an equivalent one using different variable names).

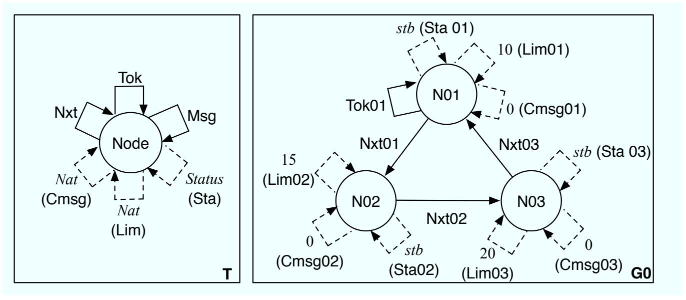


Figure 4.9: Type Graph and Initial Graph

Type graph: We replaced the Act and Stb edges of the previous specification by an attribute of type $Status$ (Sta). Moreover, we included two attributes of type Nat: one to control how many messages are currently in a station (Cmsg) and other to establish a limit to the buffer of received messages in each node (Lim). Note that, since the final algebra is used to construct the type graph, the values associated to attributes Cmsg, Lim and Sta are Nat, Nat and $Status$, respectively. Thus, this attributed graph actually defines not only the types of graphical elements, but also the types of (data) attributes that will be allowed in any instance graph.

Initial graph: The values of attributes are taken from algebra A^{TRing}. Initially, no station is transmitting through the network (the value of edges of type Sta is stb) and the message buffers are empty (the value of each Cmsg-typed edge is 0). The values of the Lim-typed edges specify the limit of received (and not treated) messages of each station.

Rules: We use the term algebra $T^{TRing}(X)$ to specify the rules. Rules $r1$, $r2$ and $r4$ keep, respectively, the same meaning of rules $\alpha1$, $\alpha2$ and $\alpha4$ previously presented, but now there is an attribute (edge) of type Sta, attributed with a variable name x. Equations are used to ensure that each rule can only be applied in case the node is in the required status: $x = standby$ for rules $r1$ and $r2$ or $x = active$ for rule $r4$. The status of the node after the application of the rule is determined by the variable in the right-hand side and the respective equation of the rule. Rule $r3$ (as $\alpha3$) also handles the receipt of a message by a $standby$ node. This rule can only be applied if the buffer of received messages has not achieved the limit, i.e., if $m < p$ (determined by the condition $(m \bmod p) = m$). In this case, the message is passed to the next node and the counter of messages is incremented. Rule $r6$ simulates the treatment of the message by a station by decrementing the message counter. Rule $r5$ (as $\alpha5$) can be applied to insert a new node into the ring. The node is inserted with a buffer that stores at most 10 messages.

Next, we describe the main steps involved in the proof of the following property: the buffer of each node never exceeds its limit. First of all, we must define two functions in the standard library: one to determine pairs of edges of fixed types with source and target in the same vertex, and another to indicate the attributes of edges of a reachable graph.

[Library function $Loop$: Edges with source and target in the same vertex] This function that returns pairs of edges (e, f), with e of type t_1 and f of type $t2$, with source and target in the same vertex:

$$Loop_{t_1,t_2}\ G0 = \Big\{(e,f) \mid \exists x\ [inc_{G0}(e,x,x) \wedge inc_{G0}(f,x,x)] \wedge t_E^{G0}(e,t_1) \wedge t_E^{G0}(f,t_2)\Big\} \tag{4.1a}$$

$$\begin{aligned}Loop_{t_1,t_2}\ ap(\alpha i,m)\ g = \Big\{(e,f) \mid & \Big[\exists x\ [inc_{Ri}(e,x,x) \wedge inc_{Ri}(f,x,x)] \wedge t_E^{Ri}(e,t_1)\wedge \\ & \wedge t_E^{Ri}(f,t_2)\Big] \vee && (4.1b)\\ & \Big[(e,f) \in Loop_{t_1,t_2}\ g \wedge \nexists w\ m_{E_{\alpha i}}(w,e) \wedge \nexists w\ m_{E_{\alpha i}}(w,f)\Big] \vee \\ & && (4.1c)\\ & \Big[\exists x\ inc_g(e,x,x) \wedge t_E^{g}(e,t_1) \wedge \nexists w\ m_{E_{\alpha i}}(w,e)\wedge \\ & \wedge \exists y\ inc_{Ri}(f,y,y) \wedge t_E^{Ri}(f,t_2) \wedge \exists z\ [\alpha i_E(z,y)\wedge \\ & \wedge m_{V_{\alpha i}}(z,x)]\Big] \vee && (4.1d)\\ & \Big[\exists x\ inc_g(f,x,x) \wedge t_E^{g}(f,t_2) \wedge \nexists w\ m_{E_{\alpha i}}(w,f)\wedge \\ & \exists y\ inc_{Ri}(e,y,y) \wedge t_E^{Ri}(e,t_1)\wedge \\ & \exists z\ [\alpha i_E(z,y) \wedge m_{V_{\alpha i}}(z,x)]\Big]\Big\} && (4.1e)\end{aligned}$$

For the initial graph, the pairs are determined by relations inc_{G0} and t_E^{G0} of $|GG|$ (4.1a). For the result of the application of rule αi at match $m = \{m_{V_{\alpha i}}, m_{E_{\alpha i}}, asg_L\}$ to graph g, the pairs are either edges of the right-hand side of the rule (4.1b), edges of graph g with source and target in the same vertex that are not image of the match (4.1c), or pairs of edges, one of Ri and other of g (that is not image of the match), which have source and

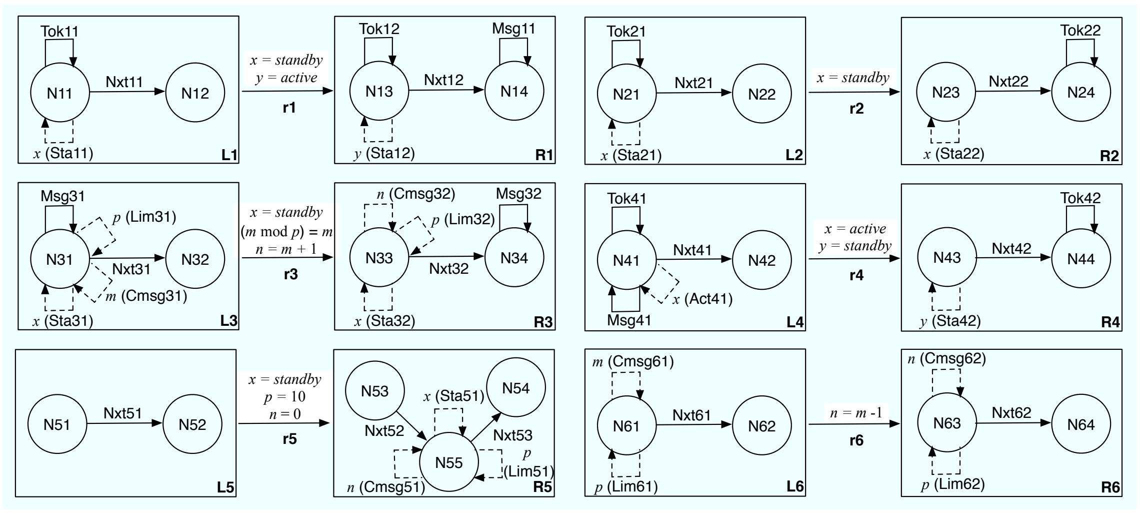


Figure 4.10: Rules

target in the same vertex after the application of the rule ((4.1d) or (4.1e)). The guarantee of having source and target in the same vertex after the application of the rule is stated by the last term of (4.1d) and (4.1e). ■

[Library function $Attr_E$: Attributes of edges of a reachable graph] The set of pairs $(edge,\ attribute)$ of a reachable graph are recursively defined by:

$$Att_E\ G0 = \{(e,a) \mid attr_{G0}(e,a)\} \tag{4.2a}$$

$$Att_E\ ap(\alpha i,m)\ g = \{(e,a) \mid attr_{Ri}(e,a) \vee [(e,a) \in Att_E\ g \wedge \nexists w\ m_{E_{\alpha i}}(w,e)]\} \tag{4.2b}$$

If we take the initial graph (4.2a), the pairs are specified by the relation $attr_{G0}$ of the relational structure. If we consider the graph obtained from the application of rule αi at match $m = \{m_{V_{\alpha i}}, m_{E_{\alpha i}}, asg_L\}$ to graph g (4.2b), the attributes are either the attributes of edges of the right-hand side of the rule or the attributes of edges (that are not image of the match) of g. ■

The proof strategy applied in verification of properties is the same described before: we use mathematical induction, considering that the relations of the relational structure define axioms to be used during the proof. Now, since we use variables as attributes in the left- and right-hand sides of rules, in many cases, at the inductive step the development of the proof involves variables. In this case, in order to establish the property, we must regard the equations of the applied rule as “local axioms". We say “local" because the equations of each rule can only be regarded as axioms for the step of the proof that involves the application of that rule. This can be done because, to apply a rule, we assume that there is a match that makes these equations true, and the property is proven only for such matches (because in other cases, it would not be possible to apply the rule). Now we can state the property to be proven.

Property 3. *The attributes of edges of type $Cmsg$ are always less than the attributes of edges of type Lim, if they both have source and target in the same vertex.*

According to the definitions previously established in the library, the property to be proven can be enunciated by the formula:

$$\forall(e_1,e_2) \in \mathit{Loop}_{\mathsf{Cmsg,Lim}} reach\_gr.[(e_1,a_1) \in \mathit{Att}_E\, reach\_gr \wedge (e_2,a_2) \in \mathit{Att}_E\, reach\_gr \Rightarrow \\ \Rightarrow a_1 \leq a_2]$$

Proof.

Basis: We have to prove

$$\forall(e_1,e_2) \in \mathit{Loop}_{\mathsf{Cmsg,Lim}} G0.[(e_1,a_1) \in \mathit{Att}_E\ G0\ \wedge (e_2,a_2) \in \mathit{Att}_E\ G0 \Rightarrow a_1 \leq a_2]$$

Considering the result of the function $\mathit{Loop}_{\mathsf{Cmsg,Lim}} G0$ (equations (4.1)) and the definition of Att_E (equation (4.2a)), this formula reduces to
$\forall(e_1,e_2) \in \{(\mathsf{Cmsg01},\mathsf{Lim01}),(\mathsf{Cmsg02},\mathsf{Lim02}),(\mathsf{Cmsg03},\mathsf{Lim03})\ .[\mathrm{attr_{G0}(e_1,a_1)} \wedge \mathrm{attr_{G0}(e_2,a_2)} \Rightarrow \mathrm{a_1 \leq a_2}]$

Now the implication must be verified for each pair of edges. For the first instance, consulting the relational structure associated to the graph grammar, the pair of edges/attributes that satisfies the antecedent are $(\mathsf{Cmsg01}, 0)$ and $(\mathsf{Lim01}, 10)$. Since $0 \leq 10$ the consequent is evaluated to true. The verification for other instances is similar. Thus, the property is valid for the initial graph.

Hypothesis: Assume that the property is valid for any reachable graph G:

$$\forall(e_1,e_2) \in \mathit{Loop}_{\mathsf{Cmsg,Lim}} G.[(e_1,a_1) \in \mathit{Att}_E\ G\ \wedge (e_2,a_2) \in \mathit{Att}_E\ G \Rightarrow a_1 \leq a_2]$$

Inductive Step: We have to prove

$$\forall(e_1,e_2) \in \mathit{Loop}_{\mathsf{Cmsg,Lim}} ap(ri,m)G.[(e_1,a_1) \in \mathit{Att}_E\ ap(ri,m)G \wedge \\ \wedge (e_2,a_2) \in \mathit{Att}_E\ ap(ri,m)G \Rightarrow a_1 \leq a_2]$$

Since we have 6 rules, we have 6 cases to consider (note that, although not depicted in Figure 4.10, all attribute edges are part of each left- and right-hand side of the rules, due to the attribute completeness requirement):

Rule $r1$**:** First, we have to construct $\mathit{Loop}_{\mathsf{Cmsg,Lim}} ap(r1,m)G$ and then check whether these pairs satisfy the required property. Since the kinds of edges we are considering are attribute edges and due to the attribute completeness property required for (typed) attributed graphs, the graphs G, L and R will have values for both attributes Cmsg and Lim. This means that we only have to consider the cases described by equations (4.1b) and (4.1c) in the definition of Loop:

(i) A pair (e,f) **that satisfies (4.1b)** : In rule $r1$, such pairs are $(\mathsf{Cmsg13},\mathsf{Lim13})$ and $(\mathsf{Cmsg14},\mathsf{Lim14})$. Assume that the names of variables associated to attributes Cmsg and Lim in graph $R1$ are $cmsg13$ and $lim13$ (connected to node $N13$) and $cmsg14$ and $lim14$ (connected to node $N14$). Then, the function Att_E will return $attr_{R1}(\mathsf{Cmsg13},\mathrm{cmsg13})$ and $attr_{R1}(\mathsf{Lim13},\mathrm{lim13})$, plus all pairs $attr_G(\mathsf{Cmsgi},\mathrm{cmsgi})$ and $attr_G(\mathsf{Limi},\mathrm{limi})$, for each node i of G that is not in the image of match m. Thus, what we have to verify is if $cmsg13 \leq lim13$ and $cmsg14 \leq lim14$. But, since these attributes were not changed by the rule, this is the same as verifying if $cmsg11 \leq lim11$ and $cmsg12 \leq lim12$ (the corresponding variables in the left-hand side of the rule). But considering that there is a match m mapping the left-hand side to G, there are corresponding values in G for these variables. By induction hypothesis, all pairs (e^G, f^G) that come from G satisfy the property, and therefore we conclude that the pair (e, f) also satisfies the property.

(ii) A pair (e, f) that satisfies (4.1c) : This pair is in $Loop_{\mathsf{Cmsg},\mathsf{Lim}}G$, and therefore by induction hypothesis satisfies the property.

Rules $r2$ and $r4$: Analogous to rule $r1$.

Rule $r3$: We start analogously to the case of $r1$, find out that we have to prove that $n \leq p$ and $cmsg34 \leq lim34$. The latter is analogous to case $r1$. To prove that $n \leq p$, we have to consider the equations of $r3$ as axioms. Then, considering $n = m + 1$ and $m \, mod \, p = m$ as valid formulas, and using the pre-defined theories corresponding to the used specification, it is possible to prove (using a theorem prover in a semi-automated way) that the property is satisfied for this case.

Rule $r5$: Here, we will have to prove for the newly created node that $n \leq p$ (for the other nodes, the proof is similar to the previous cases). Assuming the equation as an axiom, we have $n = 0$ and $p = 10$ (actually, the last equation is $p = succ^{10}(0)$). But any possible match m would need to assign the value zero to n and 10 to p (otherwise, it would not be a match for this rule). Therefore, we can conclude that $n \leq p$.

Rules $r6$: Again, here we will have to prove that $n \leq p$, assuming $n = m-1$ as true. But, since the attribute Lim is preserved by the rule, it must be also in the left-hand side, that is matched via m to a value, say $limG$, in G. Moreover, variable m must also be mapped to a value, say $cmsgG$ in G. Since G satisfies the property by induction hypothesis, we have that $cmsgG \leq limG$, and therefore we can conclude that the $m \leq p$. Together with the fact that $n = m - 1$, this makes $n \leq p$ true.

$\square$

5 EXTENDING THE APPROACH TO GRAPH GRAMMARS WITH NEGATIVE APPLICATION CONDITIONS

Application conditions specify conditions under which rules can be applied to a given state-graph in order to obtain a new state-graph. Application conditions generally comprise *contextual conditions*, specifying the existence (in case of positive conditions) or non-existence (in case of negative ones) of nodes, edges or subgraphs in the given graph and *embedding restrictions*, regarding the match morphisms. Until now, we considered only one restriction on matches: that they are injective on edges. The main purpose of the graph grammars that we have in mind in this thesis is the specification of concurrent and reactive systems. In such systems, a non-injective match means that the member of needed resource is not relevant.

In this chapter, we propose to extend the relational approach to graph grammars with contextual conditions, particularly negative ones. Negative application conditions (NACs) restrict the application of a rule by asserting that a specific structure must not be present in a state-graph before applying the rule. We adopt the concept of NACs introduced by Habel, Heckel and Taentzer in (HABEL; HECKEL; TAENTZER, 1996) due to two main reasons: first, because they propose this extension in the framework of the single-pushout approach (the same one we have been applying) and second, because their approach has a visual representation that does not affect the graphical structure of the specifications. In their work, they also proved that rules defined with both positive and negative application conditions can be expressed (through context enlargement) by new rules with just negative application conditions.

The extension of graph grammars by application conditions may raise our flexibility in the use of the relational approach for the specification of systems in all kinds of application areas. As emphasized in (HABEL; HECKEL; TAENTZER, 1996), application conditions are a necessary component of every nontrivial specification. If we do not specify them formally, we will not be able to analyse them formally.

5.1 Graph Grammar with NACs

Graph grammars with negative application conditions are graph grammars whose rules are enriched with negative application conditions. Negative application conditions are expressed by sets of total morphisms starting from the left-hand side of the rules.

Definition 32 (Rule with Negative Application Conditions)***.*** *A **rule with negative application conditions (NACs)** is a pair $\hat{\alpha} = (\alpha : L^T \to R^T, AN(\alpha))$ consisting of a rule $\alpha : L^T \to R^T$ with respect to T (Def. 3) and a set of negative application conditions*

$AN(\alpha) \subseteq \mathcal{MOR}(L^T)$, where $\mathcal{MOR}(L^T)$ denotes the set of all total typed graph morphisms from the typed graph L^T to graphs typed over T.

Graph grammars with NACs allows the specification of a set of NACs for each rule of the grammar.

Definition 33 (Graph Grammar with NACs)**.** *A **graph grammar with negative application conditions** is a tuple $GGN = (T, G0, \hat{R})$, such that T is a type graph, $G0$ is a graph typed over T and $\hat{R}$ is a set of rules with NACs.*

A rule with NACs $\hat{\alpha} = (\alpha : L^T \to R^T, AN(\alpha))$ is applicable to a graph G^T if there is a match $m : L^T \to G^T$ that satisfies all NACs from $AN(\alpha)$.

Definition 34 (Match Satisfaction, Rules with NACs Application)**.** *Let $\hat{\alpha} = (\alpha : L^T \to R^T, AN(\alpha))$ be a rule with NACs and let $m : L^T \to G^T$ be a match of α in G^T. Then **match** m **satisfies a NAC** l **from** $AN(\alpha)$, with $l : L^T \to \hat{L}^T, l \in AN(\alpha)$ if there is not a total injective*[1] *graph morphism $n : \hat{L}^T \to G^T$ such that $n \circ l = m$.*

$$\begin{tikzcd}
\hat{L}^T \arrow[dr, dashed, "Xn"'] & L^T \arrow[l, "l"'] \arrow[r, "\alpha"] \arrow[d, "m"] & R^T \\
& G^T &
\end{tikzcd}$$

***Match** m **satisfies all NACs of** $\hat{\alpha}$, if it satisfies each NAC from $AN(\alpha)$. **Rule** $\hat{\alpha}$ **is applicable to** G^T **via** m, if m satisfies all NACs of $\hat{\alpha}$. If $\hat{\alpha}$ is applicable to G^T via m the **rule application with application condition** $G^T \overset{(\hat{\alpha},m)}{\Longrightarrow} H^T$ is the rule application $G^T \overset{(\alpha,m)}{\Longrightarrow} H^T$ (see Def. 5).*

5.2 Specifying the Token Ring Protocol with NACs

In this section we show the use of graph grammars with negative application conditions for the specification of the token-ring protocol. The intent of the protocol is the same and it follows the description detailed in Section 2.2.

The graphical representation of the graph grammar is illustrated in Figure 5.1. The adoption of NACs simplifies the type and the initial graphs, suppressing the Standby edges. NACs in rules $\alpha1$, $\alpha2$, $\alpha3$ and $\alpha5$ restrict the application of the rules to non-active stations. Rules $\alpha1$ and $\alpha2$ specify the behaviour of the protocol when a non-active station receives a token: it may hold the token and send a message to the next node, becoming an active station (rule $\alpha1$) or simply pass the token to the next station (rule $\alpha2$). By rule $\alpha3$ if a non-active station receives a message, it may pass the message to the next node. Rule $\alpha4$ is not modified, detailing the receipt of a message by an active station: it removes the message from the ring and sends the token to the next station. NACs in rule $\alpha5$ restrict the insertion of new stations in the ring to occur only among non-active stations. The concrete representation used in the relational approach (detailed in next section) is depicted in Figure 5.2.

It is important to notice that we require two separate constraints, $l15$ and $l25$, in rule $\alpha5$. This means that an Act edge in any of the stations is forbidden. A key feature consists

[1]We adopt injective satisfaction in order to express cardinality restrictions.

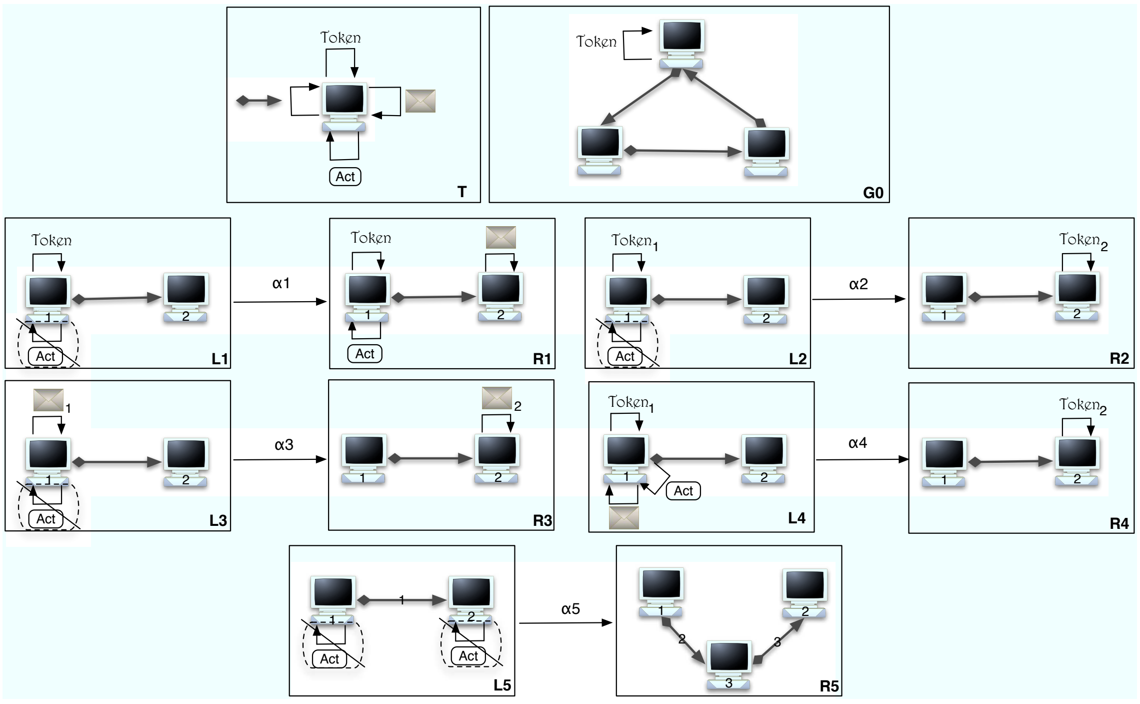


Figure 5.1: Token Ring Graph Grammar with NACs

in distinguishing this specification from rule $\alpha 5'$ depicted in Figure 5.3. In that case, an Act edge in both stations is forbidden, i.e. *both* objects must not exist at the same time. Given a graph consisting of just one of the two stations active, $\alpha 5'$ is applicable since there is no an Act edge in one of the stations, while production $\alpha 5$ is not applicable because of the existing edge.

5.3 Relational Representation of Graph Grammars with NACs

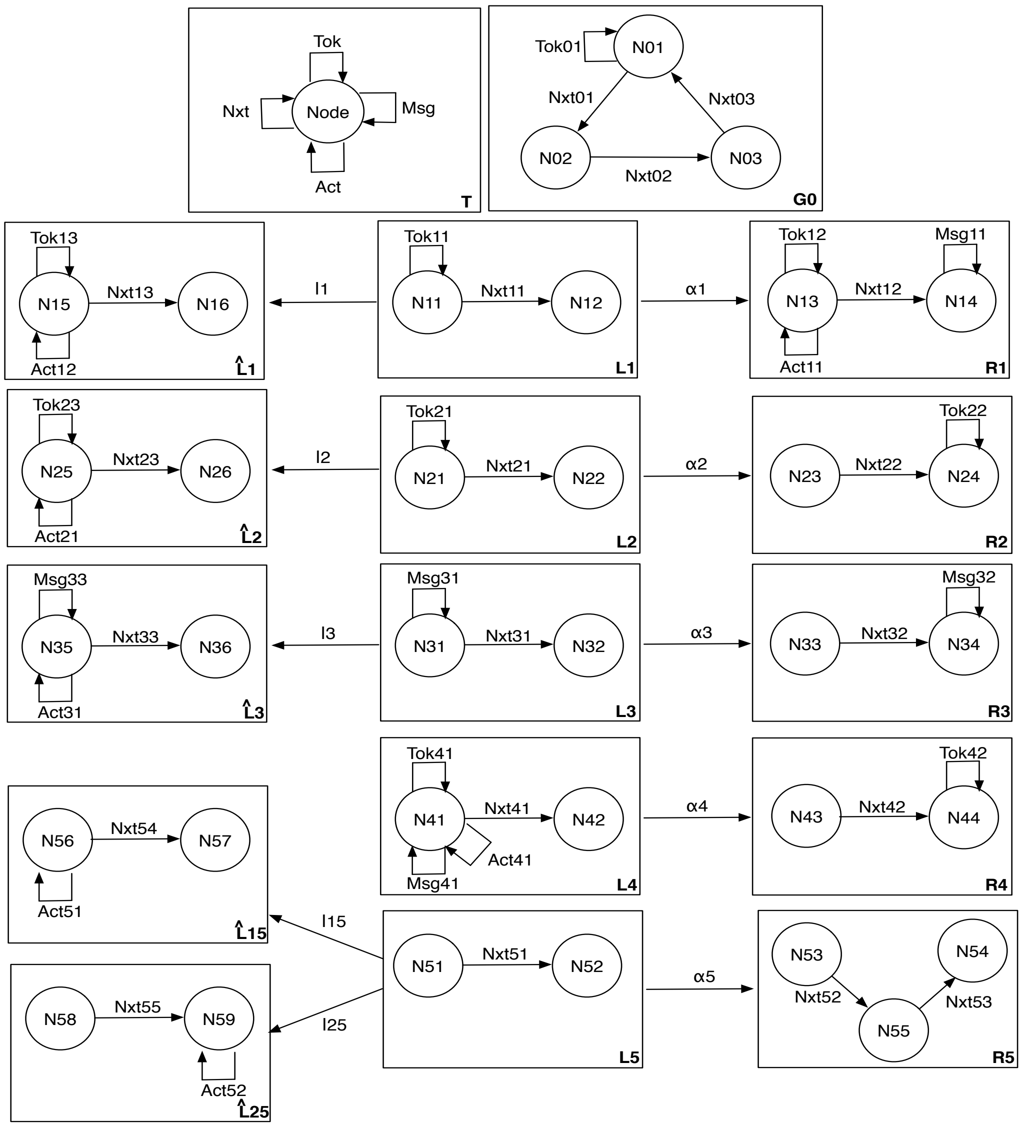
Next we detail the relational representation of graph grammars with NACs. The definition of relational rules with NACs just replaces the graph morphisms from Definition 32 by relational ones.

Definition 35 (Relational Rule with Negative Application Conditions)**.** *A **relational rule with negative application conditions (NACs)** is a pair $\hat{\alpha} = (\alpha, AN(\alpha))$ consisting of a relational rule $\alpha = \langle |L^T|, |\alpha|, |R^T| \rangle$ and a set of negative application conditions $AN(\alpha) \subseteq \mathcal{MOR}(|L^T|)$, where $\mathcal{MOR}(|L^T|)$ denotes the set of all total relational typed graph morphisms from the relational typed graph $|L^T|$ to relational typed graphs typed over $|T|$.*

Proposition 19. *A relational rule with negative application conditions is a well-defined rule with negative application conditions.*

Proof. By Proposition 6 the relational rule is well defined and by Proposition 5 the relational typed graph morphisms are well-defined. Also, all relational graph morphisms that define NACs are total. □

The extension of the relational representation of graph grammars with negative application conditions adds to the original definition (Def. 13) a tuple of relations for each relational rule which allows the specification of a set of NACs for the corresponding rule. Then, a relational graph grammar with negative application conditions is composed by a



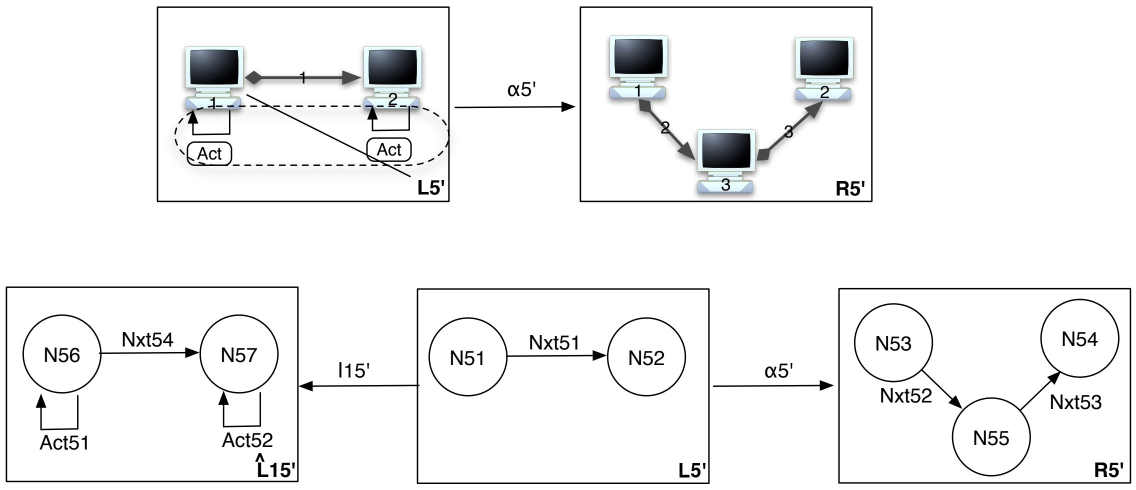
$$\alpha1\begin{cases}\alpha1_{Vert}(\text{N11}) = \text{N13}\\ \alpha1_{Vert}(\text{N12}) = \text{N14}\\ \alpha1_{Edge}(\text{Tok11}) = \text{Tok12}\\ \alpha1_{Edge}(\text{Nxt11}) = \text{Nxt12}\end{cases} \quad l1\begin{cases}l1_{Vert}(\text{N11}) = \text{N15}\\ l1_{Vert}(\text{N12}) = \text{N16}\\ l1_{Edge}(\text{Tok11}) = \text{Tok13}\\ l1_{Edge}(\text{Nxt11}) = \text{Nxt13}\end{cases}$$

$$\alpha2\begin{cases}\alpha2_{Vert}(\text{N21}) = \text{N23}\\ \alpha2_{Vert}(\text{N22}) = \text{N24}\\ \alpha2_{Edge}(\text{Nxt21}) = \text{Nxt22}\end{cases} \quad l2\begin{cases}l2_{Vert}(\text{N21}) = \text{N25}\\ l2_{Vert}(\text{N22}) = \text{N26}\\ l2_{Edge}(\text{Tok21}) = \text{Tok23}\\ l2_{Edge}(\text{Nxt21}) = \text{Nxt23}\end{cases}$$

$$\alpha3\begin{cases}\alpha3_{Vert}(\text{N31}) = \text{N33}\\ \alpha3_{Vert}(\text{N32}) = \text{N34}\\ \alpha3_{Edge}(\text{Nxt31}) = \text{Nxt32}\end{cases} \quad l3\begin{cases}l3_{Vert}(\text{N31}) = \text{N35}\\ l3_{Vert}(\text{N32}) = \text{N36}\\ l3_{Edge}(\text{Msg31}) = \text{Msg33}\\ l3_{Edge}(\text{Nxt31}) = \text{Nxt33}\end{cases} \quad \alpha4\begin{cases}\alpha4_{Vert}(\text{N41}) = \text{N43}\\ \alpha4_{Vert}(\text{N42}) = \text{N44}\\ \alpha4_{Edge}(\text{Nxt41}) = \text{Nxt42}\end{cases}$$

$$\alpha5\begin{cases}\alpha5_{Vert}(\text{N51}) = \text{N53}\\ \alpha5_{Vert}(\text{N52}) = \text{N54}\end{cases} \quad l15\begin{cases}l15_{Vert}(\text{N51}) = \text{N56}\\ l15_{Vert}(\text{N52}) = \text{N57}\\ l15_{Edge}(\text{Nxt51}) = \text{Nxt54}\end{cases} \quad l25\begin{cases}l25_{Vert}(\text{N51}) = \text{N58}\\ l25_{Vert}(\text{N52}) = \text{N59}\\ l25_{Edge}(\text{Nxt51}) = \text{Nxt55}\end{cases}$$

Figure 5.2: Alternative Definition of the Token Ring GG with NACs



$$\alpha 5' \begin{cases} \alpha 5'_{Vert}(\mathsf{N51}) = \mathsf{N53} \\ \alpha 5'_{Vert}(\mathsf{N52}) = \mathsf{N54} \end{cases} \qquad l15' \begin{cases} l15'_{Vert}(\mathsf{N51}) = \mathsf{N56} \\ l15'_{Vert}(\mathsf{N52}) = \mathsf{N57} \\ l15'_{Edge}(\mathsf{Nxt51}) = \mathsf{Nxt54} \end{cases}$$

Figure 5.3: Rule R5’

relational type graph, characterizing the types of vertices and edges allowed in a system, an *initial relational graph*, representing the initial state of a system and *a set of relational rules (possibly with negative application conditions)*, describing the possible state changes that can occur in a system.

Definition 36 (Relational Graph Grammar with NACs)**.** *Let* $\mathcal{R}_{GGN} = \{vert_T,\ inc_T,\ vert_{G0},\ inc_{G0},\ t_V^{G0},\ t_E^{G0},\ \Big(vert_{Li},\ inc_{Li},\ t_V^{Li},\ t_E^{Li}, vert_{Ri}, inc_{Ri},\ t_V^{Ri}, t_E^{Ri},\ \alpha_{i_V},\ \alpha_{i_E}, (vert_{\hat{L}ji}, inc_{\hat{L}ji},\ t_V^{\hat{L}ji},\ t_E^{\hat{L}ji}, l_{ji_V}, l_{ji_E})_{j\in\{1,\ldots,m\}}\Big)_{i\in\{1,\ldots,n\}}\}$ *be a set of relation symbols. A* ***relational graph grammar with negative application conditions*** *is a* $\mathcal{R}_{GGN}$*-structure* $|GGN| = \langle D_{GGN},\ (r)_{r\in\mathcal{R}_{GGN}}\rangle$ *where*

- $D_{GGN} = V_{GGN} \cup E_{GGN}$ *is the set of vertices and edges of the graph grammar, where:* $V_{GGN} \cap E_{GGN} = \varnothing$*,* $V_{GGN} = V_T \cup V_{G0} \cup \big(V_{Li} \cup V_{Ri} \cup (V_{\hat{L}ji})_{j\in\{1,\ldots,m\}}\big)_{i\in\{1,\ldots,n\}}$ *and* $E_{GGN} = E_T \cup E_{G0} \cup \big(E_{Li} \cup E_{Ri} \cup (E_{\hat{L}ji})_{j\in\{1,\ldots,m\}}\big)\big)_{i\in\{1,\ldots,n\}}$.

- $|T| = \langle V_T \cup E_T, \{vert_T, inc_T\}\rangle$ *defines a relational graph* ***(the type of the grammar)****.*

- $|G0^T| = \langle |G0|, |t^{G0}|, |T|\rangle$*, with* $|G0| = \langle V_{G0} \cup E_{G0}, \{vert_{G0}, inc_{G0}\}\rangle$ *and* $|t^{G0}| = \{t_V^{G0}, t_E^{G0}\}$*, defines a relational typed graph* ***(the initial graph of the grammar)****.*

- *Each collection* $\Big(vert_{Li},\ inc_{Li},\ t_V^{Li},\ t_E^{Li},\ vert_{Ri}, inc_{Ri},\ t_V^{Ri},\ t_E^{Ri},\ \alpha_{i_V},\ \alpha_{i_E}, (vert_{\hat{L}ji},\ inc_{\hat{L}ji}, t_V^{\hat{L}ji},\ t_E^{\hat{L}ji}, l_{ji_V}, l_{ji_E})_{j\in\{1,\ldots,m\}}\Big)$ *defines a* ***rule with negative application conditions****:*

  – $|Li^T| = \langle |Li|, |t^{Li}|, |T|\rangle$*, with* $|Li| = \langle V_{Li} \cup E_{Li}, \{vert_{Li}, inc_{Li}\}\rangle$ *and* $|t^{Li}| = \{t_V^{Li}, t_E^{Li}\}$*, defines a relational typed graph* ***(the left-hand side of the rule)****.*

  - *$|Ri^T| = \langle |Ri|, |t^{Ri}|, |T| \rangle$, with $|Ri| = \langle V_{Ri} \cup E_{Ri}, \{vert_{Ri}, inc_{Ri}\} \rangle$ and $|t^{Ri}| = \{t_V^{Ri}, t_E^{Ri}\}$, defines a relational typed graph **(the right-hand side of the rule)**.*
  - *$\langle |Li^T|, |\alpha_i|, |Ri^T| \rangle$, with $|\alpha_i| = \{\alpha_{i_V}, \alpha_{i_E}\}$, defines a relational rule.*
  - *each collection $(vert_{\hat{L}ji}, inc_{\hat{L}ji}, t_V^{\hat{L}ji},\ t_E^{\hat{L}ji}, l_{ji_V}, l_{ji_E})$ defines a NAC **(a negative application condition)**:*
    * *$|\hat{L}ji^T| = \langle |\hat{L}ji|, |t^{\hat{L}ji}|, |T| \rangle$, with $|\hat{L}ji| = \langle V_{\hat{L}ji} \cup E_{\hat{L}ji}, \{vert_{\hat{L}ji}, inc_{\hat{L}ji}\} \rangle$ and $|t^{\hat{L}ji}| = \{t_V^{\hat{L}ji}, t_E^{\hat{L}ji}\}$, defines a relational typed graph.*
    * *$|lji| = \{lji_V, lji_E\}$ defines a total relational graph morphism from $|Li^T|$ to $|\hat{L}ji^T|$.*

In case that variable m is set to null, no negative application condition is associated to rule αi.

Proposition 20. *The relational structure $|GGN|$ is well-defined.*

Proof. Follows immediately from Propositions 1, 4, 6 and 19. □

A relational rule with negative application conditions is applicable to a state-graph if there is a relational match which satisfies all negative application conditions of the applied rule.

Definition 37 (Relational Match Satisfaction). *Let $\langle |L^T|, |\alpha|, |R^T| \rangle$ be a relational rule, and let $|G^T| = \langle |G|, |t^G|, |T| \rangle$ be a relational typed graph.*

*A **relational match** $|m| = \{m_V, m_E\}$ **of the given rule in** $|G^T|$ **satisfies a NAC** $|lj| = \{lj_V, lj_E\}$ from $|L^T|$ to $|\hat{L}j^T|$, if there is not a total injective relational graph morphism $n = \{n_V, n_E\}$ from $|\hat{L}j^T|$ to $|G^T|$ that satisfies the following condition:*

- ***NAC Satisfaction Condition** $n_V \circ l_{j_V} = m_V$ and $n_E \circ l_{j_E} = m_E$.*

*A **relational match satisfies all NACs of a rule** if it satisfies each individual NAC of the rule.*

The graph grammar obtained after a rule (with NACs) application can be also defined as a definable transduction with θ formulas as described in Definition 16. We have just to extend the φ formula to include the satisfaction of the relational match for all NACs of the applied rule. That is, φ must also express that for each NAC $|lji| = \{lji_V, lji_E\}$ from $|L^T|$ to $|\hat{L}ji^T|$ of the selected rule, there is no total injective relational graph morphism $|n| = \{n_V, n_E\}$ from $|\hat{L}ji^T|$ to $|G0^T|$ which satisfies the following conditions:

$$\forall v \in vert_{Li}\ [lji_V(v, x) \wedge m_V(v, y) \Rightarrow n_V(x, y)]$$

$$\forall e \in inc_{Li}\ [lji_E(e, x) \wedge m_E(e, y) \Rightarrow n_E(x, y)].$$

Next, for each NAC $|lji| = \{lji_V, lji_E\}$ from $|L^T|$ to $|\hat{L}ji^T|$ of a selected rule, we describe one way of finding total relational graph morphisms from $|\hat{L}ji^T|$ to $|G0^T|$ that satisfy the NAC satisfaction condition. This can be done in two main steps:

1. We must find total injective relational graph morphisms from $|\hat{L}ji^T|$ to $|G0^T|$;
2. We must attest the NAC satisfaction condition for each morphism found in the first step.

If at least one relational graph morphism is obtained after the second step, then the selected rule can not be applied to the selected match.

STEP 1: Defining total (injective) relational graph morphisms

Let $|lj| = \{lji_V, lji_E\}$ be a NAC from $|Li^T|$ to $|\hat{L}ij^T|$ of the selected rule αi and let $m = \{m_V, m_E\}$ be the given match of $|Li^T|$ in $|G0^T|$. We have to find an embedding, of $|\hat{L}ji^T|$ to the initial graph $|G0^T|$. This is a widely explored problem, known as the *subgraph homomorphism problem*. Some works like (SCHFüRR, 1997; EDELKAMP; JABBAR; LLUCH-LAFUENTE, 2006; GEISS et al., 2006) have shown that it is possible to reduce its average-case complexity.

Our approach makes use of the representation and solution proposed in (RUDOLF, 2000). In this work, Rudolf proposes to represent and solve the problem of graph matching as a constraint satisfaction problem (CSP). The advantage of such choice relays on the possibility of applying optimized solution algorithms (KUMAR, 1992; LECOUTRE, 2009) that have already been proposed for CSPs. Such representation has also been successfully applied in the implementation of the matching subsystem of the Attributed Graph Grammar System (ERMEL; RUDOLF; TAENTZER, 1999).

A *constraint satisfaction problem (CSP)* consists of:

- a finite set of variables $X = \{x_1, \ldots, x_n\}$;
- a finite and discrete domain D_k of possible values for every variable $x_k \in X$;
- a finite set of constraints on the variables of X; a *constraint* C_S on $S = (x_1, \ldots, x_r)$ is a relation $C_S \subseteq D_1 \times \ldots \times D_r$.

Any tuple $\Gamma = (a_1, \ldots, a_n), a_k \in D_k$ denotes an *instantiation* of a CSP. We write $\Gamma(x_k) = a_k$ for the value of x_k under Γ. A constraint C_S on $S = (x_1, \ldots, x_r)$ is *satisfied* by an instantiation Γ if $(\Gamma(x_1), \ldots, \Gamma(x_r)) \in C_S$. An instantiation is a *solution* for a CSP if it satisfies all constraints of the problem.

Given two graphs $|\hat{L}ji^T| = \langle|\hat{L}ji|, |t^{\hat{L}ji}|, |T|\rangle$, with $|\hat{L}ji| = \langle V_{\hat{L}ji} \cup E_{\hat{L}ji}, \{vert_{\hat{L}ji}, inc_{\hat{L}ji}\}\rangle$ and $|G0^T| = \langle|G0|, |t^{G0}|, |T|\rangle$, with $|G0^T| = \langle V_{G0} \cup E_{G0}, \{vert_{G0}, inc_{G0}\}\rangle$, we construct a CSP as follows:

- $X = V_{\hat{L}ji} \cup E_{\hat{L}ji} = \{x_1, \ldots, x_n\}, n = |X|$;
- $D_k = \begin{cases} V_{G0}, & \text{if } x_k \in V_{\hat{L}ji} \\ E_{G0}, & \text{otherwise} \end{cases}, k \in 1, \ldots, n$
- The set of constraints is built according to Table 5.1: whenever a condition listed in the left column of the table holds for a given pair of variables (x_k, x_l), the corresponding constraint is to be included in the set of constraints.

Any solution $\Gamma = (a_1, \ldots, a_n)$ of the resulting CSP defines a total relational graph morphism $|n| = \{n_V, n_E\}$ from $|\hat{L}ji^T|$ to $|G0^T|$ as follows:

$$n_V = \{(x_k, a_k)|x_k \in V_{\hat{L}ji} \wedge \Gamma(x_k) = a_k\}$$

$$n_E = \{(x_k, a_k)|x_k \in E_{\hat{L}ji} \wedge \Gamma(x_k) = a_k\}$$

Proposition 21. *Set $n = \{n_V, n_E\}$ specified above defines a total relational graph morphism from $|\hat{L}ji^T|$ to $|G0^T|$.*

Proof. Since each variable is attributed with only one element of the respective domain, both relations define partial functions. They are total because a value is instantiated to each variable (and all vertices and edges of $|\hat{L}ji^T|$ are considered as variables). D_k, $C^{vtype}_{x_k}$ and $C^{etype}_{x_k}$ specifications guarantee the type consistency conditions. Morphism commutativity conditions hold due to $C^{source}_{(x_k,x_l)}$ and $C^{target}_{(x_k,x_l)}$ constraints. □

If we restrict ourselves only to solutions that attribute different values to each variable, we have total injective relational graph morphisms.

Table 5.1: Construction of Constraints

Condition	**Constraint**	**Intuitive Meaning**
$x_k = x_l$, $x_k \in V_{\hat{L}ji}$	$C^{vtype}_{x_k} = \{d \in D_k \mid \exists t[t^{\hat{L}ji}_V(x_k, t) \wedge t^{G0}_V(d, t)]\}$	Values instantiated to a vertex-variable x_k must be of the same type of the variable. I.e., vertices types must be preserved.
$x_k = x_l$, $x_k \in E_{\hat{L}ji}$	$C^{etype}_{x_k} = \{d \in D_k \mid \exists t[t^{\hat{L}ji}_E(x_k, t) \wedge t^{G0}_E(d, t)]\}$	Values instantiated to an edge-variable x_k must be of the same type of the variable. I.e., edges types must be preserved.
$x_k \in E_{\hat{L}ji}$, $x_l \in V_{\hat{L}ji}$, $\exists y[inc_{\hat{L}ji}(x_k, x_l, y)]$	$C^{source}_{(x_k,x_l)} = \{(d_k, d_l) \in D_k \times D_l \mid \exists y[inc_{G0}(d_k, d_l, y)]\}$	If x_l is source of x_k, the value of x_l must be source of the value of x_k. I.e., sources must be preserved.
$x_k \in E_{\hat{L}ji}$, $x_l \in V_{\hat{L}ji}$, $\exists y[inc_{\hat{L}ji}(x_k, y, x_l)]$	$C^{target}_{(x_k,x_l)} = \{(d_k, d_l) \in D_k \times D_l \mid \exists y[inc_{G0}(d_k, y, d_l)]\}$	If x_l is target of x_k, the value of x_l must be target of the value of x_k. I.e., targets must be preserved.
$x_k = x_l$, $x_k \in V_{\hat{L}ji}$	$C^{vvalue}_{x_k} = \{d_k \in D_k \mid \exists v, d_k[lji_V(v, x_k) \wedge m_V(v, d_k)\}$	NAC satisfaction condition must be satisfied for vertices.
$x_k = x_l$, $x_k \in E_{\hat{L}ji}$	$C^{evalue}_{x_k} = \{d_k \in D_k \mid \exists v, d_k[lji_E(v, x_k) \wedge m_E(v, d_k)\}$	NAC satisfaction condition must be satisfied for edges.

STEP 2: Attesting the NAC satisfaction condition

Although the restrictions $C^{vvalue}_{x_k}$ and $C^{evalue}_{x_k}$ discard values of variables that would not satisfy the NAC satisfaction condition, they do not guarantee its satisfaction. For instance, if the NAC maps two vertices to the same vertex and if the given match maps them to different ones, the total injective relational graph morphism can be defined, satisfying the constraints, but not respecting the commutativity required in the NAC satisfaction condition. Thus, after defining a total injective relational graph morphism $|n| = \{n_V, n_E\}$ from $|\hat{L}ji^T|$ to $|G0^T|$, we still have to verify the following conditions:

$$\forall v \in vert_{Li}\ [lji_V(v,x) \wedge m_V(v,y) \Rightarrow n_V(x,y)]$$

$$\forall e \in inc_{Li}\ [lji_E(e,x) \wedge m_E(e,y) \Rightarrow n_E(x,y)].$$

5.4 Token Ring Protocol with NACs Verification

In the verification step, the existence of NACs determines extra conditions that can be used during the proofs. As an illustration, we prove the property that establishes that any reachable graph has at most one active station. First of all, we have to include in the standard library, a function that returns the number of edges of specific type in a reachable graph.

[Library function $card_e$: Cardinality of Specific Edges] The number of edges of type t in a reachable graph is recursively defined by:

$$carde_t\ G0 = \sharp\{x | \exists y, z[inc_{G0}(x,y,z)] \wedge t^{G0}_E(x,t)\} \tag{5.1a}$$

$$\begin{aligned} carde_t\ ap^{\alpha i}_m G = carde_t G - \sharp\{x | \exists y, z[inc_{Li}(x,y,z)] \wedge t^{Li}_E(x,t)\} + \\ + \sharp\{x | \exists y, z[inc_{Ri}(x,y,z)] \wedge t^{Ri}_E(x,t)\} \end{aligned} \tag{5.1b}$$

The number of edges of type t (or the number of t edges) for the initial graph is determined by the number of elements of type t (specified using t^{G0}_E) that belong to relation inc_{G0}. The number of t edges of a graph resulting from a rule application to graph G is designated by the number of t edges of G, less the number of t edges of the left-hand side of the rule plus the number of t edges of the right-hand side of the rule. ■

Now, we can state the following.

Property 4. *Any reachable graph has at most one edge of the type* Act.

According to the definition of $card_e$, previously defined, the property to be proven can be stated by the formula:

$$carde_{\mathsf{Act}} G \leq 1.$$

Proof.

Basis: The property is verified for the initial graph: function $carde_{\mathsf{Act}}$ of the stated property is instantiated for $G0$. Then, just by consulting the relations of the relational structure that defines the initial graph the property is trivially evaluated to true.

Hypothesis $\Rightarrow$ Inductive Step: Assuming that the property is valid for any reachable graph G, the proof requires 5 cases, depending on the considered rule. These cases can be grouped into 3 classes:

Case Class 1 (rules $\alpha1$, $\alpha2$): *A non-active station must hold the token.* In this case, just applying the function definition, the property can be violated because (by induction hypothesis) we consider the possibility of existing an active station in G. However, we would not have such case in this class. In fact, together with Property 1 (previously demonstrated for the case without NACs, which could be similarly proved for current specification) which establishes that "any reachable graph has exactly one edge of type Token", we can also establish a property that states that "the token is always in the active station, if there is one". And then, using these pre-established properties together with the condition determined by the NAC, which determines that we must have a token in an non-active station for the rules to be applied, it is possible to deduce that we do not have an active station in G. And then, just consulting the relations of left-hand side and right-hand side of the considered rule, the property is validated.

Case Class 2 (rule $\alpha4$): *An active station must hold the token.* For this proof we can use the following statement: there is an image using the match for all items that are in the left-hand side. That is, we have one active station in G. Then, consulting the relations of the relational structure, the property is validated.

Case Class 3 (rules $\alpha3$, $\alpha5$): *Other cases.* In this class, consulting the relations of left-hand side and right-hand side of the considered rules, it is possible to identify that there are no deleted, created or preserved Act edges. Then, the proof is completed by using the induction hypothesis.

$\square$

6 PATTERNS FOR PROPERTIES OVER REACHABLE STATES IN GRAPH GRAMMARS SPECIFICATIONS

Independently of the verification technique chosen to be applied, (semi-) automated verification involves the description of both the system and its desired properties in some formal specification language. The level of maturity and experience required to write these specifications is one of the first obstacles to the adoption of such techniques. Particularly, the specification of system requirements must be precise enough to support (semi-)automated validation and accessible enough to be stated by practitioners.

Until now, we focused on the (relational) description of systems. In this chapter, we propose patterns for the presentation, codification and reuse of property specifications. The patterns have the goal of helping and simplifying the task of stating precise requirements to be verified. Besides, it must prevent ambiguities and inaccuracies during the validation stage. Differently from most existing approaches (DWYER; AVRUNIN; CORBETT, 1999; CHECHIK; PAUN, 1999; SALAMAH et al., 2007) we focus on properties about reachable states for (infinite-)state verification. Most of existing patterns for property specification describe properties about traces for finite-state verification tools. These two approaches are complementary.

The patterns are based on functions that describe typical characteristics or elements of graphs (like the type of a vertex, the set of all edges of some type, the cardinality of vertices, etc.). In this chapter we will show how these functions can be defined in the framework of relational graph grammars. Since the relational approach is actually an encoding of algebraic (Single-Pushout - SPO) graph grammars (EHRIG et al., 1997), the property patterns presented are suitable for this class of graph grammars. To generalize to other classes, it would be necessary to provide corresponding encodings for the basic operation of rule application (this is what differentiates most graph grammar approaches).

Section 6.1 defines a standard library of functions to be used in the specifications. Section 6.2 describes our taxonomy and explain the patterns. Section 6.3 illustrates the use of the pattern system instantiating properties for a very simple mobile system. Section 6.4 discusses related works.

6.1 The Standard Library of Functions

The relational approach previously defined allowed the use of the technique of mathematical induction to prove properties about the internal states of systems specified as graph grammars.The properties were stated using pre-defined functions.

To create specification patterns for the characterization of systems requirements in this approach, we firstly define a standard library of these pre-defined functions. Most functions are specified for the reachable graph data type. This data type must be defined with two constructors, one for the initial graph G0 and another one for the operator $ap_m^{\alpha i}$ that applies the rule $|\alpha i|$ at match $|m| = \{m_V^{\alpha i}, m_E^{\alpha i}\}$ to a reachable graph. Auxiliary functions are defined to operate on graphs.

Table 6.1, Table 6.2 and Table 6.3 present some library functions. The library is not complete and must grow over as new specifications are required. Each function is recursively defined. Functions defined over the reachable graph data type are specified for the initial graph in the base case and for the graph resulting of a rule application in the inductive step. Depending on the description, functions return a set (as (1) to (4), (9) to (22) and (24) to (25)), a natural number (as (5) to (8)) or a Boolean (as (23) and (26)). Relations used in definitions are from the relational structure that characterizes the graph grammar. They are considered as axioms, that is, considering $|GG| = \langle D_{GG}, (R)_{R \in \mathcal{R}_{GG}} \rangle$ the relational structure associated to the grammar, we have $R(x_1, \ldots, x_n) \equiv true$ iff $(x_1, \ldots, x_n) \in R$.

This collection must help the developer not only in the properties specification but also in the construction of proofs. For instance, function (2) $vert_{t_1}$ returns the vertices of type t_1 of a reachable graph. Such vertices of the initial graph (i.e., $vert_{t_1}$ $G0$) are determined by relation t_V^{G0} of the relational structure. The vertices of type t_1 of the graph obtained from applying rule $|\alpha i|$ at match $|m|$ to a graph $|G|$ (i.e., $vert_{t_1}$ $ap_m^{\alpha i} G$) are either vertices of graph $|G|$ or vertices of the right-hand side of the rule that are not image of the rule.

The cardinality of edges function (6) returns the number of edges of a reachable graph. The number of edges of graph $G0$ is determined by the number of elements of relation inc_{G0}. The number of edges of a graph resulting from a rule application to graph G is designated by the number of edges of G, less the number of edges of the left-hand side of the rule plus the number of edges of the right-hand side of the rule. This function is well-defined because according to the definition of rule application (see θ specifications in Definition 16). The edges of the resulting graph are the edges of the right-hand side of the applied rule together with the edges of graph G that are not image of the match. Since the match is total and injective in the edge component, the number of decreased edges is the number of edges of the left-hand side of the applied rule.

6.2 Property Patterns

Now we define a collection of patterns for state properties specifications. Instead of specifying state properties just as forbidden or desired graphs as frequently done, we adopt logical formulas to describe them. As emphasized in (STRECKER, 2008), formulas over graph structure are more expressive than pattern graphs.

Patterns were developed to capture recurrent solutions to design and coding problems. According to Dwyer et al. (DWYER; AVRUNIN; CORBETT, 1999), through a pattern system, the specifier can identify similar requirements, select patterns that fit to those requirements and instantiate solutions that incorporate the patterns. In our approach, a set of relations characterizes the initial state and the possible behaviours of the system, and a definable transduction (that can be seen as an inference rule) describes the possible next states of the system. A state property specification pattern is a generalized description of a frequently occurring requirement on the admissible states of a system. It describes the essential arrangement of some aspect of the states of the system and provides expression of this arrangement.

We attempt to give a collection of independent patterns from which a set of interesting

Table 6.1: Standard Library

Ref.	**Description**	**Function Definition**	
(1)	Edges of	$edg_{t_1}\ G0$	$= \{x \mid t_E^{G0}(x, t_1)\}$
	specific type	$edg_{t_1}\ ap_m^{\alpha i} G$	$= \{x \mid t_E^{Ri}(x, t_1) \vee [x \in edg_{t_1} G\ \wedge \nexists w\ m_E^{\alpha i}(w, x)]\}$
(2)	Vertices of	$vert_{t_1}\ G0$	$= \{x \mid t_V^{G0}(x, t_1)\}$
	specific type	$vert_{t_1}\ ap_m^{\alpha i} G$	$= \{x \mid [t_V^{Ri}(x, t_1) \wedge\ \nexists w\ \alpha i_V(w, x)]\ \vee\ x \in vert_{t_1} G\}$
(3)	Edges with	$edg\ G0$	$= \{(x, y, z) \mid inc_{G0}(x, y, z)\}$
	source and	$edg\ ap_m^{\alpha i} G$	$= \{(x, y, z) \mid [(x, y, z) \in inc\ G \wedge \nexists w\ m_E^{\alpha i}(w, x)] \vee$
	target vertices		$\vee\ [\exists r, s\ inc_{Ri}(x, r, s) \wedge \exists w_1, w_2\ [\alpha i_V(w_1, r) \wedge$
			$\wedge\ \alpha i_V(w_2, s) \wedge m_V^{\alpha i}(w_1, y) \wedge m_V^{\alpha i}(w_2, z)]] \vee$
			$\vee\ [inc_{Ri}(x, y, z) \wedge \nexists w_1, w_2\ [\alpha i_V(w_1, y) \wedge$
			$\wedge\ \alpha i_V(w_2, z)]] \vee \exists r\ [inc_{Ri}(x, r, z) \wedge$
			$\wedge\ \exists w_1\ [\alpha i_V(w_1, r) \wedge m_V^{\alpha i}(r, y)] \wedge$
			$\wedge\ \nexists w_2\ \alpha i_V(w_2, z)] \vee \exists s\ [inc_{Ri}(x, y, s) \wedge$
			$\wedge\ \nexists w_1\ \alpha i_V(w_1, y) \wedge \exists w_2 [\alpha i_V(w_2, s) \wedge m_V^{\alpha i}(s, z)]]]\}$
(4)	Vertices	$vert\ G0$	$= \{x \mid vert_{G0}(x)\}$
		$vert\ ap_m^{\alpha i} G$	$= \{x \mid x \in vert\ G \vee [vert_{Ri}(x) \wedge \nexists w\ \alpha i_V(w, x)]\}$
(5)	Cardinality	$card_V\ G0$	$= \sharp vert_{G0}$
	of vertices	$card_V\ ap_m^{\alpha i} G$	$= card_V G + \sharp vert_{Ri} - \sharp vert_{Li}$
(6)	Cardinality	$card_E\ G0$	$= \sharp inc_{G0}$
	of edges	$card_E\ ap_m^{\alpha i} G$	$= card_E G - \sharp inc_{Li} + \sharp inc_{Ri}$
(7)	Cardinality	$cardv_{t_1}\ G0$	$= \sharp\{x \mid vert_{G0}(x) \wedge t_V^{G0}(x, t_1)\}$
	of specific	$cardv_{t_1}\ ap_m^{\alpha i} G$	$= cardv_{t_1} G + \sharp\{x \mid vert_{Ri}(x) \wedge t_V^{Ri}(x, t_1)\} -$
	vertices		$- \sharp\{x \mid vert_{Li}(x) \wedge t_V^{Li}(x, t_1)\}\}$
(8)	Cardinality	$carde_{t_1}\ G0$	$= \sharp\{x \mid \exists y, z[inc_{G0}(x, y, z)] \wedge t_E^{G0}(x, t_1)\}$
	of specific	$carde_{t_1}\ ap_m^{\alpha i} G$	$= carde_{t_1} G -$
	edges		$- \sharp\{x \mid \exists y, z[inc_{Li}(x, y, z)] \wedge t_E^{Li}(x, t_1)\} +$
			$+ \sharp\{x \mid \exists y, z[inc_{Ri}(x, y, z)] \wedge t_E^{Ri}(x, t_1)\}$
(9)	Pairs of loop	$ploop_{t_1, t_2}\ G0$	$= \{(e, f) \mid \exists x\ [inc_{G0}(e, x, x) \wedge inc_{G0}(f, x, x)] \wedge$
	edges of		$\wedge\ t_E^{G0}(e, t_1) \wedge t_E^{G0}(f, t_2)\}$
	specific types	$ploop_{t_1, t_2}\ ap_m^{\alpha i} G$	$= \{(e, f) \mid [\exists x\ [inc_{Ri}(e, x, x) \wedge inc_{Ri}(f, x, x)] \wedge$
	with source		$\wedge\ t_E^{Ri}(e, t_1) \wedge t_E^{Ri}(f, t_2)] \vee [(e, f) \in ploop_{t_1, t_2}\ G \wedge$
	and target in		$\wedge\ \nexists w\ m_E^{\alpha i}(w, e) \wedge \nexists w\ m_E^{\alpha i}(w, f)] \vee$
	the same vertex		$\vee\ [\exists x\ inc_G(e, x, x) \wedge t_E^G(e, t_1) \wedge$
			$\wedge\ \nexists w\ m_E^{\alpha i}(w, e) \wedge \exists y\ inc_{Ri}(f, y, y) \wedge$
			$\wedge\ t_E^{Ri}(f, t_2) \wedge \exists z[\alpha i_E(z, y) \wedge m_V^{\alpha i}(z, x)]] \vee$
			$\vee\ [\exists x\ inc_G(f, x, x) \wedge t_E^G(f, t_2) \wedge$
			$\wedge\ \nexists w\ m_E^{\alpha i}(w, f) \wedge \exists y\ inc_{Ri}(e, y, y)\ \wedge$
			$\wedge\ t_E^{Ri}(e, t_1) \wedge \exists z\ [\alpha i_E(z, y) \wedge m_V^{\alpha i}(z, x)]]$
(10)	Edges with	$edgs_{t_1}\ G0$	$= \{x \mid \exists y, z[inc_{G0}(x, y, z)] \wedge t_V^{G0}(y, t_1)\}$
	specific	$edgs_{t_1}\ ap_m^{\alpha i} G$	$= \{x \mid [x \in edgs_{t_1} G \wedge \nexists w\ m_E^{\alpha i}(w, x)] \vee$
	source		$\vee\ [\exists y, z[inc_{Ri}(x, y, z)] \wedge t_V^{Ri}(y, t_1)]\}$
(11)	Edges with	$edgt_{t_1}\ G0$	$= \{x \mid \exists y, z[inc_{G0}(x, y, z)] \wedge t_V^{G0}(z, t_1)\}$
	specific	$edgt_{t_1}\ ap_m^{\alpha i} G$	$= \{x \mid [x \in edgt_{t_1} G \wedge \nexists w\ m_E^{\alpha i}(w, x)] \vee$
	target		$\vee\ [\exists y, z[inc_{Ri}(x, y, z)] \wedge t_V^{Ri}(z, t_1)]\}$
(12)	Edges with	$edgl_{t_1, t_2}\ G0$	$= \{x \mid \exists y, z[inc_{G0}(x, y, z)] \wedge t_V^{G0}(y, t_1) \wedge t_V^{G0}(z, t_2)\}$
	specific source	$edgl_{t_1, t_2}\ ap_m^{\alpha i} G$	$= \{x \mid [x \in edgl_{t_1, t_2} G \wedge \nexists w\ m_E^{\alpha i}(w, x)] \vee$
	and target		$\vee\ [\exists y, z[inc_{Ri}(x, y, z)] \wedge t_V^{Ri}(y, t_1) \wedge t_V^{Ri}(z, t_2)]\}$
(13)	Loop	$loop\ G0$	$= \{x \mid \exists y\ inc_{G0}(x, y, y)\}$
	edges	$loop\ ap_m^{\alpha i} G$	$= \{x \mid \exists y\ inc_{Ri}(x, y, y) \vee [x \in loop\ G \wedge$
			$\wedge\ \nexists w\ m_E^{\alpha i}(w, x)]\}$

Table 6.2: Standard Library (Cont.)

Ref.	Description	Function Definition	
(14)	Attributes of edges	$att_E\ G0$	$= \{(x,a) \mid attr_{G0}(x,a)\}$
		$att_E\ ap_m^{\alpha i}G$	$= \{(x,a) \mid attr_{Ri}(x,a) \vee [(x,a) \in att_E\ G \wedge$ $\wedge \nexists w\ m_E^{\alpha i}(w,x)]\}$
(15)	Attributes of specific type	$att_{t_1}\ G0$	$= \{(x,a) \mid attr_{G0}(x,a) \wedge t_A^{G0}(x,t_1)\}$
		$att_{t_1}\ ap_m^{\alpha i}G$	$= \{(x,a) \mid [attr_{Ri}(x,a) \wedge t_A^{Ri}(x,t_1)] \vee$ $\vee [(x,a) \in att_{t_1}\ G \wedge \nexists w\ m_E^{\alpha i}(w,x)]\}$
(16)	Source vertices	$verto\ G0$	$= \{x \mid vert_{G0}(x) \wedge \exists y,z\ inc_{G0}(y,x,z) \wedge$ $\wedge \nexists y,z\ inc_{G0}(y,z,x)\}$
		$verto\ ap_m^{\alpha i}G$	$= \{x \mid [vert_{Ri}(x) \wedge \exists y,z\ [inc_{Ri}(y,x,z)] \wedge$ $\wedge \nexists w\ \alpha i_V(w,x) \wedge \nexists y,z\ [inc_{Ri}(y,z,x)]] \vee$ $\vee \Big[vert_G(x) \wedge [\exists w_1 m_V^{\alpha i}(w_1,x) \rightarrow$ $\rightarrow [\exists y,z\ inc_G(y,x,z) \wedge \nexists w\ m_E^{\alpha i}(w,y) \wedge$ $\wedge [\exists y,z\ inc_G(y,z,x) \rightarrow \exists w\ m_E^{\alpha i}(w,y)] \wedge$ $\wedge \exists w_2\ [\alpha i_V(w_1,w_2) \wedge \nexists y,z\ inc_{Ri}(y,z,w_2)]] \vee$ $\vee [\exists w_2\ \alpha i_V(w_1,w_2) \wedge \exists y,z\ inc_{Ri}(y,w_2,z) \wedge$ $\wedge \nexists y,z\ inc_{Ri}(y,z,w_2) \wedge [\exists y,z\ inc_G(y,z,x) \rightarrow$ $\rightarrow\ \exists w\ m_E^{\alpha i}(w,y)]]]\Big] \vee [vert_G(x) \wedge [\nexists w\ m_E^{\alpha i}(w,x) \rightarrow$ $\rightarrow \exists y,z\ inc_G(y,x,z) \wedge \nexists y,z\ inc_G(y,z,x)]]$
(17)	Sink vertices	$verti\ G0$	$= \{x \mid vert_{G0}(x) \wedge \exists y,z\ inc_{G0}(y,z,x) \wedge$ $\wedge \nexists y,z\ inc_{G0}(y,x,z)\}$
		$verti\ ap_m^{\alpha i}G$	$= \{x \mid [vert_{Ri}(x) \wedge \exists y,z\ [inc_{Ri}(y,z,x)] \wedge$ $\wedge \nexists w\ \alpha i_V(w,x) \wedge \nexists y,z\ [inc_{Ri}(y,x,z)]] \vee$ $\vee \Big[vert_G(x) \wedge [\exists w_1 m_V^{\alpha i}(w_1,x) \rightarrow$ $\rightarrow [\exists y,z\ inc_G(y,z,x) \wedge \nexists w\ m_E^{\alpha i}(w,y) \wedge$ $\wedge [\exists y,z\ inc_G(y,x,z) \rightarrow \exists w\ m_E^{\alpha i}(w,y)] \wedge$ $\wedge \exists w_2\ [\alpha i_V(w_1,w_2) \wedge \nexists y,z\ inc_{Ri}(y,w_2,z)]] \vee$ $\vee [\exists w_2\ \alpha i_V(w_1,w_2) \wedge \exists y,z\ inc_{Ri}(y,z,w_2) \wedge$ $\wedge \nexists y,z\ inc_{Ri}(y,w_2,z) \wedge [\exists y,z\ inc_G(y,x,z) \rightarrow$ $\rightarrow\ \exists w\ m_E^{\alpha i}(w,y)]]]\Big] \vee [vert_G(x) \wedge [\nexists w\ m_V^{\alpha i}(w,x) \rightarrow$ $\rightarrow \exists y,z\ inc_G(y,z,x) \wedge \nexists y,z\ inc_G(y,x,z)]]$
(18)	Isolated vertices	$ivert\ G0$	$= \{x \mid vert_{G0}(x) \wedge \nexists y,z\ [inc_{G0}(y,x,z) \vee$ $\vee inc_{G0}(y,z,x)]\}$
		$ivert\ ap_m^{\alpha i}G$	$= \{x \mid [vert_{Ri}(x) \wedge \nexists y,z\ [inc_{Ri}(y,z,x) \vee$ $\vee inc_{Ri}(y,x,z)] \wedge \nexists w\ \alpha i_V(w,x)] \vee$ $[vert_G(x) \wedge [\exists y,z\ [inc_G(y,z,x) \vee$ $\vee inc_G(y,x,z)] \rightarrow \exists w\ m_E^{\alpha i}(w,y)] \wedge$ $\wedge [\exists w\ m_V^{\alpha i}(w,x) \rightarrow \exists w_1\ [\alpha i_V(w,w_1)$ $\wedge \nexists y,z\ [inc_{Ri}(y,z,w_1) \vee inc_{Ri}(y,w_1,z)]]]]$
(19)	Vertices that are source of specific edges	$verts_{t_1}\ G0$	$= \{x \mid vert_{G0}(x) \wedge \exists y,z\ inc_{G0}(y,x,z) \wedge t_E^{G0}(y,t_1)\}$
		$verts_{t_1}\ ap_m^{\alpha i}G$	$= \{x \mid [vert_{Ri}(x) \wedge \nexists w\ \alpha i_V(w,x) \wedge$ $\wedge \exists y,z\ inc_{Ri}(y,x,z) \wedge t_E^{Ri}(y,t_1)] \vee$ $\vee [vert_G(x) \wedge \exists y,z\ inc_G(y,x,z) \wedge$ $\wedge t_E^G(y,t_1) \wedge \nexists w\ m_E^{\alpha i}(w,y)] \vee$ $\vee [vert_G(x) \wedge \exists w_1,w_2\ [m_V^{\alpha i}(w_1,x) \wedge$ $\wedge \alpha i_V(w_1,w_2) \wedge \exists y,z\ inc_{Ri}(y,w_2,z) \wedge t_E^{Ri}(y,t_1)]]$

Table 6.3: Standard Library (Cont.)

Ref.	Description	Function Definition
(20)	Vertices that are target of specific edges	$vertt_{t_1}\ G0 = \{x \mid vert_{G0}(x) \wedge \exists y, z\ inc_{G0}(y,z,x) \wedge t_E^{G0}(y,t_1)\}$
		$vertt_{t_1}\ ap_m^{\alpha i}G = \{x \mid [vert_{Ri}(x) \wedge \nexists w\ \alpha i_V(w,x) \wedge \wedge \exists y, z\ inc_{Ri}(y,z,x) \wedge t_E^{Ri}(y,t_1)] \vee \vee [vert_G(x) \wedge \exists y, z\ inc_G(y,z,x) \wedge \wedge t_E^G(y,t_1) \wedge \nexists w\ m_E^{\alpha i}(w,y)] \vee \vee [vert_G(x) \wedge \exists w_1, w_2\ [m_V^{\alpha i}(w_1,x) \wedge \wedge \alpha i_V(w_1,w_2) \wedge \exists y, z\ inc_{Ri}(y,z,w_2) \wedge \wedge t_E^{Ri}(y,t_1)]]$
(21)	Vertices that are reachable from a specific vertex	$rvert_v\ G0 = \{x \mid [x = v \wedge vert_{G0}(v)] \vee \exists y, z\ [y \in rvert_v\ G0 \wedge \wedge inc_{G0}(z,y,x)]\}$
		$rvert_v\ ap_m^{\alpha i}G = \{x \mid [x = v \wedge x \in vert\ ap_m^{\alpha i}G] \vee \vee \exists y, z\ [y \in rvert_v\ ap_m^{\alpha i}G \wedge (z,y,x) \in edg\ ap_m^{\alpha i}G]\}$
(22)	Transitive closure of t_1 edges in G	$tranc_{t_1}G = \{(x,y) \mid [inc_G(a,x,y) \wedge t_E^G(a,t_1)] \vee \vee [(x,z) \in tranc_{t_1}G \wedge (z,y) \in tranc_{t_1}G]$
(23)	Ring topology	$ring_{t_1}\ G0 = \forall x\ [vert_{G0}(x) \rightarrow tranc_{t_1}G0(x,x)] \wedge \wedge \forall a, b, x, y, z\ [inc_{G0}(a,x,y) \wedge t_E^{G0}(a,t_1) \wedge \wedge inc_{G0}(b,x,z) \wedge t_E^{G0}(b,t_1) \rightarrow a = b] \wedge \wedge \forall x, z\ [vert_{G0}(x) \wedge vert_{G0}(z) \rightarrow \rightarrow\ tranc_{t_1}G0(x,z)]$
		$ring_{t_1}\ ap_m^{\alpha i}G = \mathrm{ring}_{t_1}\ G \wedge \wedge \forall a, x, y, z, w\ [inc_{Li}(a,x,y) \wedge t_E^{Li}(a,t_1) \wedge \wedge \alpha i_V(x,z) \wedge \alpha i_V(y,w) \rightarrow tranc_{t_1}Ri(z,w)] \wedge \wedge \forall a, b, x, y, z\ [inc_{Ri}(a,x,y) \wedge t_E^{Ri}(a,t_1) \wedge inc_{Ri}(b,x,z) \wedge t_E^{Ri}(b,t_1) \rightarrow a = b]$
(24)	Root vertices in G	$root\ G = \{x \mid vert_G(x) \wedge \nexists y, z\ inc_G(y,z,x) \wedge \wedge \exists y, z\ inc_G(y,x,z)\}$
(25)	Reachable vertices from v in G	$reach_v\ G = \{v\} \cup \{x \mid \exists y, z\ [y \in reach_v\ G \wedge \wedge inc_G(z,y,x)]\}$
(26)	Tree topology	$tree\ G0 = \exists! x\ root\ G0(x) \wedge\ \forall x\ [\neg root\ G0(x) \rightarrow \rightarrow \exists! y, z\ inc_{G0}(y,z,x)] \wedge \nexists x, y\ inc_{G0}(x,y,y) \wedge \wedge \forall x, y, z, w\ [inc_{G0}(x,y,z) \wedge\ inc_{G0}(w,y,z) \rightarrow \rightarrow x = w] \wedge \forall x, y\ [vert_{G0}(x) \wedge root\ G0(y) \rightarrow \rightarrow reach_y\ G0(x)]$
		$tree\ ap_m^{\alpha i}G = \Big[\exists x_1, y_1, z_1\ [inc_{Li}(x_1,y_1,z_1) \wedge \nexists w\ \alpha i_E(x_1,w)] \rightarrow \rightarrow\ \exists x_2, y_2, z_2\ [inc_{Ri}(x_2,y_2,z_2) \wedge z_2 = \alpha i_V(z_1) \wedge \wedge \exists v\ reach_{\alpha i_V(v)}Ri(y_2)]\Big] \wedge \Big[\exists x_1, y_1, z_1[inc_{Ri}(x_1,y_1,z_1) \wedge \nexists w\ \alpha i_E(w,x_1)] \rightarrow \rightarrow\ [[\nexists w\ \alpha i_V(w,z_1) \wedge \exists v\ reach_{\alpha i_V(v)}Ri(y_1)] \vee \vee [\exists x_2, y_2, w_2[\alpha i_V(w_2,z_1) \wedge inc_{Li}(x_2,y_2,w_2) \wedge \wedge \nexists w_1\ \alpha i_E(x_2,w_1)] \wedge \exists v\ reach_{\alpha i_V(v)}Ri(y_1)] \vee \vee [\exists w_1, w_2\ [\alpha i_V(w_1,z_1) \wedge m_V^{\alpha i}(w_1,w_2) \wedge \wedge root\ G(w_2)] \wedge \nexists w\ \alpha i_V(w,y_1)]]\Big] \wedge \Big[[\exists x_1\ vert_{Ri}(x_1) \wedge \nexists w\ \alpha i_V(w,x_1)] \rightarrow [\exists y_1, z_1[inc_{Ri}(y_1,z_1,x_1) \wedge \nexists w\ \alpha i_E(w,y_1) \wedge \wedge \exists v\ reach_{\alpha i_V(v)}Ri(z_1)] \vee [\exists y_2, z_2[inc_{Ri}(y_2,x_1,z_2) \wedge \nexists w\ \alpha i_E(w,y_2) \wedge \exists w, w_2[\alpha i_V(w,z_2) \wedge m_V^{\alpha i}(w,w_2) \wedge root\ G(w_2)]]]]\Big]$

specifications about the internal state of the systems can be constructed. We do not intend to provide the smallest set of patterns that can generate all useful specifications nor a complete list of specifications. We indeed try to specify patterns, which must commonly appear as state property specifications and expect that this collection be expanded, as new property specifications do not match with the existing patterns.

Table 6.4: A Pattern Taxonomy

1. Functional	2. Structural
1.1 Resources	2.1 Topology
1.1.1 Absence	2.1.1 Absence
1.1.2 Existence	2.1.2 Existence
1.1.3 Universality	2.2 Adjacency
1.1.4 Cardinality	2.2.1 Absence
1.1.5 Dependence	2.2.2 Existence
1.2 Data	2.2.3 Universality
1.2.1 Absence	
1.2.2 Existence	
1.2.3 Universality	
1.2.4 Comparison	
1.2.5 Dependence	

The patterns must assist developers into the process of mapping descriptions of the states of the system into the formalism, allowing the specification of state properties without much expertise. To help the user in finding the appropriate pattern for each situation, we organized the patterns using the taxonomy in Table 6.4. We define three levels of hierarchy. The first level differentiates properties that express *functional* aspects of the system from properties that specify *structural* characteristics of the states. The functional pattern is divided in the second level according to the kind of information that it describes: the pattern *resources* deal with relations between vertices, edges (that do not describe attributes) and their types; the pattern *data* handle attribute edges. The structural pattern consider the arrangement between vertices and edges: in its second level, the *topology* pattern depicts the physical configuration of the states, determining how the vertices are connected, while the *adjacency* pattern treats the neighboring between vertices, edges and their types. The third level distinguishes, for each specificity, if the properties occur, do not occur or occur for all items of definite characteristics. The resource pattern still discriminates properties that deal with cardinality and dependence of specific items. The data pattern, besides dependence, also identifies properties that compare attributes. In the following, we briefly describe the formulas of the third level of the taxonomy:

Absence: state formulas specifying the non-occurrence of particular characteristics in all reachable states.

Existence: state formulas specifying the occurrence of particular characteristics in all reachable states.

Universality: state formulas specifying characteristics of all vertices or edges (possibly of some specific type) occurring in all reachable states.

Cardinality: state formulas specifying characteristics about the number of vertices or edges (possibly of some specific type) occurring in all reachable states.

Dependence: conditional state formulas occurring in all reachable states.

Comparison: state formulas specifying relations between attributes (possibly of specific types) in all reachable graphs.

Table 6.5 depicts (part of) the absence of resources pattern. A pattern consists of a name, a brief explanation of the pattern's intent, a list of properties mappings and descriptions of uses and purposes. For each stated property, we list the functions of the standard library used in the pattern. We do not express all patterns in full detail. Instead, in Table 6.6 we list the statement of some properties together with its classification. Also, in Table 6.7 we list another properties with its classification and the respective functions of the standard library that must be used in its assertion.

6.3 Specification of a Mobile System

We describe the use of the pattern system specifying a very simple mobile system. The system consists of a network of interconnected antennas and mobile users. Each user, connected to a single antenna, may start/finish a communication with another user. The user may be switched to another antenna. New antennas and users can be added to the system at any time.

Figure 6.1 illustrates the graph grammar for the example. The type graph T describes two types of nodes Ant (Antenna) and Usr (User) and three types of edges Acn (connection between antennas), Ucn (connection between users and antennas) and Cal (communication between users). The initial graph $\mathsf{G0}$, in Figure 6.1, specifies a system with two antennas and two users.

Rule $\alpha 1$ models the establishment of a communication between users. Rule $\alpha 2$ describes the introduction of a new antenna into the network. Rule $\alpha 3$ specifies the situation in which a user is switched to another antenna. Rule $\alpha 4$ express the end of communication between users. The inclusion of new users is depicted by rule $\alpha 5$ and the introduction of new links between existing antennas is delineated by rule $\alpha 6$.

The pattern system and the standard library previously presented can assist, for example, in the statement of the properties detailed in Table 6.8. Since the example does not involve attributes, properties of the Data pattern were not enunciated.

It is important to notice that in many cases the property to be stated will not be exactly the same property listed in the pattern, but instead it will be a composition of some described properties. For instance, the last two properties enunciated in Table 6.8 fit in this case. Nevertheless, as well as the other cases both the standard library and the patterns are very helpful to assist the developer in these specifications. In fact, the description of such properties must use functions of the standard library and, in most cases they must be instantiated through some inclusion, deletion or variation of the Boolean operators of one of the detailed patterns.

6.4 Related Work

Dwyer, Avrunin and Corbett (DWYER; AVRUNIN; CORBETT, 1998, 1999) introduced a specification pattern system for finite-state verification. The system is designed

Table 6.5: Absence of Resources Pattern

Absence of Resources Pattern

Express characteristics that are not allowed in all reachable states of the system.

Properties Mappings

Property	**Functions of Std. Library**	**Pattern**
There are no edges of type t_1 and t_2 simultaneously.	(1)	$\nexists x, y\ [x \in edg_{t_1} g \wedge y \in edg_{t_2} g]$
There are no vertices of type t_1 and t_2 simultaneously.	(2)	$\nexists x, y\ [x \in vert_{t_1} g \wedge y \in vert_{t_2} g]$
There are no edges of type t_1 and vertices of type t_2 simultaneously.	(1,2)	$\nexists x, y\ [x \in edg_{t_1} g \wedge y \in vert_{t_2} g]$
There are no pairs of loop edges of types t_1 and t_2 with source and target in the same vertex.	(9)	$\nexists (x, y)\ [(x, y) \in ploop_{t_1,t_2} g]$
There are no edges with source in a vertex of type t and edges with target in a vertex of type t simultaneously.	(10, 11)	$\nexists x, y\ [x \in edgs_t g \wedge y \in edgt_t g]$
There are no two edges with source and target of the same type.	(12)	$\forall t_1, t_2\ [\nexists x, y\ [x \in edgl_{t_1,t_2} \wedge y \in edgl_{t_1,t_2}]]$
There are no loop edges of types t_1 and t_2 simultaneously.	(1, 13)	$\nexists x, y\ [x \in loop\ g \wedge y \in loop\ g \wedge x \in edg_{t_1} g \wedge y \in edg_{t_2} g]$
There are no source vertices.	(16)	$\nexists x[x \in verto\ g]$
There are no source vertices of type t.	(2, 16)	$\nexists x[x \in verto\ g \wedge x \in vert_t g]$
There are no sink vertices.	(17)	$\nexists x[x \in verti\ g]$
There are no sink vertices of type t.	(2, 17)	$\nexists x[x \in verti\ g \wedge x \in vert_t g]$
There are no isolated vertices.	(18)	$\nexists x[x \in ivert\ g]$
There are no isolated vertices of type t.	(2, 18)	$\nexists x[x \in ivert\ g \wedge x \in vert_t g]$
There are no edges of type t and isolated vertices simultaneously.	(1, 18)	$\nexists x, y\ [x \in edg_t g \wedge y \in ivert\ g]$

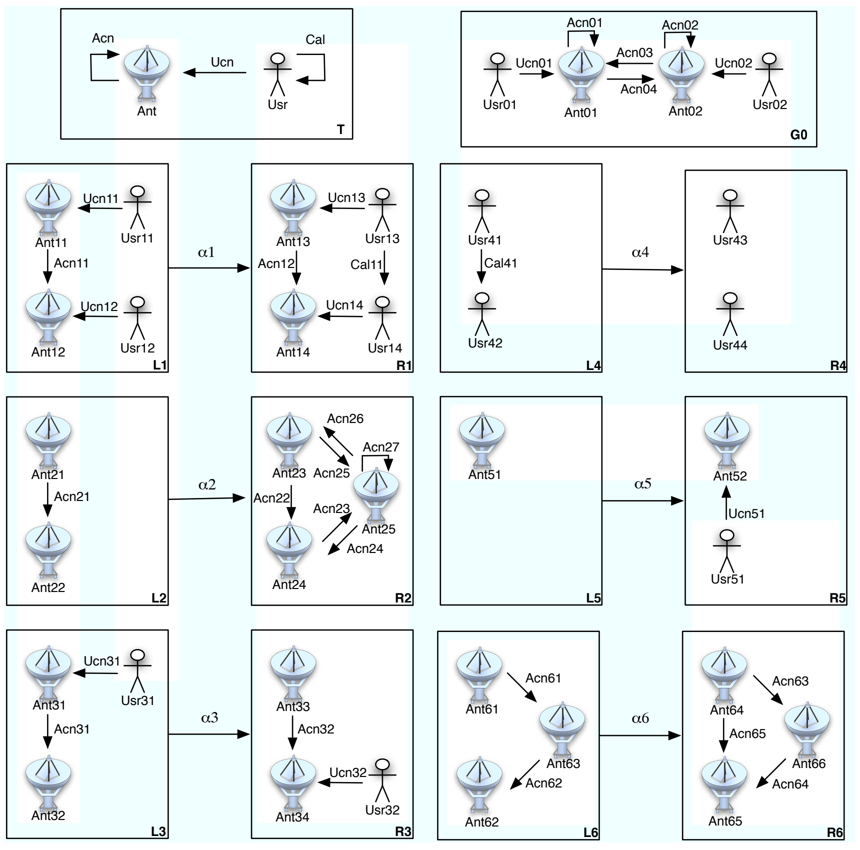
Uses and Purposes
This pattern can be applied to describe the impossibility of
specific actions and the inexistence of physical connections or physical resources.

Table 6.6: List of Properties

Property	**Functions of Std. Lib.**	**Pattern**	**Classif.**
There is an edge of type t.	(1)	$\exists x\ [x \in edg_t g]$	(1.1.2)
There is a vertex of type t_1 that is source of an edge of type t_2 and target of an edge of type t_3.	(2,19,20)	$\exists x\ [x \in vert_{t_1} g \wedge \wedge\ x \in verts_{t_2} g \wedge \wedge\ x \in vertt_{t_3} g]$	(1.1.2)
All vertices of type t_1 are source of edges of type t_2.	(2,19)	$\forall x\ [x \in vert_{t_1} g\ \rightarrow \rightarrow x \in verts_{t_2}\ g]$	(1.1.3)
All vertices of type t are not isolated.	(1,18)	$\forall x\ [x \in vert_t g\ \rightarrow \rightarrow x \notin ivert\ g]$	(1.1.3)
There are at least n vertices of type t.	(8)	$cardv_t g \geq n$	(1.1.4)
There is only one source vertex of type t.	(2, 16)	$\exists! x [x \in vert_t g \wedge \wedge\ x \in verto\ g]$	(1.1.4)
If there is an edge with source in a vertex of type t_1, then there is an edge with target in a vertex of type t_2.	(10,11)	$\exists x [x \in edgs_{t_1} g] \rightarrow \rightarrow \exists y [y \in edgt_{t_2} g]$	(1.1.5)
There are no attributes of type t_1 that are equal to attributes of type t_2.	(15)	$\forall x, y, a, b [(x, a) \in att_{t_1} g \wedge (y, b) \in att_{t_2} g \rightarrow a \neq b]$	(1.2.1)
There is an attribute of type t	(15)	$\exists x, a [(x, a) \in att_t g]$	(1.2.2)
All attributes of type t are less than n.	(15)	$\forall x, a [(x, a) \in att_t g \rightarrow \rightarrow a < n]$	(1.2.3)
All attributes of edges of type t_1 are always great than attributes of edges of type t_2, if they both have source and target in the same vertex.	(9,14)	$\forall x, y, a, b [(x, y) \in ploop_{t_1, t_2} g \wedge\ (x, a) \in att_E g \wedge\ (y, b) \in att_E g \rightarrow \rightarrow a > b]$	(1.2.4)
If there is an attribute of type t_1 then there is an attribute of type t_2.	(15)	$\exists x [x \in att_{t_1} g] \rightarrow \rightarrow \exists y [y \in att_{t_2} g]$	(1.2.5)
There is a tree topology.	(26)	$tree\ g \equiv true$	(2.1.2)
There is a vertex of type t_1 that is not reachable from any vertex of type t_2.	(2,21)	$\exists x [x \in vert_{t_1} g\ \wedge \forall y [y \in vert_{t_2} g \rightarrow x \notin rvert_y g]]$	(2.2.1)
There is a vertex that is target of an edge of type t_1 and source of an edge of type t_2.	(19,20)	$\exists x [x \in vertt_{t_1} g \wedge \wedge\ x \in verts_{t_2} g]$	(2.2.2)
All vertices of type t_1 are reachable from vertices of type t_2.	(2,21)	$\forall x, y [x \in vert_{t_1} g\ \wedge\ y \in vert_{t_2} g \rightarrow x \in rvert_y g]$	(2.2.3)

Table 6.7: List of Another Properties

Property	**Functions of Std. Library**	**Classif.**
There is a vertex of type t.	(2)	(1.1.2)
There is an edge of type t_1 and a vertex of type t_2.	(1,2)	(1.1.2)
All vertices of type t are not sink.	(1,17)	(1.1.3)
There is only one edge of type t.	(1)	(1.1.4)
There is only one vertex of type t.	(7)	(1.1.4)
There is only one vertex of type t_1 that is source of an edge of type t_2.	(2,19)	(1.1.4)
The number of edges is odd.	(6)	(1.1.4)
The number of edges is less than n.	(6)	(1.1.4)
The number of edges of type t is even.	(8)	(1.1.4)
There are at least n vertices.	(5)	(1.1.4)
If there is no edge of type t_1, then there is no edge of type t_2.	(1)	(1.1.5)
If there is an edge with source in a vertex of type t_1 and target in a vertex of type t_2, then there is a loop edge of type t_3.	(1,12,13)	(1.1.5)
If there is an edge of type t_1, then there is a vertex of type t_2.	(1,2)	(1.1.5)
If there is no vertex of type t_1, then there is no vertex of type t_2.	(2)	(1.1.5)
There is no attribute of type t that is great than n.	(15)	(1.2.1)
All attributes of type t_1 are less than attributes of type t_2.	(15)	(1.2.4)
There is not a ring topology of edges of type t.	(22,23)	(2.1.1)
There is not a tree topology.	(24,25,26)	(2.1.1)
There is a ring topology of edges of type t.	(22,23)	(2.1.2)
There is one or more isolated vertices.	(18)	(2.2.1)
There is an isolated vertex of type t.	(2,18)	(2.2.1)
There is a vertex of type t_1 that is reachable from a vertex of type t_2.	(2,21)	(2.2.2)
All isolated vertices are of type t	(2,18)	(2.2.3)
All vertices that are target of edges of type t_1 are source of edges of type t_2.	(19,20)	(2.2.3)



$$\alpha 1 \begin{cases} \alpha 1_V \begin{cases} (\text{Ant11}, \text{Ant13}) \\ (\text{Ant12}, \text{Ant14}) \\ (\text{Usr11}, \text{Usr13}) \\ (\text{Usr12}, \text{Usr14}) \end{cases} \\ \alpha 1_E \begin{cases} (\text{Acn11}, \text{Acn12}) \\ (\text{Ucn11}, \text{Ucn13}) \\ ((\text{Ucn12}, \text{Ucn14}) \end{cases} \end{cases} \qquad \alpha 2 \begin{cases} \alpha 2_V \begin{cases} (\text{Ant21}, \text{Ant23}) \\ (\text{Ant22}, \text{Ant24}) \end{cases} \\ \alpha 2_E \{ \ (\text{Acn21}, \text{Acn22}) \end{cases} \qquad \alpha 3 \begin{cases} \alpha 3_V \begin{cases} (\text{Ant31}, \text{Ant33}) \\ (\text{Ant32}, \text{Ant34}) \end{cases} \\ \alpha 3_E \{ \ (\text{Acn31}, \text{Acn32}) \end{cases}$$

$$\alpha 4 \left\{ \alpha 4_V \begin{cases} (\text{Usr41}, \text{Usr43}) \\ (\text{Usr42}, \text{Usr44}) \end{cases} \right. \qquad \alpha 5 \left\{ \alpha 5_V \left\{ \ (\text{Ant51}, \text{Ant52}) \right. \right. \qquad \alpha 6 \begin{cases} \alpha 6_V \begin{cases} (\text{Ant61}, \text{Ant64}) \\ (\text{Ant62}, \text{Ant65}) \\ (\text{Ant63}, \text{Ant66}) \end{cases} \\ \alpha 6_E \begin{cases} (\text{Acn61}, \text{Acn63}) \\ (\text{Acn62}, \text{Acn64}) \end{cases} \end{cases}$$

Figure 6.1: Mobile System Graph Grammar

to describe a portion (a scope) of a system's execution that is free or contains instances of certain events or states. It is organized in a hierarchy based on the kind of system behaviour they describe. Inside each pattern the properties are divided by scope and it is provided mappings to five formalisms - LTL, CTL, Graphical Interval Logic, Quantified Regular Expressions and INCA query language - which are input languages of finite-state verification tools, such as SPIN (HOLZMANN, 1997b), SMV (MCMILLAN, 1992) and many others. Their intent has been to capture the knowledge of experts in formal methods to assist practitioners in the task of writing their properties.

Many other researchers have used patterns to the specification of properties for finite-state verification. For instance, in (SMITH et al., 2002; COBLEIGH; AVRUNIN; CLARKE, 2006), Cobleigh and her co-authors have proposed templates using disciplined natural language, finite state automata and question tree to construct the patterns described in Dwyer et al. Corbett and his colleagues (CORBETT et al., 2000) used the same pattern system

Table 6.8: Properties Specification for the Mobile System

Description	**Property**	**Formula**	**Class.**
There are no antennas outside of the network.	There are no isolated vertices of type Ant.	$\nexists x[x \in ivert\ g \land x \in vert_{\mathsf{Ant}} g]$	1.1.1
There are no disconnected users.	There are no isolated vertices of type Usr.	$\nexists x[x \in ivert\ g \land x \in vert_{\mathsf{Usr}} g]$	1.1.1
Users are always connected to antennas.	All vertices of type Usr are source of edges of type Ucn.	$\forall x\ [x \in vert_{\mathsf{Usr}}\ g \rightarrow x \in verts_{\mathsf{Ucn}}\ g]$	1.1.3
It is always possible to make a call into the network.	There is an edge of type Acn.	$\exists x\ [x \in edg_{\mathsf{Acn}}\ g]$	1.1.2
There are at least two antennas into the network.	The number of vertices of type Ant is great or equal to 2.	$cardv_{\mathsf{Ant}}\ g \geq 2$	1.1.4
It is possible to establish a connection between each pair of antennas.	Each vertex of type Ant is reachable from any other vertex of type Ant.	$\forall x, y[x \in vert_{\mathsf{Ant}} g \land y \in vert_{\mathsf{Ant}} g \rightarrow y \in rvert_x g]$	2.2.3
Each antenna allows the start of a communication.	For each vertex of type Ant, there is at least one edge of type Acn with source in this vertex.	$\forall x[x \in vert_{\mathsf{Ant}} g \rightarrow \exists y, w[(y, x, w) \in edg\ g \land y \in edg_{\mathsf{Acn}} g]]$	1.1.3
If there are users in communication, then there is a connection between their antennas.	If there is an edge y of type Cal with source in a vertex $y1$ of type Usr and target in a vertex $y2$ of type Usr, then there is an edge $z1$ of type Ucn with source in $y1$ and target in $w1$, an edge $z2$ of type Ucn with source in $y2$ and target in $w2$ and an edge w of type Acn with source $w1$ and target $w2$.	$\exists y, y1, y2\ [y \in edg_{\mathsf{Cal}} g \land y1 \in vert_{\mathsf{Usr}} g \land y2 \in vert_{\mathsf{Usr}} g] \rightarrow \exists z1, z2, w1, w2, w\ [(z1, y1, w1) \in edg\ g \land z1 \in edg_{\mathsf{Ucn}} g \land (z2, y2, w2) \in edg\ g \land z2 \in edg_{\mathsf{Ucn}} g \land (w, w1, w2) \in edg\ g \land w \in edg_{\mathsf{Acn}}\ g]$	1.1.5

to provide a structured-English specification language. In (YANG; EVANS, 2004) , Yang and Evans also used pattern templates to infer temporal properties. Alternatively, the work of Jörges et al. (JöRGES; MARGARIA; STEFFEN, 2006) combines formula graphs with the pattern system for the specification of temporal properties. Bitsch (BITSCH, 2001) created a catalogue for the specification of safety properties.

Other researchers combine specification languages to extend the property patterns. Chechick and Paun (CHECHIK; PAUN, 1999; PAUN; CHECHIK, 1999) extend the pattern system of Dwyer et al. to reason about events. Drusinsky (DRUSINSKY, 2004) combine LTL with Harel statecharts to enable visual, logical and non-deterministic specifications. The Property Specification Tool (Prospec) (MONDRAGON; GATES, 2004; MONDRAGON et al., 2007) introduce composite propositions for specifying properties that include multiple events or conditions. And the work of Salamah et al. (SALAMAH et al., 2007) provide general templates for generating specifications in LTL for all pattern, scope and composite propositions combinations. Yu et al. (YU et al., 2006) also extend the Dwyer et al.'s pattern system with a logical composition of patterns to allow the specification of complex requirements.

Specification patterns for probabilistic and real-time models have also been delineated. The ProProST (GRUNSKE, 2008) pattern system can be used to formulate requirements in probabilistic logics and the real-time specification patterns described in (KONRAD; CHENG, 2005) support the formalization of properties in terms of real-time temporal logics. A similar system for the specification of real-time requirements can also be found in (GRUHN; LAUE, 2006). Letier et al. (LETIER; LAMSWEERDE, 2002) introduced operationalization patterns to specify typical occurring goals that include real-time information. Besides, Flake et al. (FLAKE; MUELLER, 2000) have developed structured English sentences to help in the formal description of real-time properties.

A number of other works have investigated the processing of natural language specifications into formal logics. In (ALI, 1994) is proposed a logical language designated for natural language processing. The Attempto project (FUCHS; SCHWERTEL; SCHWITTER, 1998) translates a subset of standard English language into a syntactic variant of first-order logic and offers a tool to support automatic reasoning. Similarly, the Circe project (AMBRIOLA; GERVASI, 2006) leads with the translation of natural language properties into propositional logic. Furthermore, a tool to identify and analyse logical inconsistencies in natural language requirements is proposed in (GERVASI; ZOWGHI, 2005).

Differently from previous pattern systems, our proposal makes use of the theorem proving technique (RUSHBY, 2001), which allows us to deal with the verification of both finite and infinite systems. The focus of the work, until now, has been to treat internal properties that are valid for all reachable states of (infinite-state) systems specified as graph grammars. Our intent has been to provide a simple way of stating properties about the arrangement of the internal states. For this reason, together with the definition of the standard library of functions, the pattern has the purpose of offering several possible direct instantiations of properties over states or simply of guiding the developer of which functions must be used in the specifications. We have used first-order logic as the underlying language of specification, whereas natural language has been used to describe informally what the property is designated to assert. We believe that our pattern system complements the existing approaches and provides the first steps in the direction of a pattern for infinite-state verification through graph grammars.

7 THEOREM PROVING GRAPH GRAMMARS USING EVENT-B

In this chapter we use Event-B to analyse properties of graph grammars. Event-B (DEPLOY, 2010) is a state-based formal method closely related to Classical B (ABRIAL, 1996). It has been successfully used in several applications, having available tool support for both model specification and analysis. Due to the similarity between Event-B models and graph grammar specifications, specially concerning the rule-based behaviour, we propose to translate graph grammar specifications in Event-B structures, such that it is possible to use the Event-B provers to demonstrate properties of a graph grammar. This translation is based on the relational definition of graph grammars. Up to now, we restrict ourselves only to graph grammars without attributes or negative application conditions.

The chapter is organized as follows. Section 7.1 briefly introduces the Event-B formalism and Section 7.2 shows how a graph grammar can be translated into an Event-B program.

7.1 Event-B

Event-B (DEPLOY, 2010) is a state-based formalism closely related to Classical B (ABRIAL, 1996) and Action Systems (BACK; SERE, 1989).

Definition 38 (Event-B Model, Event)**.** *An Event-B Model is defined by a tuple* $EBModel = (c, s, P, v, I, R_I, E)$ *where* c *are constants and* s *are sets known in the model;* v *are the model variables*[1]*;* $P(c, s)$ *is a collection of axioms constraining* c *and* s*;* $I(c, s, v)$ *is a model invariant limiting the possible states of* v *s.t.* $\exists c, s, v \cdot P(c, s) \wedge I(c, s, v)$ *- i.e.* P *and* I *characterise a non-empty set of model states;* $R_I(c, s, v')$ *is an initialization action computing initial values for the model variables; and* E *is a set of model* events.

Given states v, v' *an event is a tuple* $e = (H, S)$ *where* $H(c, s, v)$ *is the guard and* $S(c, s, v, v')$ *is the before-after predicate that defines a relation between current and next states. We also denote an event guard by* $H(v)$*, the before-after predicate by* $S(v, v')$ *and the initialization action by* $R_I(v')$*.*

An Event-B model is assembled from two parts, a *context* which defines the triple (c, s, P) and a *machine* which defines the other elements (v, I, R_I, E).

Model correctness is demonstrated by generating and discharging a collection of proof obligations. The model *consistency* condition states that whenever an event or an initialization action is attempted, there exists a suitable new state v' such that the model in-

[1]For convenience, as in (ABRIAL, 1996), no distinction is made between a set of variables and a state of a system.

variant is maintained - $I(v')$. This is usually stated as two separate proof obligations: a feasibility ($I(v) \wedge H(v) \Rightarrow \exists v' \cdot S(v, v')$) and an invariant satisfaction obligation ($I(v) \wedge H(v) \wedge S(v, v') \Rightarrow I(v')$). The behaviour of an Event-B model is the transition system defined as follows.

Definition 39 (Event-B Model Behaviour)**.** *Given* $EBModel = (c, s, P, v, I, R_I, E)$*, its behaviour is given by a transition system* $BST = (BState, BS_0, \rightarrow)$ *where:* $BState = \{\langle v\rangle | v \text{ is a } state\} \cup Undef$*,* $BS_0 = Undef$*, and* $\rightarrow \subseteq BState \times BState$ *is the transition relation given by the rules:*

$$\boxed{\textsf{start}}\,\frac{R_I(v') \wedge I(v')}{Undef \rightarrow \langle v'\rangle}$$

$$\boxed{\textsf{transition}}\,\frac{\exists (H, S) \in E \cdot I(v) \wedge H(v) \wedge S(v, v') \wedge I(v')}{\langle v\rangle \rightarrow \langle v'\rangle}$$

According to rule $start$ the model is initialized to a state satisfying $R_I \wedge I$ and then, as long as there is an enabled event (rule $transition$), the model may evolve by firing an enabled event and computing the next state according to the event's before-after predicate. Events are atomic. In case there is more than one enabled event at a certain state, the choice is non-deterministic. The semantics of an Event-B model is given in the form of proof semantics, based on Dijkstra's work on weakest preconditions (DIJKSTRA, 1976).

An extensive tool support through the Rodin Platform makes Event-B especially attractive (DEPLOY, 2010; ABRIAL et al., 2010). An integrated Eclipse-based development environment is actively developed, and open to third-party extensions in the form of Eclipse plug-ins. The main verification technique is theorem proving supported by a collection of theorem provers, but there is also some support for model checking. The support offered for theorem proving through the platform allows one to: browse the proof structure; select hypotheses and lemmas to be used; invoke different provers integrated to the platform; define and select tactics to be used; among others (ABRIAL et al., 2010).

7.2 Graph Grammars in Event-B

The behaviour of an Event-B model is similar to a graph grammar: there is a notion of state (given by a set of variables in Event-B, and by a graph in a graph grammar) and a step is defined by an atomic operation on the current state (an event that updates variables in Event-B and a rule application in a graph grammar). Each step should preserve properties of the state. In Event-B, these properties are stated as invariants. In a graph grammar, the properties that are guaranteed to be preserved are related to the graph structure (only well-formed graphs can be generated).

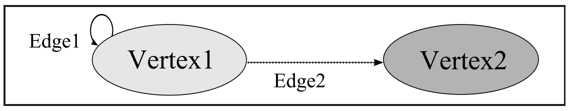
Now, we first present an overview of the translation of a graph grammar GG in an event-B model, and then explain in more details how each component is transformed. The translation is based on the relational graph grammar $|GG|$ corresponding to GG. Assume that GG has n rules, named α_1 to α_n and $i \in \{1, \ldots, n\}$. The event-B components describing this graph grammar are:

- The *sets* known in the model are $vert_T$, $edge_T$ (the sets of vertices and edges of the type graph T), $vert_{Li}$, $edge_{Li}$, $vert_{Ri}$ and $edge_{Ri}$ (the sets of vertices and edges of the left and right-hand side of each rule i) .
- The *constants* include vertices and edges of the type graph and rules, as well as (names of) typing functions t_V^{Li}, t_E^{Li}, t_V^{Ri} and t_E^{Ri}, source and target functions $source_T$,

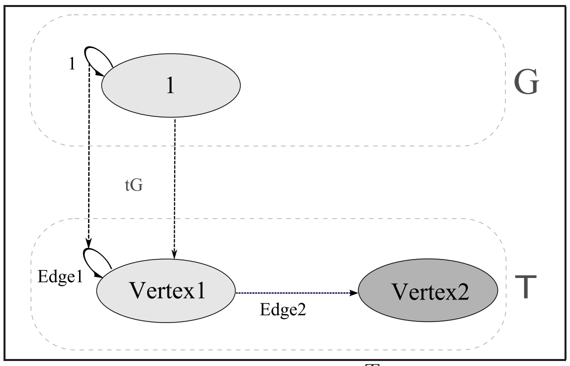
$target_T$, $source_{Li}$, $target_{Li}$, $source_{Ri}$ and $target_{Ri}$ and relational rules α_{i_V} and α_{i_E}.

- The *axioms* define explicitly all sets and functions of the model (whose names were declared above). The types of functions are also declared as axioms.
- The *model variables* are specified by relations $vert_G$, $edge_G$, $source_G$, $target_G$[2], t_V^G and t_E^G. They define a state of the system as a reachable graph G typed over T (see Def. 10).
- The *invariants* are used to define the types of variables.
- The *initialization action* defines the initial values for the variables $vert_G$, $edge_G$, $source_G$, $target_G$, t_V^G and t_E^G. It specifies the initial graph $G0$.
- The set of *events* models rule applications. An event is defined for each rule of the grammar. The *guard* guarantees the existence of a match of the left-hand side of the corresponding rule in a state-graph of the grammar. The *before-after* predicate is defined by a parallel assignment (to the variables that model the current state graph) and implements the formulas in Def. 16.

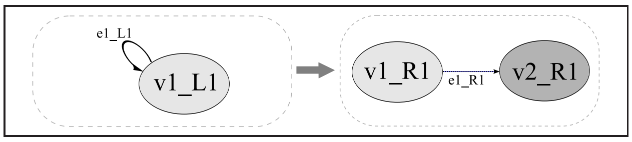
Now, we present this translation in more details. We will use a simple example, depicted in Figure 7.1, to illustrate how graphs, typed graphs and rules can be translated to Event-B components.



(a) Type Graph T



(b) Start Graph G^T



(c) Rule $\alpha 1$

Figure 7.1: Example of Graph Grammar

Graphs: According to Def. 7 and Def. 13, sets V_{GG} and E_{GG} contain all possible vertices and edge names that may appear in graphs of this relational structure. We will define these sets as:

[2]Relations $edge_G$ (unary), $source_G$ (binary) and $target_G$ (binary) are an alternative representation for inc_G (ternary). We have $(x, y, z) \in inc_G$ iff $x \in edge_G \wedge (x, y) \in source_G \wedge (x, z) \in target_G$.

$V_{GG} = vert_T \cup \mathbb{N}$, where $vert_T$ is the set of names used as vertex types in GG (we assume that $vert_T \cap \mathbb{N} = \varnothing$);

$E_{GG} = edge_T \cup \mathbb{N}$, where $edge_T$ is the set of names used as edge types in GG (we assume that $edge_T \cap \mathbb{N} = \varnothing$).

Moreover, we assume that $vert_T \cap edge_T = \varnothing$.

The type graph T is defined in an event-B context as described in Figure 7.2, where we define all vertex and edge types as constants, as well as the incidence relation relating them. In the axioms, we define these sets explicitly (for example, axiom $axm1$ means that $vertT = \{Vertex1, Vertex2\}$). Text after a $//$ is a comment.

CONTEXT ctx_GG
SETS
```
    vertT    // (Type Graph ) Vertices
    edgeT    // (Type Graph ) Edges
```
CONSTANTS
```
    Vertex1 Vertex2
    Edge1 Edge2
    incT
```
AXIOMS

axm1 : $partition(vertT, \{Vertex1\}, \{Vertex2\})$

axm2 : $partition(edgeT, \{Edge1\}, \{Edge2\})$

axm3 : $incT \subseteq (edgeT \times vertT \times vertT)$

axm4 : $partition(incT, \{Edge1 \mapsto Vertex1 \mapsto Vertex1\}, \{Edge2 \mapsto Vertex1 \mapsto Vertex2\})$

END

Figure 7.2: Event-B Type Graph

Instances of vertices and edges that appear in graphs representing states will be described by natural numbers. It is not necessary to have distinct numbers for vertices and edges: a graph may have a vertex with identity 1 as well as an edge with identity 1, these elements will be different because one will be mapped to a vertex type and the other to an edge type. To be able to manipulate instances easily, we define the functions $source$, $target$ and $edgeName$ (see Figure 7.3).

A graph typed over a type graph T (Def. 10) is modelled by a set of variables describing its set of vertices, incidence relation, and typing functions. It is possible to state the type consistency and morphism commutativity conditions (stated in Def. 8) as invariants. However, since we will always generate well-formed graphs (the start graph is well-formed and events implement the single-pushout construction), we will skip these invariants (each invariant that is used generates proof obligations and therefore it is advisable to use only the necessary ones).

Figure 7.4 shows the definition of a graph G typed over T. Invariants are used to define the types of the variables (for example, tG_V is a total function from $vertG$ to $vertT$ and tG_E is a partial function from the set of natural numbers to $edgeT$).

There is a special event in an event-B model that is executed before any other. This is the initialization event. In our encoding, this event will be used to create the start graph of a graph grammar. This is done by assigning initial values in the variables

CONSTANTS

  source
  target
  edgeName

AXIOMS

  axm5 : $\mathit{source} \in (\mathbb{N} \times \mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$
  axm6 : $\forall a, b, c \cdot a \in \mathbb{N} \land b \in \mathbb{N} \land c \in \mathbb{N} \Rightarrow \mathit{source}(a \mapsto b \mapsto c) = b$
  axm7 : $\mathit{target} \in (\mathbb{N} \times \mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$
  axm8 : $\forall a, b, c \cdot a \in \mathbb{N} \land b \in \mathbb{N} \land c \in \mathbb{N} \Rightarrow \mathit{target}(a \mapsto b \mapsto c) = c$
  axm9 : $\mathit{edgeName} \in (\mathbb{N} \times \mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$
  axm10 : $\forall a, b, c \cdot a \in \mathbb{N} \land b \in \mathbb{N} \land c \in \mathbb{N} \Rightarrow \mathit{edgeName}(a \mapsto b \mapsto c) = a$

END

Figure 7.3: Auxiliary Functions

MACHINE mch_GG
SEES ctx_GG
VARIABLES

  vertG // (Graph) Vertices
  incG // (Graph) Edges
  tG_V // Typing of vertices
  tG_E // Typing of edges

INVARIANTS

  inv_vertG : $\mathit{vertG} \in \mathbb{P}(\mathbb{N})$
  inv_incG : $\mathit{incG} \in \mathbb{P}(\mathbb{N} \times \mathbb{N} \times \mathbb{N})$
  inv_tG_V : $\mathit{tG\_V} \in \mathit{vertG} \rightarrow \mathit{vertT}$
  inv_tG_E : $\mathit{tG\_E} \in \mathbb{N} \nrightarrow \mathit{edgeT}$

EVENTS

Initialisation

  begin
    act1 : $\mathit{vertG} := \{1\}$
    act2 : $\mathit{incG} := \{1 \mapsto 1 \mapsto 1\}$
    act3 : $\mathit{tG\_V} := \{1 \mapsto \mathit{Vertex1}\}$
    act4 : $\mathit{tG\_E} := \{1 \mapsto \mathit{Edge1}\}$
  end

Figure 7.4: Event-B Graph G

that correspond to graph G (see Figure 7.4) depicted in Figure 7.1. In an event, there is no notion of order in the attributions belonging to the same event. A triple $(a, b, c) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ is denoted by $a \mapsto b \mapsto c$ in event-B.

Rules: Left- and right-hand sides of rules are graphs, and thus will have representations as defined previously. Additionally, we have to define the partial morphism (α_V, α_E) that maps elements from the left- to the right-hand side of the rule (Def. 12). The Event-B enconding of rule $\alpha 1$ depicted in Figure 7.1 is shown in Figure 7.5. Since rules do not change during execution, their structures are defined as constants.

It is important to notice that some axioms and invariants listed in an event-B specification guarantee the logical conditions imposed in the relational definitions. For

SETS

vertL1
edgeL1
vertR1
edgeR1

CONSTANTS

v1_L1 // vertex of LHS
e1_L1 // edge of LHS
v1_R1 // vertex of RHS
v2_R1 // vertex of RHS
e1_R1 // edge of RHS
sourceL1
targetL1
edgeNameL1
incL1
tL1_V // (Rule 1) Typing vertices of LHS
tL1_E // (Rule 1) Typing edges of LHS
incR1
tR1_V // (Rule 1) Typing vertices of RHS
tR1_E // (Rule 1) Typing edges of RHS
alpha1V // (Rule 1) Rule morphism: mapping vertices
alpha1E // (Rule 1) Rule morphism: mapping edges

AXIOMS

axm11 : $partition(vertL1, \{v1\_L1\})$
axm12 : $partition(edgeL1, \{e1\_L1\})$
axm13 : $incL1 \subseteq (edgeL1 \times vertL1 \times vertL1)$
axm14 : $partition(incL1, \{e1\_L1 \mapsto v1\_L1 \mapsto v1\_L1\})$
axm15 : $tL1\_V \in vertL1 \rightarrow vertT$
axm16 : $partition(tL1\_V, \{v1\_L1 \mapsto Vertex1\})$
axm17 : $tL1\_E \in edgeL1 \rightarrow edgeT$
axm18 : $partition(tL1\_E, \{e1\_L1 \mapsto Edge1\})$
axm19 : $partition(vertR1, \{v1\_R1\}, \{v2\_R1\})$
axm20 : $partition(edgeR1, \{e1\_R1\})$
axm21 : $incR1 \subseteq (edgeR1 \times vertR1 \times vertR1)$
axm22 : $partition(incR1, \{e1\_R1 \mapsto v1\_R1 \mapsto v2\_R1\})$
axm23 : $tR1\_V \in vertR1 \rightarrow vertT$
axm24 : $partition(tR1\_V, \{v1\_R1 \mapsto Vertex1\}, \{v2\_R1 \mapsto Vertex2\})$
axm25 : $tR1\_E \in edgeR1 \rightarrow edgeT$
axm26 : $partition(tR1\_E, \{e1\_R1 \mapsto Edge2\})$
axm27 : $sourceL1 \in (edgeL1 \times vertL1 \times vertL1) \rightarrow vertL1$
axm28 : $\forall a, b, c \cdot a \in edgeL1 \land b \in vertL1 \land c \in vertL1 \Rightarrow sourceL1(a \mapsto b \mapsto c) = b$
axm29 : $targetL1 \in (edgeL1 \times vertL1 \times vertL1) \rightarrow vertL1$
axm30 : $\forall a, b, c \cdot a \in edgeL1 \land b \in vertL1 \land c \in vertL1 \Rightarrow targetL1(a \mapsto b \mapsto c) = c$
axm31 : $edgeNameL1 \in (edgeL1 \times vertL1 \times vertL1) \rightarrow edgeL1$
axm32 : $\forall a, b, c \cdot a \in edgeL1 \land b \in vertL1 \land c \in vertL1 \Rightarrow edgeNameL1(a \mapsto b \mapsto c) = a$
axm33 : $alpha1V \in vertL1 \rightarrow vertR1$
axm34 : $partition(alpha1V, \{v1\_L1 \mapsto v1\_R1\})$
axm35 : $alpha1E \in edgeL1 ⤔ edgeR1$
axm36 : $alpha1E = \varnothing$
axm37 : $dom(edgeNameL1) = incL1$
axm38 : $ran(edgeNameL1) = dom(tL1\_E)$

END

Figure 7.5: Event-B Rule Structure

instance, $tR1\_V$ (respect. $tR1\_E$) defines a total function that relates vertices (respect. edges) of $R1$ to vertices (respect. edges) of T. Axioms $\mathrm{axm37}$ and $\mathrm{axm38}$ are included for well-definedness of the edge type compatibility that must be guaranteed when finding a match (see guard $\mathrm{grd8}$ in Figure 7.6).

The behaviour of a rule (Def. 16) is described by an event (for the example, by event $rule1$ in Figure 7.6). Whenever there are concrete values for variables $mV, mE, newV, newE$ that satisfies the guard conditions, the event may occur. Guard conditions $\mathrm{grd1}$, $\mathrm{grd2}$ and $\mathrm{grd7}$ to $\mathrm{grd9}$ assure that this pair is actually a match from the left-hand side of the rule to graph G (see Def. 15). Guard conditions $\mathrm{grd3}$ and $\mathrm{grd4}$ ensure that $newV$ and $newE$ are new fresh elements in the graph. Remaining guard conditions ($\mathrm{grd5}$ and $\mathrm{grd6}$) guarantee the well-definedness of the action that update the set tG_E. The actions update the state graph (graph G) according to the rule. In this example one loop edge is deleted and a vertex and a new edge are created. A vertex $newV$ is created with type $Vertex2$, and an edge $newE$ with type $Edge2$ is also created. The source of this new edge is the image of the only vertex in the left-hand side of the rule in G and the target is the newly created vertex.

```
EVENTS
Event  rule1 $\widehat{=}$
      any
            $mV$
            $mE$
            $newV$
            $newE$
      where
            grd1 : $mV \in vertL1 \rightarrow vertG$      // total on vertices
            grd2 : $mE \in incL1 \rightarrowtail incG$      // total and injective on edges
            grd3 : $newV \in \mathbb{N} \setminus vertG$      // newV is a fresh vertex name
            grd4 : $newE \in (\mathbb{N} \setminus \{x | x \in \mathbb{N} \land (\exists y, z \cdot y \in \mathbb{N} \land z \in \mathbb{N} \land (x \mapsto y \mapsto z) \in incG)\}) \setminus$
                  $dom(tG\_E)$      // newE is a fresh edge name
            grd5 : $ran(mE) \subseteq dom(edgeName)$      // well-definedness of act6
            grd6 : $ran(edgeName \circ mE) \subseteq dom(tG\_E)$      // well-definedness of act6
            grd7 : $\forall v \cdot v \in vertL1 \Rightarrow tL1\_V(v) = tG\_V(mV(v))$
                  // vertex type compatibility
            grd8 : $\forall e \cdot e \in incL1 \Rightarrow tL1\_E(edgeNameL1(e)) = tG\_E(edgeName(mE(e)))$
                  // edge type compatibility
            grd9 : $\forall e \cdot e \in incL1 \Rightarrow mV(sourceL1(e)) = source(mE(e)) \land mV(targetL1(e)) =$
                  $target(mE(e))$
                  // source/target compatibility
      then
            act3 : $vertG := vertG \cup \{newV\}$
            act4 : $incG := \{newE \mapsto source(mE(e1\_L1 \mapsto v1\_L1 \mapsto v1\_L1)) \mapsto newV\} \cup (incG \setminus$
                  $\{mE(e1\_L1 \mapsto v1\_L1 \mapsto v1\_L1)\})$
            act5 : $tG\_V := tG\_V \cup \{newV \mapsto Vertex2\}$
            act6 : $tG\_E := (tG\_E \setminus \{edgeName(mE(e1\_L1 \mapsto v1\_L1 \mapsto v1\_L1)) \mapsto Edge1\}) \cup$
                  $\{newE \mapsto Edge2\}$
      end
END
```

Figure 7.6: Event-B Rule Event

Preservation of semantics: According to Def. 39, the semantics of Event-B, the first event that occurs must be the initialization event. This event occurs only once in

INVARIANTS

prop1 : $\mathit{finite}(\mathit{incG})$
prop2 : $\mathit{card}(\mathit{incG}) \leq 2$

Figure 7.7: Stating Properties

any computation. In our translation, the occurrence of this event will generate a state that represents the start graph of the grammar. From this point on, any enabled event may happen. Each of the other events represents one rule of the grammar: the guard describes the existence of a match, and the actions describe the effect of the rule application. Whenever there is a match for a rule according to Def. 15 the guard of the corresponding event will be true and this event will be enabled (and also, if the event is enabled, there must be a match for the corresponding rule). Among all enabled events, the choice of the one that will happen is non-deterministic, exactly as defined in the semantics of a graph grammar. The effect of the occurrence of an event is a parallel assignment to the variables that compose the description of the state graph. These assignments were defined according to Def. 16, that was proven to be equivalent to a SPO derivation step. Thus, the transition system of the event-B model generated from a graph grammar corresponds exactly to the behavior of the grammar.

Proving Properties: Once the start graph and all rules are represented in the event-B model, the property to be proved can be stated as an invariant. For example, we could add the invariants $finite(incG)$ and $card(incG) \leq 2$, meaning respectively that any reachable graph has a finite number of edges and that no reachable graph can have more than 2 edges (see Figure 7.7). For the given example, these properties are true, and this can be easily proven by the Rodin platform.

The example described above generated 24 proof obligations with 22 of them proved automatically. The event-B specification of the Token Ring example is detailed in Appendix B.

8 CONCLUSION

In this thesis we introduced a relational and logical approach to graph grammars to allow the analysis of asynchronous distributed systems with infinite state space. We have used relational structures to characterize graph grammars and defined rule applications as definable transductions. We have first considered graph grammars defined over simple (typed) graphs, and then we extended the representation to attributed graphs and grammars with negative application conditions. We have shown that our approach offers a faithful encoding for SPO graph grammars and can thus be used as basis to enable the use of the theorem proving techniques to prove properties within this approach, complementing the existing approaches based on model checking techniques. Our main contribution should not be seen as a new approach to describe graph grammars, but rather as a way to allow theorem proving techniques (and tools) to be used in existing approaches (we modelled SPO here, but the theory could be used as basis to handle other approaches as well). This is relevant since graph grammars offer an interesting specification technique for a variety of application areas and up to now theorem proving techniques could not be used to analyse properties of graph grammars. The main contributions of this work are:

- The *relational and logical representation of graph grammars* (Chapter 3) establishes the theoretical foundations for the analysis of graph grammars through theorem proving. We represent graph grammars and their behaviour using relational and logical structures because they are the basis of theorem provers. Related works (STRECKER, 2008; BARESI; SPOLETINI, 2006) that adopt a description of graph grammars based on logical or set theoretical representations either are not effectively verifying properties of graph grammars or are limited to analyse a system for a finite scope, whose size is user-defined. Approaches for analysing infinite-state graph grammars (BALDAN; CORRADINI; KÖNIG, 2008; BALDAN; KÖNIG; RENSINK, 2005) derive the model as approximations, which can result in inconclusive verification reports.

  The definition of graph grammars as relational structures (Def. 13) allows the association of a graph grammar to a tuple composed of a set and a collection of relations over this set. The set describes the domain of the structure (the set of vertices and edges of the graph grammar) and the relations define the type graph, the initial graph and the rules. A series of logical conditions impose restrictions to the elements of these relations such that they really represent the components of a graph grammar (graphs, typed graphs, graph morphisms and rules). The application of a rule is described by a definable transduction (Def. 16), that can be seen as an inference rule on the relational structure associated to a graph grammar. The result of the transduction is another graph grammar whose initial state corresponds to the result of the application of a rule at a given match to the initial state of the original grammar. The other components of the grammar remain unchanged (i.e., the resulting grammar has the same type graph and rules of the original one). Propositions 7 and 9 assure that the adopted encoding is well-defined. For verification purposes, the relations of the relational structure define axioms that are used in the proofs and properties about reachable states are proven by induction: first (base case) the property is verified for the initial graph and then, at the inductive step, the property is verified for every rule of the grammar applicable to a reachable graph G, considering that the property is valid for G.

- The *relational approach for attributed graph grammars* (Chapter 4) is an extension of the basic formalism integrating the use of data types into graphs. Attributed graph grammars are very interesting from a practical point of view, since it is possible to use variables and terms when specifying the behaviour expressed by rules. These values (or terms) come from algebras specified as abstract data types. The use of attributed graphs gives the specifier a language that is more suitable for specification, merging the advantages of the graphical representation with the standard representation of classical data types. From a practical perspective, attributed graphs are needed, since it is not feasible to encode data types like natural numbers or strings, etc. into graphs. For verification, the presence of attributes poses additional problems, since data types are often infinite sets. In fact, even restricting to only finite sets, specifications using attributed graphs often give rise to non-verifiable systems due to state-explosion. There are few approaches to verify attributed graph grammars, like (KASTENBERG, 2006) and (KÖNIG; KOZIOURA, 2008) and they work for limited classes of grammars. We show that attributes can be smoothly integrated in our representation of graph grammars. Our approach provides a basis for a framework to reason about a large class of graph grammars, including those grammars that specify systems with infinite state-space, without using any kind of approximation.

  Definitions 29 and 31 express the relational representation of an attributed graph grammar. Propositions 16 and 18 assure that the relational extension is well-defined. The proof strategy applied in verification step is the same described before: we use mathematical induction, considering that the relations of the relational structure define axioms to be used during the proof. The difference is that now we use variables as attributes in the left- and right-hand sides of rules, and then, in many cases, at the inductive step the development of the proof involves variables. In this case, in order to establish the property, we must regard the equations of the applied rule as axioms.

- The *extension to graph grammars with negative application conditions* (Chapter 5) allows the specification that a certain structure is forbidden when performing a rule application, enhancing the expressiveness of the transformation. Particularly, negative application conditions restrict the application of a rule by expressing that a specific structure (e.g. nodes, edges or subgraphs) must not be present before applying the rule to a certain state-graph. Application conditions are commonly used in nontrivial specifications. As emphasized in (HABEL; HECKEL; TAENTZER, 1996), they are frequently expressed informally by assuming a kind of control mechanism that is not specified. Nevertheless such strategy prohibits formal specification and verification. The expression of NACs is currently possible in graph grammar tools

(ERMEL; RUDOLF; TAENTZER, 1999; SCHüRR; WINTER; ZüNDORF, 1999) that focus on the analysis of conflicts and functional behaviour. NACs can also be specified in GROOVE (KASTENBERG; RENSINK, 2006b) for the analysis of infinite-state graph grammars in case that the state-space can be computed on a finitely representable fragment .

Definition 36 associates a relational structure to a graph grammar with negative application conditions. Proposition 20 shows the well-definedness of our relational definition. In this approach, extra conditions must be checked before a rule application assuming that the forbidden elements are not in the state-graph. In verification step, the existence of NACs determines extra conditions that can be used during the proofs.

- The *property patterns* (Chapter 6) proposal contains 15 pattern classes in which functional and structural requirements of reachable states can be formulated. The patterns have the goal of helping and simplifying the task of stating precise requirements to be verified. It must provides enough help for the specification of properties over reachable states of graph grammars. We believe that the proposed patterns represent the first step towards a specification pattern for properties over states in the context of graph grammars. Differently from most existing approaches (DWYER; AVRUNIN; CORBETT, 1999; CHECHIK; PAUN, 1999; SALAMAH et al., 2007), we focus on properties about reachable states for (infinite-)state verification. Most of existing patterns for property specification describe properties about traces for finite-state verification tools. These two approaches are complementary.

  Tables 6.1, 6.2 and 6.3 describe a standard library of functions that describe typical characteristics or elements of graphs (like the set of vertices of some type, the set of edges of some type, the cardinality of vertices, etc.). These functions were defined in the framework of relational graph grammars. Table 6.4 proposes a pattern taxonomy and Tables 6.6 and 6.7 list a collection of patterns for property specifications.

- The *modelling of graph grammar specifications in event-B structures* (Chapter 7) enables the use of event-B provers (for instance, through the Rodin platform) to demonstrate properties of a graph grammar. Event-B (DEPLOY, 2010) has been successfully used in several other applications, having available tool support for both model specification and analysis. Event-B was chosen due to the similarity between event-B models and graph grammar specifications, specially concerning the rule-based behaviour. Many other works have been concerning on theorem proving concurrent systems (ZEYDA; CAVALCANTI, 2009; ISOBE; ROGGENBACH, 2008a; LEHMANN; LEUSCHEL, 2003), but for asynchronous systems, graph grammars have the advantage because of its visual and modular style.

  To define the event-B model, we used the relational definition of graph grammars. The type graph is defined in an event-B context, where vertex, edge types and the incidence relation relating them are constants. A set of axioms define these sets explicitly. A graph typed over a type graph is modelled by a set of variables describing its set of vertices, incidence relation, and typing functions. The compatibility conditions of types and source and target of edges can be stated as invariants. The initialization event is used to create the start graph. The structure of a rule is defined by sets, constants and axioms. The behaviour of a rule is described by an event with guard conditions. A set of actions update the state graph according to the rule.

Finally, we can say that the research field in theorem proving graph grammars is just in its first stages. There are a number of open issues that may be subject of future works.

- Besides implementation, case studies are necessary to evaluate and improve the proposed approach. Up to now, the extensions of the graph grammar basic formalism were not specified in Rodin platform. We could also investigate to which extent the theory of refinement, that is very well-developed in event-B, could be used to validate a stepwise development based on graph grammars. Another plan is the implementation of the data type reachable graph to be used in the specification and verification of graph grammar models. This strategy must be compared and evaluated with relation to the adopted implementation.

- Other classes of graph grammars not considered in this thesis comprise many practical applications. For instance, object-based graph grammars (DOTTI et al., 2003), timed object-based graph grammars (MICHELON; COSTA; RIBEIRO, 2007, 2006), object-oriented graph grammars (FERREIRA; FOSS; RIBEIRO, 2007) and many others (SCHFüRR, 1997) possibly with other kind of graph structures, like hypergraphs, labelled and attributed hypergraphs, have their own application fields. So it should be appealing to investigate a general description of the relational approach such that many kinds of graphs and/or grammars become instances of this general framework.

- The approach here proposed may be defined for Double-Pushout (DPO) graph grammars (EHRIG et al., 1997) without any mayor problems. In the SPO approach it is just necessary to find an image of the left-hand side of a rule into a reachable graph in order that a rule could be applied. In the DPO approach some extra restrictions must be checked, named *gluing condition*, before a rule can be applied. This means that some extra logic formulas (or extra guard conditions in case of event B structures) must be included to be verified before a rule application.

- The property patterns may also be incorporated in a proof framework. It would be helpful, as far as possible, to detail for each stated requirement the properties or lemmas that must be claimed for the conclusion of the proof, including strategies of proofs that can be adopted in each case. Simultaneously, a structured English grammar could be developed to assist the formulation of properties. Besides that, a natural extension of the stated patterns would be the investigation of requirements described with higher-order logics. We should, at last, complement and evaluate our pattern system surveying an appropriate number of real-world specifications.

- Another topic of investigation is the use of theorem proving technique to analyse other kind of properties, like safety and liveness properties. For instance, controllability, or the property of reaching a particular (set of) state(s) of the system whatever be the current one, is an important subject of analysis. Such property can not be verified through mathematical induction, since it is not finitary. It should be defined over all future behaviours of the system.

REFERENCES

ABDULLA, P. A.; JONSSON, B.; NILSSON, M.; D'ORSO, J.; SAKSENA, M. Regular Model Checking for LTL(MSO). In: CAV, 2004. **Proceedings...** Springer, 2004. p.348–360. (Lecture Notes in Computer Science, v.3114).

ABRIAL, J.-R. **The B-book**: assigning programs to meanings. New York, NY, USA: Cambridge University Press, 1996.

ABRIAL, J.-R. Formal methods in industry: achievements, problems, future. In: ICSE '06: PROCEEDING OF THE 28TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 2006, New York, NY, USA. **Proceedings...** ACM Press, 2006. p.761–768.

ABRIAL, J.-R. A System Development Process with Event-B and the Rodin Platform. In: ICFEM, 2007. **Proceedings...** Springer, 2007. p.1–3. (Lecture Notes in Computer Science, v.4789).

ABRIAL, J.-R.; BUTLER, M.; HALLERSTEDE, S.; HOANG, T. S.; MEHTA, F.; VOISIN, L. Rodin: an open toolset for modelling and reasoning in event-b. **International Journal on Software Tools for Technology Transfer (STTT)**, [S.l.], April 2010.

AIT-AMEUR, Y.; BARON, M.; KAMEL, N.; MOTA, J.-M. Encoding a process algebra using the Event B method: application to the validation of human&#x2013;computer interactions. **Int. J. Softw. Tools Technol. Transf.**, Berlin, Heidelberg, v.11, n.3, p.239–253, 2009.

ALI, S. S. **ANALOG**: a logical language for natural language processing. 1994.

ALUR, R.; DILL, D. L. A Theory of Timed Automata. **Theoretical Computer Science**, [S.l.], v.126, n.2, p.183–235, 1994.

AMBRIOLA, V.; GERVASI, V. On the Systematic Analysis of Natural Language Requirements with CIRCE. **Automated Software Engg.**, Hingham, MA, USA, v.13, n.1, p.107–167, 2006.

BACK, R.-J.; SERE, K. Stepwise Refinement of Action Systems. In: Proceedings of the International Conference on Mathematics of Program Construction, 375th Anniversary of the Groningen University, 1989, London, UK. **Proceedings...** Springer-Verlag, 1989. p.115–138.

BALDAN, P.; CORRADINI, A.; KÖNIG, B. Unfolding-Based Verification for Graph Transformation Systems. In: UNIGRA '03: UNIFORM APPROACHES TO GRAPHICAL SPECIFICATION TECHNIQUES (WARSAW), 2003. **Proceedings...** [S.l.: s.n.], 2003.

BALDAN, P.; CORRADINI, A.; KöNIG, B. A framework for the verification of infinite-state graph transformation systems. **Inf. Comput.**, Duluth, MN, USA, v.206, n.7, p.869–907, 2008.

BALDAN, P.; CORRADINI, A.; KÖNIG, B. A Framework for the Verification of Infinite-State Graph Transformation Systems. **Information and Computation**, [S.l.], v.206, p.869–907, 2008.

BALDAN, P.; CORRADINI, A.; MONTANARI, U.; RIBEIRO, L. Unfolding semantics of graph transformation. **Inf. Comput.**, Duluth, MN, USA, v.205, n.5, p.733–782, 2007.

BALDAN, P.; KÖNIG, B. Approximating the behaviour of graph transformation systems. In: ICGT '02 (INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATION), 2002. **Proceedings...** Springer, 2002. p.14–29. (LNCS, v.2505).

BALDAN, P.; KÖNIG, B.; KÖNIG, B. A Logic for Analyzing Abstractions of Graph Transformation Systems. In: SAS '03 (INTERNATIONAL STATIC ANALYSIS SYMPOSIUM), 2003. **Proceedings...** Springer, 2003. p.255–272. (LNCS, v.2694).

BALDAN, P.; KÖNIG, B.; RENSINK, A. Graph Grammar Verification through Abstraction. In: ABSTRACTS COLLECTION – GRAPH TRANSFORMATIONS AND PROCESS ALGEBRAS FOR MODELING DISTRIBUTED AND MOBILE SYSTEMS, 2005. **Proceedings...** [S.l.: s.n.], 2005. (Dagstuhl Seminar Proceedings 04241).

BARESI, L.; SPOLETINI, P. On the Use of Alloy to Analyze Graph Transformation Systems. In: ICGT, 2006. **Proceedings...** Springer, 2006. p.306–320. (LNCS, v.4178).

BARROS LUCENA, E. de. Reasoning about Petri Nets in HOL. In: TPHOLS, 1991. **Proceedings...** IEEE Computer Society, 1991. p.384–394.

BASTEN, T.; HOOMAN, J. Process Algebra in PVS. In: TACAS '99: PROCEEDINGS OF THE 5TH INTERNATIONAL CONFERENCE ON TOOLS AND ALGORITHMS FOR CONSTRUCTION AND ANALYSIS OF SYSTEMS, 1999, London, UK. **Proceedings...** Springer-Verlag, 1999. p.270–284.

BEHM, P.; BENOIT, P.; FAIVRE, A.; MEYNADIER, J.-M. Météor: a successful application of B in a large project. In: FM '99: PROCEEDINGS OF THE WOLD CONGRESS ON FORMAL METHODS IN THE DEVELOPMENT OF COMPUTING SYSTEMS-VOLUME I, 1999, London, UK. **Proceedings...** Springer-Verlag, 1999. p.369–387.

BEHRMANN, G.; DAVID, A.; LARSEN, K. G. Tutorial on UPPAAL. In: **Formal Methods for the Design of Real-Time Systems**. [S.l.]: Springer, 2004. p.200–236. (LNCS, v.3185).

BIERE, A.; CIMATTI, A.; CLARKE, E. M.; ZHU, Y. Symbolic Model Checking without BDDs. In: TACAS '99: PROCEEDINGS OF THE 5TH INTERNATIONAL CONFERENCE ON TOOLS AND ALGORITHMS FOR CONSTRUCTION AND ANALYSIS OF SYSTEMS, 1999, London, UK. **Proceedings...** Springer-Verlag, 1999. p.193–207.

BITSCH, F. Safety Patterns - The Key to Formal Specification of Safety Requirements. In: SAFECOMP ’01: PROCEEDINGS OF THE 20TH INTERNATIONAL CONFERENCE ON COMPUTER SAFETY, RELIABILITY AND SECURITY, 2001, London, UK. **Proceedings. . .** Springer-Verlag, 2001. p.176–189.

BLUMENSATH, B.; COURCELLE, B. Recognizability, hypergraph operations, and logical types. **Information and Computation**, [S.l.], v.204, n.6, p.853–919, 2006.

BOTTONI, P.; TAENTZER, G.; SCHüRR, A. Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation. In: VL ’00: PROCEEDINGS OF THE 2000 IEEE INTERNATIONAL SYMPOSIUM ON VISUAL LANGUAGES (VL’00), 2000, Washington, DC, USA. **Proceedings. . .** IEEE Computer Society, 2000. p.59.

BOUAJJANI, A.; HABERMEHL, P.; VOJNAR, T. Abstract Regular Model Checking. In: CAV, 2004. **Proceedings. . .** Springer, 2004. p.372–386. (Lecture Notes in Computer Science, v.3114).

BOWEN, J. P.; HINCHEY, M. G. The use of industrial-strength formal methods. In: COMPSAC ’97: PROCEEDINGS OF THE 21ST INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, 1997, Washington, DC, USA. **Proceedings. . .** IEEE Computer Society, 1997. p.332–337.

BOWEN, J. P.; HINCHEY, M. G. Ten commandments revisited: a ten-year perspective on the industrial application of formal methods. In: FMICS ’05: PROCEEDINGS OF THE 10TH INTERNATIONAL WORKSHOP ON FORMAL METHODS FOR INDUSTRIAL CRITICAL SYSTEMS, 2005, New York, NY, USA. **Proceedings. . .** ACM Press, 2005. p.8–16.

BOWEN, J. P.; HINCHEY, M. G. Ten Commandments of Formal Methods ...Ten Years Later. **Computer**, Los Alamitos, CA, USA, v.39, n.1, p.40–48, 2006.

BUNKE, H. Graph Grammars - a Useful Tool for Pattern Recognition? In: INTERNATIONAL WORKSHOP ON GRAPH-GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 4., 1991, London, UK. **Proceedings. . .** Springer-Verlag, 1991. p.43–46.

BURCH, J. R.; CLARKE, E. M.; MCMILLAN, K. L.; DILL, D. L.; HWANG, L. J. Symbolic model checking: 1020 states and beyond. **Information and Computation**, Duluth, MN, USA, v.98, n.2, p.142–170, 1992.

CAMILLERI, A.; INVERARDI, P.; NESI, M. Combining interaction and automation in process algebra verification. In: TAPSOFT ’91: PROCEEDINGS OF THE INTERNATIONAL JOINT CONFERENCE ON THEORY AND PRACTICE OF SOFTWARE DEVELOPMENT ON ADVANCES IN DISTRIBUTED COMPUTING (ADC) AND COLLOQUIUM ON COMBINING PARADIGMS FOR SOFTWARE DEVELOPMENT (CCPSD): VOL. 2, 1991, New York, NY, USA. **Proceedings. . .** Springer-Verlag New York: Inc., 1991. p.283–296.

CANSELL, D.; GOPALAKRISHNAN, G.; JONES, M.; MéRY, D.; WEINZOEPFLEN, A. Incremental Proof of the Producer/Consumer Property for the PCI Protocol. In: ZB ’02: PROCEEDINGS OF THE 2ND INTERNATIONAL CONFERENCE OF B AND Z

USERS ON FORMAL SPECIFICATION AND DEVELOPMENT IN Z AND B, 2002, London, UK. **Proceedings. . .** Springer-Verlag, 2002. p.22–41.

CERVO, L. V.; RIBEIRO, L. DNA-Based Modelling of Parallel Algorithms. In: WOB, 2002. **Proceedings. . .** [S.l.: s.n.], 2002. p.16–23.

CHECHIK, M.; PAUN, D. O. Events in Property Patterns. In: INTERNATIONAL SPIN WORKSHOPS ON THEORETICAL AND PRACTICAL ASPECTS OF SPIN MODEL CHECKING, 5., 1999, London, UK. **Proceedings. . .** Springer-Verlag, 1999. p.154–167.

CHOPPY, C.; MAYERO, M.; PETRUCCI, L. Experimenting Formal Proofs of Petri Nets Refinements. **Electron. Notes Theor. Comput. Sci.**, Amsterdam, The Netherlands, The Netherlands, v.214, p.231–254, 2008.

CLARKE, E. M.; GRUMBERG, O.; JHA, S.; LU, Y.; VEITH, H. Progress on the State Explosion Problem in Model Checking. In: INFORMATICS - 10 YEARS BACK. 10 YEARS AHEAD., 2001, London, UK. **Proceedings. . .** Springer-Verlag, 2001. p.176–194.

CLARKE, E. M.; WING, J. M. Formal methods: state of the art and future directions. **ACM Computing Surveys**, New York, NY, USA, v.28, n.4, p.626–643, 1996.

COBLEIGH, R. L.; AVRUNIN, G. S.; CLARKE, L. A. User guidance for creating precise and accessible property specifications. In: SIGSOFT ’06/FSE-14: PROCEEDINGS OF THE 14TH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, 2006, New York, NY, USA. **Proceedings. . .** ACM, 2006. p.208–218.

COMPUTER AIDED VERIFICATION, 16TH INTERNATIONAL CONFERENCE, CAV 2004, BOSTON, MA, USA, JULY 13-17, 2004, PROCEEDINGS, 2004. **Proceedings. . .** Springer, 2004. (Lecture Notes in Computer Science, v.3114).

CONSORTIUM, E. **Eclipse Graphical Editing Framework (GEF) Version 3.5.2**. 2010.

COPSTEIN, B.; COSTA MÓRA, M. da; RIBEIRO, L. An Environment for Formal Modeling and Simulation of Control Systems. In: SS ’00: PROCEEDINGS OF THE 33RD ANNUAL SIMULATION SYMPOSIUM, 2000, Washington, DC, USA. **Proceedings. . .** IEEE Computer Society, 2000. p.74.

CORBETT, J. C.; DWYER, M. B.; HATCLIFF, J.; ROBBY. A Language Framework for Expressing Checkable Properties of Dynamic Software. In: INTERNATIONAL SPIN WORKSHOP ON SPIN MODEL CHECKING AND SOFTWARE VERIFICATION, 7., 2000, London. **Proceedings. . .** Springer, 2000. p.205–223.

COURCELLE, B. The Monadic Second-Order Logic of Graphs I: recognizable sets of finite graphs. **Information and Computation**, [S.l.], v.85, n.1, p.12–75, 1990.

COURCELLE, B. Graph grammars, monadic second-order logic and theory of graph minors. In: GRAPH STRUCTURE THEORY, 1991. **Proceedings. . .** American Mathematical Society, 1991. p.565–590. (Contemporary Mathematics, v.147).

COURCELLE, B. The Monadic Second order Logic of Graphs VI: on several representations of graphs by relational structures. **Discrete Applied Mathematics**, [S.l.], v.54, n.2-3, p.117–149, 1994.

COURCELLE, B. Monadic Second-Order Definable Graph Transductions: a survey. **Theoretical Computer Science**, [S.l.], v.126, n.1, p.53–75, 1994.

COURCELLE, B. The Expression of Graph Properties and Graph Transformations in Monadic Second-Order Logic. In: HANDBOOK OF GRAPH GRAMMARS, 1997, River Edge, NJ, USA. **Proceedings...** World Scientific Publishing Co.: Inc., 1997. p.313–400.

COURCELLE, B. Graph Operations and Monadic Second-Order Logic: a survey. In: LPAR, 2000. **Proceedings...** Springer, 2000. p.20–24. (LNCS, v.1955).

COURCELLE, B. Recognizable Sets of Graphs, Hypergraphs and Relational Structures: a survey. In: DEVELOPMENTS IN LANGUAGE THEORY, 2004. **Proceedings...** Springer, 2004. p.1–11. (LNCS, v.3340).

COURCELLE, B.; ENGELFRIET, J.; ROZENBERG, G. Handle-Rewriting Hypergraph Grammars. **Journal of Computer and System Sciences (JCSS)**, [S.l.], v.46, n.2, p.218–270, 1993.

CRAIGEN, D.; GERHART, S.; RALSTON, T. **An International Survey of Industrial Applications of Formal Methods (Volume 1**: purpose, approach, analysis and conclusions, volume 2: case studies). National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161, USA: [s.n.], 1993. (NIST GCR 93/626-V1 & NIST GCR 93-626-V2 (Order numbers: PB93-178556/AS & PB93-178564/AS)).

DAMCHOOM, K.; BUTLER, M. Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B. In: SBMF 2009, 2009. **Proceedings...** Springer, 2009. v.5902, p.134–152. Springer LNCS 5902.

DAMCHOOM, K.; BUTLER, M.; ABRIAL, J.-R. Modelling and Proof of a Tree-Structured File System in Event-B and Rodin. In: ICFEM ’08: PROCEEDINGS OF THE 10TH INTERNATIONAL CONFERENCE ON FORMAL METHODS AND SOFTWARE ENGINEERING, 2008, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2008. p.25–44.

DAVIS, J. F. The affordable application of formal methods to software engineering. In: SIGADA ’05: PROCEEDINGS OF THE 2005 ANNUAL ACM SIGADA INTERNATIONAL CONFERENCE ON ADA, 2005, New York, NY, USA. **Proceedings...** ACM Press, 2005. p.57–62.

DELZANNO, G. Automatic Verification of Parameterized Cache Coherence Protocols. In: CAV ’00: PROCEEDINGS OF THE 12TH INTERNATIONAL CONFERENCE ON COMPUTER AIDED VERIFICATION, 2000, London, UK. **Proceedings...** Springer-Verlag, 2000. p.53–68.

DEPLOY. **Event-B and the Rodin Platform**. http://www.event-b.org/ (last accessed January 2010). Rodin Development is supported by European Union ICT Projects DEPLOY (2008 to 2012) and RODIN (2004 to 2007).

DIJKSTRA, E. **A Discipline of Programming**. [S.l.]: Prentice-Hall International, 1976.

DIXIT, V. V.; MOLDOVAN, D. I. Minimal State Space Search in Parallel Production Systems. **IEEE Trans. on Knowl. and Data Eng.**, Piscataway, NJ, USA, v.3, n.4, p.435–443, 1991.

DOTTI, F. L.; FOSS, L.; RIBEIRO, L.; SANTOS, O. M. Verification of distributed object-based systems. In: INTERNATIONAL CONFERENCE ON FORMAL METHODS FOR OPEN OBJECT-BASED DISTRIBUTED SYSTEMS, 6., 2003. **Proceedings...** Springer, 2003. p.261–275. (LNCS, v.2884).

DOTTI, F. L.; RIBEIRO, L. Specification of Mobile Code Systems using Graph Grammars. In: **Formal Methods for Open Object-Based Distributed Systems**. [S.l.]: Kluwer, 2000. p.45–64.

DOTTI, F. L.; RIBEIRO, L.; SANTOS, O. M. dos. Specification and Analysis of Fault Behaviours Using Graph Grammars. In: AGTIVE, 2003. **Proceedings...** Springer, 2003. p.120–133. (Lecture Notes in Computer Science, v.3062).

DOTTI, F. L.; RIBEIRO, L.; SANTOS, O. M. dos; PASINI, F. Verifying Object-based Graph Grammars: an assume-guarantee approach. **Software and System Modeling**, [S.l.], v.5, n.3, p.289–311, 2006.

DRUSINSKY, D. Visual formal specification using (N)TLCharts: statechart automata with temporal logic and natural language conditioned transitions. In: PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, 2004. PROCEEDINGS. 18TH INTERNATIONAL, 2004. **Proceedings...** [S.l.: s.n.], 2004. p.268–.

DUARTE, L. M.; DOTTI, F. L.; COPSTEIN, B.; RIBEIRO, L. Simulation of Mobile Applications. In: COMMUNICATION NETWORKS AND DISTRIBUTED SYSTEMS MODELING AND SIMULATION CONFERENCE, 2002. **Proceedings...** [S.l.: s.n.], 2002. v.1, p.1–15.

DUTERTRE, B.; SCHNEIDER, S. Using a PVS Embedding of CSP to Verify Authentication Protocols. In: TPHOLS ’97: PROCEEDINGS OF THE 10TH INTERNATIONAL CONFERENCE ON THEOREM PROVING IN HIGHER ORDER LOGICS, 1997, London, UK. **Proceedings...** Springer-Verlag, 1997. p.121–136.

DWYER, M. B.; AVRUNIN, G. S.; CORBETT, J. C. Property specification patterns for finite-state verification. In: FMSP ’98: PROCEEDINGS OF THE SECOND WORKSHOP ON FORMAL METHODS IN SOFTWARE PRACTICE, 1998, New York, NY, USA. **Proceedings...** ACM, 1998. p.7–15.

DWYER, M. B.; AVRUNIN, G. S.; CORBETT, J. C. Patterns in property specifications for finite-state verification. In: ICSE ’99: PROCEEDINGS OF THE 21ST INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 1999, New York, NY, USA. **Proceedings...** ACM, 1999. p.411–420.

DWYER, M. B.; HATCLIFF, J.; ROBBY, R.; PASAREANU, C. S.; VISSER, W. Formal Software Analysis Emerging Trends in Software Model Checking. In: FOSE ’07: 2007 FUTURE OF SOFTWARE ENGINEERING, 2007. **Proceedings...** IEEE Computer Society, 2007. p.120–136.

EDELKAMP, S.; JABBAR, S.; LLUCH-LAFUENTE, A. Heuristic Search for the Analysis of Graph Transition Systems. In: ICGT, 2006. **Proceedings...** Springer, 2006. p.414–429. (LNCS, v.4178).

EDMUND M. CLARKE, J.; GRUMBERG, O.; PELED, D. A. **Model checking**. Cambridge, MA, USA: MIT Press, 1999.

EHRIG, H.; EHRIG, K.; PRANGE, U.; TAENTZER, G. Fundamental Theory for Typed Attributed Graphs and Graph Transformation based on Adhesive HLR Categories. **Fundamta Informaticae**, Amsterdam, The Netherlands, The Netherlands, v.74, n.1, p.31–61, 2006.

EHRIG, H.; ENGELS, G.; KREOWSKI, H.-J.; ROZENBERG, G. (Ed.). **Handbook of graph grammars and computing by graph transformation**: vol. 2: applications, languages, and tools. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1999.

EHRIG, H.; HECKEL, R.; KORFF, M.; LöWE, M.; RIBEIRO, L.; WAGNER, A.; CORRADINI, A. Algebraic approaches to graph transformation. Part II: single pushout approach and comparison with double pushout approach. **Handbook of graph grammars and computing by graph transformation: volume I. foundations**, River Edge, NJ, USA, p.247–312, 1997.

EHRIG, K.; ERMEL, C.; HäNSGEN, S.; TAENTZER, G. Generation of visual editors as eclipse plug-ins. In: ASE '05: PROCEEDINGS OF THE 20TH IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, 2005, New York, NY, USA. **Proceedings...** ACM, 2005. p.134–143.

ERMEL, C.; RUDOLF, M.; TAENTZER, G. The AGG approach: language and environment. **Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools**, River Edge, NJ, USA, p.551–603, 1999.

FERRARI, G.; GNESI, S.; MONTANARI, U.; PISTORE, M.; RISTORI, G. Verifying Mobile Processes in the HAL Environment. In: CAV '98: PROCEEDINGS OF THE 10TH INTERNATIONAL CONFERENCE ON COMPUTER AIDED VERIFICATION, 1998, London, UK. **Proceedings...** Springer-Verlag, 1998. p.511–515.

FERREIRA, A. P. L.; FOSS, L.; RIBEIRO, L. Formal Verification of Object-Oriented Graph Grammars Specifications. **Electr. Notes Theor. Comput. Sci.**, [S.l.], v.175, n.4, p.101–114, 2007.

FISHER, M.; KONEV, B.; LISITSA, A. Practical Infinite-State Verification with Temporal Reasoning. In: VISSAS, 2005. **Proceedings...** IOS Press, 2005. p.91–100. (NATO Security through Science Series D: Information and Communication Security, v.1).

FLAKE, S.; MUELLER, W. Structured English for model checking specifications, Proc. Methoden u. Beschreibungssprachen zur Modellierung u. Verifikation von Schaltungen u. Systemen, VDE Verlag 2000B. In: TRANS. AMER. MATH. SOC, 2000. **Proceedings...** VDE Verlag, 2000. p.2547–2552.

FORMAL METHODS FOR INDUSTRIAL CRITICAL SYSTEMS, 14TH INTERNATIONAL WORKSHOP, FMICS 2009, EINDHOVEN, THE NETHERLANDS, NOVEMBER 2-3, 2009. PROCEEDINGS, 2009. **Proceedings...** Springer, 2009. (Lecture Notes in Computer Science, v.5825).

FORMAL METHODS IN SOFTWARE AND SYSTEMS MODELING, ESSAYS DEDICATED TO HARTMUT EHRIG, ON THE OCCASION OF HIS 60TH BIRTHDAY, 2005. **Proceedings. . .** Springer, 2005. (Lecture Notes in Computer Science, v.3393).

FOSS, L. **A Translation from Object-Based Hypergraph Grammars into pi-Calculus (in Portuguese)**. 2003. Dissertação (Mestrado em Ciência da Computação) — PPGC, UFRGS.

FOSS, L.; RIBEIRO, L. A Translation from Object-Based Hypergraph Grammars into pi-Calculus. **Electr. Notes Theor. Comput. Sci.**, [S.l.], v.95, p.245–267, 2004.

FRANCA, R. B.; BECKER, L. B.; BODEVEIX, J.-P.; FARINES, J.-M.; FILALI, M. Towards Safe Design of Synchronous Bus Protocols in Event-B. In: SBMF, 2009. **Proceedings. . .** Springer, 2009. p.170–185. (Lecture Notes in Computer Science, v.5902).

FUCHS, N. E.; SCHWERTEL, U.; SCHWITTER, R. Attempto Controlled English - Not Just Another Logic Specification Language. In: LOPSTR '98: PROCEEDINGS OF THE 8TH INTERNATIONAL WORKSHOP ON LOGIC PROGRAMMING SYNTHESIS AND TRANSFORMATION, 1998, London. **Proceedings. . .** Springer, 1998. p.1–20.

FUSS, C.; MOSLER, C.; RANGER, U.; SCHULTCHEN, E. The Jury is still out: a comparison of agg, fujaba, and progres. **ECEASST**, [S.l.], v.6, 2007.

GEISS, R.; BATZ, G. V.; GRUND, D.; HACK, S.; SZALKOWSKI, A. GrGen: a fast spo-based graph rewriting tool. In: ICGT, 2006. **Proceedings. . .** Springer, 2006. p.383–397. (LNCS, v.4178).

GERBER, R.; GUNTER, E. L.; LEE, I. Implementing a Real-Time Process Algebra in HOL. In: TPHOLS, 1991. **Proceedings. . .** IEEE Computer Society, 1991. p.144–154.

GERVASI, V.; ZOWGHI, D. Reasoning about inconsistencies in natural language requirements. **ACM Trans. Softw. Eng. Methodol.**, New York, NY, USA, v.14, n.3, p.277–330, 2005.

GORDON, M. J. C.; MELHAM, T. F. (Ed.). **Introduction to HOL**: a theorem proving environment for higher order logic. New York, NY, USA: Cambridge University Press, 1993.

GRAPH-GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 3RD INTERNATIONAL WORKSHOP, WARRENTON, VIRGINIA, USA, DECEMBER 2-6, 1986, 1987. **Proceedings. . .** Springer, 1987. (Lecture Notes in Computer Science, v.291).

GRAPH TRANSFORMATIONS, THIRD INTERNATIONAL CONFERENCE, ICGT 2006, NATAL, RIO GRANDE DO NORTE, BRAZIL, SEPTEMBER 17-23, 2006, PROCEEDINGS, 2006. **Proceedings. . .** Springer, 2006. (LNCS, v.4178).

GROENBOOM, R.; HENDRIKS, C.; POLAK, I.; TERLOUW, J.; UDDING, J. T. Algebraic Proof Assistants in HOL. In: MPC '95: MATHEMATICS OF PROGRAM CONSTRUCTION, 1995, London, UK. **Proceedings. . .** Springer-Verlag, 1995. p.304–321.

GRUHN, V.; LAUE, R. Patterns for Timed Property Specifications. **Electr. Notes Theor. Comput. Sci.**, [S.l.], v.153, n.2, p.117–133, 2006.

GRUNSKE, L. Specification patterns for probabilistic quality properties. In: ICSE '08: PROCEEDINGS OF THE 30TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 2008, New York, NY, USA. **Proceedings...** ACM, 2008. p.31–40.

GUREVICH, Y. Monadic Second-Order Theories. In: BARWISE, J.; FEFERMAN, S. (Ed.). **Model-Theoretic Logics**. [S.l.]: Springer, 1985. p.479–506.

HABEL, A.; HECKEL, R.; TAENTZER, G. Graph grammars with negative application conditions. **Fundam. Inf.**, Amsterdam, The Netherlands, The Netherlands, v.26, n.3-4, p.287–313, 1996.

HAMID, N. A. Theorem proving with the COQ proof assistant: tutorial presentation. **J. Comput. Small Coll.**, , USA, v.24, n.2, p.230–230, 2008.

HANEBERG, D.; SCHELLHORN, G.; GRANDY, H.; REIF, W. Verification of Mondex electronic purses with KIV: from transactions to a security protocol. **Form. Asp. Comput.**, London, UK, v.20, n.1, p.41–59, 2007.

HAUSMANN, J. H.; HECKEL, R.; TAENTZER, G. Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation. In: ICSE '02: PROCEEDINGS OF THE 24TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 2002, New York, NY, USA. **Proceedings...** ACM, 2002. p.105–115.

HECKEL, R.; KüSTER, J. M.; TAENTZER, G. Confluence of Typed Attributed Graph Transformation Systems. In: ICGT '02: PROCEEDINGS OF THE FIRST INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATION, 2002, London, UK. **Proceedings...** Springer-Verlag, 2002. p.161–176.

HECKEL, R.; WAGNER, A. Ensuring Consistency of Conditional Graph Grammars - A Constructive Approach -. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.2, p.118 – 126, 1995. SEGRAGRA 1995, Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation.

HEITMEYER, C. Developing safety-critical systems: the role of formal methods and tools. In: SCS '05: PROCEEDINGS OF THE 10TH AUSTRALIAN WORKSHOP ON SAFETY CRITICAL SYSTEMS AND SOFTWARE, 2006, Darlinghurst, Australia, Australia. **Proceedings...** Australian Computer Society: Inc., 2006. p.95–99.

HINCHEY, M. G.; BOWEN, J. P. (Ed.). **Applications of Formal Methods**. [S.l.]: Prentice Hall, 1995.

HINCHEY, M. G.; BOWEN, J. P. (Ed.). **Industrial-Strength Formal Methods in Practice**. [S.l.]: Springer, 1999.

HOARE, C. A. R. Communicating sequential processes. **Commun. ACM**, New York, NY, USA, v.21, n.8, p.666–677, 1978.

HOLZMANN, G. J. The model checker Spin. **IEEE Transactions on Software Engineering**, Los Alamitos, CA, USA, v.23, n.5, p.279–295, 1997.

HOLZMANN, G. J. The Model Checker SPIN. **IEEE Trans. Softw. Eng.**, Piscataway, NJ, USA, v.23, n.5, p.279–295, 1997.

HOMMERSOM, A.; GROOT, P.; LUCAS, P. J. F.; BALSER, M.; SCHMITT, J. Verification of Medical Guidelines Using Background Knowledge in Task Networks. **IEEE Trans. on Knowl. and Data Eng.**, Piscataway, NJ, USA, v.19, n.6, p.832–846, 2007.

HUSSEIN, H. K.; HASSANIEN, A. E. Graph Grammar Algebraic Approach For Generating Fractal Pattern. In: WSCG'99 CONFERENCE PROCEEDINGS, 1999. **Proceedings...** [S.l.: s.n.], 1999.

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

INTERNATIONAL WORKSHOP ON THE HOL THEOREM PROVING SYSTEM AND ITS APPLICATIONS, AUGUST 1991, DAVIS, CALIFORNIA, USA, 1991., 1992. **Proceedings...** IEEE Computer Society, 1992.

ISOBE, Y.; ROGGENBACH, M. CSP-Prover - A Proof Tool for the Verification of Scalable Concurrent Systems. **JSSST (Japan Society for Software Science and Technology) Computer Software**, [S.l.], v.25, 2008.

ISOBE, Y.; ROGGENBACH, M. Proof Principles of CSP – CSP-Prover in Practice. In: LDIC 2007, 2008. **Proceedings...** Springer, 2008.

ISOBE, Y.; ROGGENBACH, M.; GRUNER, S. Extending CSP-Prover by deadlock-analysis: towards the verification of systolic arrays. In: FOSE 2005, 2005. **Proceedings...** Kindai-kagaku-sha, 2005. (Japanese Lecture Notes Series 31).

JACOBSON, I.; BOOCH, G.; RUMBAUGH, J. **The unified software development process**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

JEMNI BEN AYED, L.; SIALA, F. Specification and Verification of Multi-agent Systems Interaction Protocols Using a Combination of AUML and Event B. **Interactive Systems. Design, Specification, and Verification: 15th International Workshop, DSV-IS 2008 Kingston, Canada, July 16-18, 2008 Revised Papers**, Berlin, Heidelberg, p.102–107, 2008.

JöRGES, S.; MARGARIA, T.; STEFFEN, B. FormulaBuilder: a tool for graph-based modelling and generation of formulae. In: ICSE '06: PROCEEDINGS OF THE 28TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 2006, New York, NY, USA. **Proceedings...** ACM, 2006. p.815–818.

KASTENBERG, H. Towards Attributed Graphs in Groove: work in progress. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.154, n.2, p.47 – 54, 2006. Proceedings of the Workshop on Graph Transformation for Verification and Concurrency (GT-VC 2005).

KASTENBERG, H.; RENSINK, A. Model Checking Dynamic States in GROOVE. In: MODEL CHECKING SOFTWARE (SPIN), 2006. **Proceedings...** Springer-Verlag, 2006. p.299–305. (Lecture Notes in Computer Science, v.3925).

KASTENBERG, H.; RENSINK, A. Model Checking Dynamic States in GROOVE. In: SPIN, 2006. **Proceedings...** Springer, 2006. p.299–305. (Lecture Notes in Computer Science, v.3925).

KESTEN, Y.; MALER, O.; MARCUS, M.; PNUELI, A.; SHAHAR, E. Symbolic model checking with rich assertional languages. **Theor. Comput. Sci.**, Essex, UK, v.256, n.1-2, p.93–112, 2001.

KNIGHT, J. C. Challenges in the Utilization of Formal Methods. In: FTRTFT ’98: PROCEEDINGS OF THE 5TH INTERNATIONAL SYMPOSIUM ON FORMAL TECHNIQUES IN REAL-TIME AND FAULT-TOLERANT SYSTEMS, 1998, London, UK. **Proceedings. . .** Springer-Verlag, 1998. p.1–17.

KÖNIG, B.; KOZIOURA, V. Augur—A Tool for the Analysis of Graph Transformation Systems. **EATCS Bulletin**, [S.l.], v.87, p.125–137, November 2005. Appeared in The Formal Specification Column.

KöNIG, B.; KOZIOURA, V. Augur 2 — A New Version of a Tool for the Analysis of Graph Transformation Systems. **Electron. Notes Theor. Comput. Sci.**, Amsterdam, The Netherlands, The Netherlands, v.211, p.201–210, 2008.

KÖNIG, B.; KOZIOURA, V. Towards the Verification of Attributed Graph Transformation Systems. In: GRAPH TRANSFORMATIONS, 4TH INT. CONFERENCE, ICGT 2008, 2008. **Proceedings. . .** Springer, 2008. p.305–320. (Lecture Notes in Computer Science, v.5214).

KONRAD, S.; CHENG, B. H. C. Real-time specification patterns. In: SOFTWARE ENGINEERING, 27., 2005, New York. **Proceedings. . .** ACM, 2005. p.372–381.

KORFF, M. Application of Graph Grammars to Rule-Based Systems. In: INTERNATIONAL WORKSHOP ON GRAPH-GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 4., 1991, London, UK. **Proceedings. . .** Springer-Verlag, 1991. p.505–519.

KUMAR, V. Algorithms for Constraint Satisfaction Problems: a survey. **AI Magazine**, [S.l.], v.13, p.32–44, 1992.

LECOUTRE, C. **Constraint Networks**: techniques and algorithms. 592 pages: International Scientific and Technical Encyclopedia (ISTE Ltd) - John Wiley Inc., 2009. ISBN: 9781848211063.

LEHMANN, H.; LEUSCHEL, M. Inductive Theorem Proving by Program Specialisation: generating proofs for isabelle using ecce. In: LOPSTR, 2003. **Proceedings. . .** Springer, 2003. p.1–19. (Lecture Notes in Computer Science, v.3018).

LEMMA1-LTD. **The Proof Power Webpages**. 2010.

LETIER, E.; LAMSWEERDE, A. van. Deriving operational software specifications from system goals. **SIGSOFT Softw. Eng. Notes**, New York, NY, USA, v.27, n.6, p.119–128, 2002.

LLADÓS, J.; SÁNCHEZ, G. Symbol recognition using graphs. In: ICIP (2), 2003. **Proceedings. . .** [S.l.: s.n.], 2003. p.49–52.

LLUCH-LAFUENTE, A.; EDELKAMP, S.; LEUE, S. Partial Order Reduction in Directed Model Checking. In: INTERNATIONAL SPIN WORKSHOP ON MODEL CHECKING OF SOFTWARE, 9., 2002, London, UK. **Proceedings. . .** Springer-Verlag, 2002. p.112–127.

LöWE, M. Algebraic approach to single-pushout graph transformation. **Theor. Comput. Sci.**, Essex, UK, v.109, n.1-2, p.181–224, 1993.

LöWE, M.; KORFF, M.; WAGNER, A. An algebraic framework for the transformation of attributed graphs. In: **Term graph rewriting**: theory and practice. Chichester, UK, UK: John Wiley and Sons Ltd., 1993. p.185–199.

MCMILLAN, K. L. **Symbolic model checking**: an approach to the state explosion problem. 1992. Tese (Doutorado em Ciência da Computação) — , Pittsburgh, PA, USA.

MCNEW, J.-M.; KLAVINS, E. Model-Checking and Control of Self-Assembly. In: AMERICAN CONTROL CONFERENCE, 2006., 2006. **Proceedings. . .** [S.l.: s.n.], 2006. p.14–21.

MICHAEL, H. C.; W., B. R. **Impediments to Industrial Use of Formal Methods**. [S.l.: s.n.], 1996.

MICHELON, L.; COSTA, S. A. da; RIBEIRO, L. Formal Specification and Verification of Real-Time Systems using Graph Grammars. **Journal of The Brazilian Computer Society (JBCS)**, [S.l.], v.13, n.4, p.51–68, 2007.

MICHELON, L.; COSTA, S. A.; RIBEIRO, L. Specification of Real-Time Systems with Graph Grammars. In: BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, 2006. **Proceedings. . .** [S.l.: s.n.], 2006. p.97–112.

MILNER, R. **Communicating and mobile systems**: the π-calculus. New York, NY, USA: Cambridge University Press, 1999.

MONDRAGON, O.; GATES, A. Q. Supporting Elicitation And Specification Of Software Properties Through Patterns And Composite Propositions. **International Journal of Software Engineering and Knowledge Engineering**, [S.l.], v.14, n.1, p.21–41, 2004.

MONDRAGON, O.; GATES, A. Q.; ROACH, S.; MENDOZA, H.; SOKOLSKY, O. Generating Properties for Runtime Monitoring from Software Specification Patterns. **International Journal of Software Engineering and Knowledge Engineering**, [S.l.], v.17, n.1, p.107–126, 2007.

NIPKOW, T.; PAULSON, L. C.; WENZEL, M. **Isabelle/HOL — A Proof Assistant for Higher-Order Logic**. [S.l.]: Springer, 2002. (LNCS, v.2283).

OLIVEIRA, M. V. M.; CAVALCANTI, A. L. C.; WOODCOCK, J. C. P. Unifying theories in ProofPower-Z. In: UTP 2006: First International Symposium on Unifying Theories of Programming, 2006. **Proceedings. . .** Springer-Verlag, 2006. p.123–140. (LNCS, v.**4010**).

OWRE, S.; RUSHBY, J. M.; SHANKAR, N. PVS: a prototype verification system. In: CADE, 1992. **Proceedings. . .** Springer, 1992. p.748–752. (Lecture Notes in Computer Science, v.607).

PAULSON, L. C. **Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)**. [S.l.]: Springer, 1994. (Lecture Notes in Computer Science, v.828).

PAUN, D. O.; CHECHIK, M. Events in Linear-Time Properties. In: RE ’99: PROCEEDINGS OF THE 4TH IEEE INTERNATIONAL SYMPOSIUM ON REQUIREMENTS ENGINEERING, 1999, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1999. p.123–132.

PROJECT, C. **Circus**. 2010.

RANGER, U.; WEINELL, E. The Graph Rewriting Language and Environment PROGRES. **Applications of Graph Transformations with Industrial Relevance: Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers**, Berlin, Heidelberg, p.575–576, 2008.

RENSINK, A. The GROOVE Simulator: a tool for state space generation. In: APPLICATIONS OF GRAPH TRANSFORMATIONS WITH INDUSTRIAL RELEVANCE (AGTIVE), 2004. **Proceedings...** Springer-Verlag, 2004. p.479–485. (Lecture Notes in Computer Science, v.3062).

RENSINK, A. Canonical Graph Shapes. In: PROGRAMMING LANGUAGES AND SYSTEMS — EUROPEAN SYMPOSIUM ON PROGRAMMING (ESOP), 2004. **Proceedings...** Springer-Verlag, 2004. p.401–415. (Lecture Notes in Computer Science, v.2986).

RENSINK, A.; DOTOR, A.; ERMEL, C.; JURACK, S.; KNIEMEYER, O.; de Lara, J.; MAIER, S.; STAIJEN, T.; ZÜNDORF, A. Ludo: a case study for graph transformation tools. In: APPLICATIONS OF GRAPH TRANSFORMATION WITH INDUSTRIAL RELEVANCE, PROCEEDINGS OF THE THIRD INTERNATIONAL AGTIVE 2007 SYMPOSIUM, 2008, Heidelberg. **Proceedings...** [S.l.: s.n.], 2008. p.493–513. (LNCS, v.5088).

RENSINK, A.; SCHMIDT, Á.; VARRÓ, D. Model Checking Graph Transformations: a comparison of two approaches. In: ICGT 2004: SECOND INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATION, 2004. **Proceedings...** Springer, 2004. p.226–241. (LNCS, v.3256).

REZAZADEH, A.; EVANS, N.; BUTLER, M. Redevelopment of an Industrial Case Study Using Event-B and Rodin. In: BCS-FACS CHRISTMAS 2007 MEETING - FORMAL METHODS IN INDUSTRY, 2007. **Proceedings...** [S.l.: s.n.], 2007.

RIBEIRO, L. **Parallel Composition and Unfolding Semantics of Graph Grammars**. 1996. Tese (Doutorado em Ciência da Computação) — Technische Universit/"at Berlin.

RIBEIRO, L.; COPSTEIN, B. Specifying simulation models using graph grammars. In: ESS98: EUROPEAN SIMULATION SYMOPSIUM, 1998, Nottinhgham. **Proceedings...** [S.l.: s.n.], 1998. p.60–64.

RIBEIRO, L.; DOTTI, F. L.; BARDOHL, R. A Formal Framework for the Development of Concurrent Object-Based Systems. In: FORMAL METHODS IN SOFTWARE AND SYSTEMS MODELING, 2005. **Proceedings...** Springer, 2005. p.385–401. (Lecture Notes in Computer Science, v.3393).

ROBINSON, J. A.; VORONKOV, A. (Ed.). **Handbook of Automated Reasoning (in 2 volumes)**. [S.l.]: Elsevier and MIT Press, 2001.

RÖDEL, E. T.; DUARTE, L. M.; SANTOS, O. M. dos; , F. L. D. . Simulation of Mobile Applications in Open Environments. In: ANAIS DO IV WORKSHOP DE COMUNICAÇÃO SEM FIO E COMPUTAÇÃO MÓVEL, 2002, São Paulo. **Proceedings. . .** [S.l.: s.n.], 2002. p.246–256.

ROSENBLUM, D. S. Formal methods and testing: why the state-of-the art is not the state-of-the practice. **SIGSOFT Softw. Eng. Notes**, New York, NY, USA, v.21, n.4, p.64–66, 1996.

ROSSI, U. Can we really do without the support of formal methods in the verification of large designs? In: DAC '05: PROCEEDINGS OF THE 42ND ANNUAL CONFERENCE ON DESIGN AUTOMATION, 2005, New York, NY, USA. **Proceedings. . .** ACM Press, 2005. p.672–673.

ROZENBERG, G. (Ed.). **Handbook of graph grammars and computing by graph transformation**: volume I. Foundations. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1997.

RUDOLF, M. Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In: TAGT'98: SELECTED PAPERS FROM THE 6TH INTERNATIONAL WORKSHOP ON THEORY AND APPLICATION OF GRAPH TRANSFORMATIONS, 2000, London, UK. **Proceedings. . .** Springer-Verlag, 2000. p.238–251.

RUSHBY, J. Theorem proving for verification. **Modeling and verification of parallel processes**, New York, NY, USA, p.39–57, 2001.

SAID, M. yah; BUTLER, M.; SNOOK, C. Language and Tool Support for Class and State Machine Refinement in UML-B. In: FM2009 - 16TH INTERNATIONAL SYMPOSIUM ON FORMAL METHODS, 2009. **Proceedings. . .** Springer, 2009. n.LNCS 5, p.579–595.

SAKSENA, M.; WIBLING, O.; JONSSON, B. Graph Grammar Modeling and Verification of Ad Hoc Routing Protocols. In: TACAS, 2008. **Proceedings. . .** Springer, 2008. p.18–32. (Lecture Notes in Computer Science, v.4963).

SALAMAH, S.; GATES, A. Q.; KREINOVICH, V.; ROACH, S. Verification of Automatically Generated Pattern-Based LTL Specifications. In: HASE '07: PROCEEDINGS OF THE 10TH IEEE HIGH ASSURANCE SYSTEMS ENGINEERING SYMPOSIUM, 2007, Washington, USA. **Proceedings. . .** IEEE Comp. Soc., 2007. p.341–348.

SCHFüRR, A. Programmed graph replacement systems. **Handbook of graph grammars and computing by graph transformation: volume I. foundations**, River Edge, NJ, USA, p.479–546, 1997.

SCHüRR, A.; WINTER, A. J.; ZüNDORF, A. The PROGRES approach: language and environment. **Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools**, River Edge, NJ, USA, p.487–550, 1999.

SMITH, R. L.; AVRUNIN, G. S.; CLARKE, L. A.; OSTERWEIL, L. J. PROPEL: an approach supporting property elucidation. **Software Engineering, International Conference on**, Los Alamitos, CA, USA, v.0, p.11, 2002.

SNOOK, C.; BUTLER, M. UML-B and Event-B: an integration of languages and tools. In: THE IASTED INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING - SE2008, 2008. **Proceedings. . .** [S.l.: s.n.], 2008.

SONG, G.; ZHANG, K.; WONG, R. K.; KONG, J. Management of Web Data Models Based on Graph Transformation. In: WI ’04: PROCEEDINGS OF THE 2004 IEEE/WIC/ACM INTERNATIONAL CONFERENCE ON WEB INTELLIGENCE, 2004, Washington, DC, USA. **Proceedings. . .** IEEE Computer Society, 2004. p.398–404.

STRECKER, M. Modeling and Verifying Graph Transformations in Proof Assistants. **Electronic Notes in Theoretical Computer Science**, Amsterdam, The Netherlands, The Netherlands, v.203, n.1, p.135–148, 2008.

TAMáS MéSZáROS, I. M.; MEZEI, G. AntWorld Simulation Case Study Modeled by Tiger. In: INTERNATIONAL WORKSHOP ON GRAPH-BASED TOOLS: THE CONTEST, 4., 2008. **Proceedings. . .** [S.l.: s.n.], 2008.

TANENBAUM, A. **Computer Networks**. [S.l.]: Prentice Hall, 2002.

TEJ, H.; WOLFF, B. A Corrected Failure Divergence Model for CSP in Isabelle/HOL. In: FME ’97: PROCEEDINGS OF THE 4TH INTERNATIONAL SYMPOSIUM OF FORMAL METHODS EUROPE ON INDUSTRIAL APPLICATIONS AND STRENGTHENED FOUNDATIONS OF FORMAL METHODS, 1997, London, UK. **Proceedings. . .** Springer-Verlag, 1997. p.318–337.

THE AGG System. Last accessed March 2010, Available at http://user.cs.tu-berlin.de/ gragra/agg/.

The MathWorks. **Stateflow and stateflow coder, user’s guide**. Available at http://www.mathworks.com/products/stateflow/.

TIGER Project. Last accessed March 2010, Available at http://user.cs.tu-berlin.de/ tiger-prj/.

VICTOR, B.; MOLLER, F. The Mobility Workbench - A Tool for the pi-Calculus. In: CAV ’94: PROCEEDINGS OF THE 6TH INTERNATIONAL CONFERENCE ON COMPUTER AIDED VERIFICATION, 1994, London, UK. **Proceedings. . .** Springer-Verlag, 1994. p.428–440.

WANKMÜLLER, F. Application of Graph Grammars in Music Composing Systems. In: GRAPH-GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 1986. **Proceedings. . .** Springer, 1986. p.580–592. (Lecture Notes in Computer Science, v.291).

WOODCOCK, J. C. P.; CAVALCANTI, A. L. C. A Concurrent Language for Refinement. In: IWFM’01: 5TH IRISH WORKSHOP IN FORMAL METHODS, 2001, Dublin, Ireland. **Proceedings. . .** [S.l.: s.n.], 2001. (BCS Electronic Workshops in Computing).

WOODCOCK, J.; CAVALCANTI, A. The Semantics of Circus. In: ZB ’02: PROCEEDINGS OF THE 2ND INTERNATIONAL CONFERENCE OF B AND Z USERS ON FORMAL SPECIFICATION AND DEVELOPMENT IN Z AND B, 2002, London, UK. **Proceedings. . .** Springer-Verlag, 2002. p.184–203.

WOODCOCK, J.; DAVIES, J. **Using Z**: specification, refinement, and proof. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

WOODCOCK, J.; LARSEN, P. G.; BICARREGUI, J.; FITZGERALD, J. Formal methods: practice and experience. **ACM Comput. Surv.**, New York, NY, USA, v.41, n.4, p.1–36, 2009.

YANG, J.; EVANS, D. Dynamically inferring temporal properties. In: PASTE '04: PROCEEDINGS OF THE 5TH ACM SIGPLAN-SIGSOFT WORKSHOP ON PROGRAM ANALYSIS FOR SOFTWARE TOOLS AND ENGINEERING, 2004, New York, NY, USA. **Proceedings...** ACM, 2004. p.23–28.

YU, J.; MANH, T. P.; HAN, J.; JIN, Y.; HAN, Y.; WANG, J. Pattern Based Property Specification and Verification for Service Composition. In: IN: PROCEEDINGS OF 7TH INTERNATIONAL CONFERENCE ON WEB INFORMATION SYSTEMS ENGINEERING (WISE, 2006. **Proceedings...** [S.l.: s.n.], 2006. p.156–168.

ZEYDA, F.; CAVALCANTI, A. Mechanical Reasoning about Families of UTP Theories. In: SBMF 2008, Brazilian Symposium on Formal Methods, 2008. **Proceedings...** [S.l.: s.n.], 2008. p.145–160. Best paper award.

ZEYDA, F.; CAVALCANTI, A. Encoding Circus Programs in ProofPowerZ. In: 2009. **Proceedings...** Springer, 2009. (Lecture Notes in Computer Science). Awaiting publication.

ZHANG, K.-B.; ZHANG, K.; ORGUN, M. A. Using graph grammar to implement global layout for a visual programming language generation system. In: VIP '01: PROCEEDINGS OF THE PAN-SYDNEY AREA WORKSHOP ON VISUAL INFORMATION PROCESSING, 2001, Darlinghurst, Australia, Australia. **Proceedings...** Australian Computer Society: Inc., 2001. p.115–121.

APPENDIX A ALGEBRAIC SPECIFICATIONS

A.1 Basic Concepts of Algebraic Specifications

Definition 40 (Signature). *A signature $SIG = (S, OP)$ consists of a set S of sorts and a set OP of constant and operations symbols. The set OP is the union of pairwise disjoint subsets:*

- *K_s, set of constant symbols of sorts $s \in S$.*
- *$OP_{w,s}$, set of operation symbols with argument sorts $w \in S^+$ and range sort $s \in S$, for all $s \in S$ and $w \in S^+$.*

Definition 41 (Algebra). *An algebra $A = (S_A, OP_A)$ of a signature $SIG = (S, OP)$, also called SIG-Algebra, is given by two families $S_A = (A_s)_{s \in S}$ and $OP = (N_A)_{N \in OP}$ where*

1. *A_s are sets for all $s \in S$, called base sets or carrier sets of A.*
2. *N_A are elements $N_A \in A_s$ for all constant symbols $N \in K_s$ i. e. $N :\to s$ e $s \in S$, called constants of A.*
3. *$N_A : A_{s1} \times A_{s2} \times \cdots \times A_{sn} \to A_s$ are functions for all operation symbols $N \in OP_{s1 \ldots sn,s}$ (i.e. $N : s1 \ldots sn \to s$) and $s1 \ldots sn \in S^+$, $s \in S$, called operations of A, where "$\times$" denotes the cartesian product of sets.*

Definition 42 (Variables and Terms). *Let $SIG = (S, OP)$ be a signature and X_s for each $s \in S$ a set, called set of variables of sort s. We assume that these sets X_s are pairwise disjoint and also disjoint with OP. The union $X = \bigcup_{s \in S} X_s$ is called **set of variables with respect to SIG**.*

*The sets $T_{OP,s}(X)$ of **terms of sort s** is inductively defined by:*

1. *$X_s \cup K_s \subseteq T_{OP,s}(X)$ where K_s is the set of constant symbols of sort s.*
2. *$N(t_1, \ldots, t_n) \in T_{OP,s}(X)$ for all operation symbols $N \in OP$ with $N : s_1 \ldots s_n \to s$ and all terms $t_1 \in T_{OP,s_1}, \ldots, t_n \in T_{OP,s_n}$.*
3. *There are no further terms of sort $s \in T_{OP,s}(X)$.*

*The set $T_{OP,s}$ of terms without variables of sorts s, also called **ground terms** of sort s, is defined for the empty set $X = \varnothing$ of variables by: $T_{OP,s} = T_{OP,s}(\varnothing)$*

The set of terms $T_{OP}(X)$ and the set of terms without variables T_{OP} are defined by: $T_{OP}(X) = \bigcup_{s \in S} T_{OP,s}(X)$ and $T_{OP} = \bigcup_{s \in S} T_{OP,s}$

Definition 43 (Evaluation of Terms)**.** *Let T_{OP} be the set of terms for a signature $SIG = (S, OP)$ and A a SIG-algebra. The **evaluation** $eval : T_{OP} \to A$ is recursively defined by:*

(i) $eval(N) = N_A$ for all constant symbols $N \in K$

(ii) $eval(N(t_1, \dots, t_n)) =$
$N_A(eval(t_1), \dots, eval(t_n))$ for all $N(t_1, \dots, t_n) \in T_{OP}$.

*Given a set of variables X for $SIG = (S, OP)$ and an **assignment** $asg : X \to A$ with $asg(x) \in A$ for $x \in X_s$ and $s \in S$. The **extended assignment** $\overline{asg} : T_{OP}(X) \to A$ of the assignment $asg : X \to A$ is recursively defined by:*

(i) $\overline{asg}(x) = asg(x)$ for all variables $x \in X$
$\overline{asg}(N) = N_A$ for all constant symbols $N \in K$

(ii) $\overline{asg}(N(t_1, \dots, t_n)) =$
$N_A(\overline{asg}(t_1), \dots, \overline{asg}(t_n))$ for all $N(t_1, \dots, t_n) \in T_{OP}(X)$.

Definition 44 (Equations and Validity)**.** *Given a signature $SIG = (S, OP)$ and variables X with respect to SIG. A triple $e = (X, L, R)$ with $L, R \in T_{OP,s}(X)$ for some $s \in S$ is called an **equation** of sort s with respect to SIG. The equation $e = (X, L, R)$ is called **valid** in a SIG-algebra A if for all assignments $asg : X \to A$ we have $\overline{asg}(L) = \overline{asg}(R)$ where $\overline{asg}$ is the extended assignment of asg. If e is valid in A we also say that A **satisfies** e.*

***Ground equations** are equations $e = (X, L, R)$ with $X = \varnothing$. In this case L and R are ground terms.*

Definition 45 (Derivation of Terms)**.** *Given a set E of equations for a signature $SIG = (S, OP)$ with a fixed set of variables $X = X_e$ for each equation e. $(L, R) \in E$ defines two **substitution Trules**:*

(1) $L \Rightarrow R$ (R – L – rule)
(2) $R \Rightarrow L$ (L – R – rule)

*A **Trule** $t_1 \Rightarrow t_2$ **is applicable** to a term $t \in T_{OP}(X)$ if there is an assignment $asg : X \to T_{OP}(X)$ with extension $\overline{asg} : T_{OP}(X) \to T_{OP}(X)$ such that we have for $\overline{t_1} = \overline{asg}(t_1)$ and $\overline{t_2} = \overline{asg}(t_2)$: $\overline{t_1}$ is a subterm of t.*

*The replacement of $\overline{t_1}$ by $\overline{t_2}$ in t yields a term t' and is denoted by $t' = t(\overline{t_1}/\overline{t_2})$. In this case we write $t \Rightarrow t'$, called **direct derivation from** t **to** t' **via** E, using Trule $t1 \Rightarrow t_2$ and assignment asg.*

*A sequence of $n \geq 0$ direct derivations $t_0 \Rightarrow t_1 \Rightarrow \cdots \Rightarrow t_n$ with $t = t_0$ and $t' = t_n$ written as $t \overset{*}{\Rightarrow} t'$, is called **derivation from** t **to** t' **via** E and $e' = (t, t')$ is called **derived equation** from E with fixed X. The derivation $t \overset{*}{\Rightarrow} t'$ is **correct** with respect to a SIG-algebra A if we have for each assignment $asg : X \to A$, $\overline{asg}(t) = \overline{asg}(t')$.*

Definition 46 (Specification and SPEC-algebra)**.** *A **specification** $SPEC = (S, OP, E)$ consists of a signature $SIG = (S, OP)$ and a set E of equations e with respect to SIG. An **algebra** A of the specification $SPEC$, short **SPEC-algebra**, is an algebra A of the signature SIG which satisfies all equations in E.*

Definition 47 (Homomorphism)**.** *Let A and B be algebras of the same signature $SIG = (S, OP)$ or specification $SPEC = (S, OP, E)$. A **homomorphism** $f : A \to B$, also called **SIG- or SPEC-homomorphism**, is a family of functions*

$$f_s : A_s \to B_s \quad \text{for } s \in S$$

such that for each constant symbol $N :\to s$ in OP and $s \in S$

$$f_s(N_A) = N_B$$

and for each operation symbol $N : s_1 \dots s_n \to s$ in OP and all $a_i \in A_{s_i}$, for $i = 1, \dots, n$

$$f_s(N_A(a_1, \dots, a_n)) = N_B(f_{s_1}(a_1), \dots, f_{s_n}(a_n))$$

***Alg(SIG)** denotes the category of all SIG-algebras and SIG-homomorphisms. $\mathcal{U}$ denotes the forgetful functor from **Alg(SIG)** to **Set** yielding the disjoint union of carrier sets (and homomorphisms).*

Definition 48 (Congruence on Ground Terms)**.** *Given a specification $SPEC = (S, OP, E)$ the relation $\equiv$ on ground terms defined for all $t_1, t_2 \in T_{OP}$ by*

$t_1 \equiv t_2$ if and only if $eval_A(t_1) = eval_A(t_2)$ for all $\mathrm{SPEC} - \mathrm{algebras}\ A$

*is called **congruence on ground terms**.*

It satisfies the following conditions for all $t_1, t_2, t_3 \in T_{OP}$:

1. $t_1 \equiv t_1$. (reflexivity)
2. $t_1 \equiv t_2$ implies $t_2 \equiv t_1$. (symmetry)
3. $t_1 \equiv t_2$ and $t_2 \equiv t_3$ implies $t_1 \equiv t_3$. (transitivity)
4. $t_1 \equiv t_1', \dots, t_n \equiv t_n'$ implies $N(t_1, \dots, t_n) \equiv N(t_1', \dots, t_n')$ (congruence)
   for all operation symbols $N : s_1 \dots s_n \to s$ in OP with $n \geq 1$ and all ground terms t_i, t_i' of sort s_i for $i = 1, \dots, n$.
5. Each derivation $t_1 \overset{*}{\Rightarrow} t_2$ via E between ground terms $t_1, t_2 \in T_{OP}$ implies $t_1 \equiv t_2$.
6. If there is a $\mathrm{SPEC} - \mathrm{algebra}$ A with $eval_A(t_1) \neq eval_A(t_2)$ for some ground terms $t_1, t_2 \in T_{OP}$ then we have $t_1 \not\equiv t_2$.

Definition 49 (Algebra of Terms)**.** *The algebra (S_T, OP_T) with*

(i) $S_T = (T_{OP,s})_{s \in S}$ *as the family of base sets.*

(ii) $N_T = N$ *as constant for* $N :\to s$.

(iii) $N_T : T_{OP,s_1}(X) \times \cdots \times T_{OP,s_n}(X) \to T_{OP,s}(X)$*defined by*

$$N_T(t_1, \dots, t_n) = N(t_1, \dots, t_n)$$

for $N : s_1 \dots s_n \to s$ and $t_i \in T_{OP,s_i}(X)$, $i = 1, \dots, n$, as the operations.

*is called the **algebra of terms with respect to SIG and X**, or simply the **term algebra**.*

Definition 50 (Quotient Term Algebra)**.** *Given a specification $SPEC = (S, OP, E)$ the **quotient term algebra** $T_{SPEC} = ((Q_s)_{s \in S'}, (N_Q)_{N \in OP})$ is defined by:*

1. *For each $s \in S$ we have a base set*

   $$Q_s = \{[t]/t \in T_{OP,s}\}$$

   where the congruence class $[t]$ is defined by:

   $$[t] = \{t'/t' \equiv t\}.$$

2. *For each constant symbol $N :\to s$ in OP the constant N_Q is the congruence class generated by N: $N_Q = [N]$*

3. *For each operation symbol $N : s_1 \dots s_n \to s$ in OP the operation $N_Q : Q_{s_1} \times \cdots \times Q_{s_n} \to Q_s$ is defined by*

   $$N_Q([t_1], \dots, [t_n]) = [N(t_1, \dots, t_n)]$$

   for all terms t_i of sort s_i and all $i = 1, \dots, n$.

APPENDIX B   TOKEN RING SPECIFICATION

Next we describe the Event-B specification of the Token Ring protocol. This model was based on the relational structure depicted in Example 4. The model generated 86 proof obligations with 57 of them proved automatically. It is also important to notice that the great majority of proof obligations discharged by interactive proof involved just the direct execution of an event-b prover, the simple addition of hypothesis or the instantiation of universal quantifiers.

B.1   Event-B Context of Token Ring

An Event-B Specification of ctx_trAll
Creation Date: 8 Mar 2010 @ 08:23:33 PM

CONTEXT ctx_trAll
SETS

```
    V_GG     // (Domain) Vertices names – V_GG ⊆ ℕ
    E_GG     // (Domain) Edges names – E_GG ⊆ ℕ
    vertT     // (Type Graph T) Vertices
    edgeT     // (Type Graph T) Types of edges
    vertL1     // (Rule 1) Left Graph L1 – Vertices
    edgeL1     // (Rule 1) Left Graph L1 – Edges
    vertR1     // (Rule 1) Right Graph R1 – Vertices
    edgeR1     // (Rule 1) Right Graph R1 – Edges
    vertL2     // (Rule 2) Left Graph L2 – Vertices
    edgeL2     // (Rule 2) Left Graph L2 – Edges
    vertR2     // (Rule 2) Right Graph R2 – Vertices
    edgeR2     // (Rule 2) Right Graph R2 – Edges
    vertL3     // (Rule 3) Left Graph L3 – Vertices
    edgeL3     // (Rule 3) Left Graph L3 – Edges
    vertR3     // (Rule 3) Right Graph R3 – Vertices
    edgeR3     // (Rule 3) Right Graph R3 – Edges
    vertL4     // (Rule 4) Left Graph L4 – Vertices
    edgeL4     // (Rule 4) Left Graph L4 – Edges
    vertR4     // (Rule 4) Right Graph R4 – Vertices
    edgeR4    // (Rule 4) Right Graph R4 – Edges
    vertL5    // (Rule 5) Left Graph L5 – Vertices
    edgeL5    // (Rule 5) Left Graph L5 – Edges
    vertR5    // (Rule 5) Right Graph R5 – Vertices
    edgeR5    // (Rule 5) Right Graph R5 – Edges
CONSTANTS
    Node    // Type of node
    Nxt    // Type of edge
    Tok    // Type of edge
    Msg    // Type of edge
    Stb    // Type of edge
    Act    // Type of edge
    N11    // (Rule 1) Vertex name – N11 ∈ vertL1
    N12    // (Rule 1) Vertex name – N12 ∈ vertL1
    N13    // (Rule 1) Vertex name – N13 ∈ vertR1
    N14    // (Rule 1) Vertex name – N14 ∈ vertR1
    N21    // (Rule 2) Vertex name – N21 ∈ vertL2
    N22    // (Rule 2) Vertex name – N22 ∈ vertL2
    N23    // (Rule 2) Vertex name – N23 ∈ vertR2
    N24    // (Rule 2) Vertex name – N24 ∈ vertR2
    N31    // (Rule 3) Vertex name – N31 ∈ vertL3
    N32    // (Rule 3) Vertex name – N32 ∈ vertL3
    N33    // (Rule 3) Vertex name – N33 ∈ vertR3
    N34    // (Rule 3) Vertex name – N34 ∈ vertR3
    N41    // (Rule 4) Vertex name – N41 ∈ vertL4
    N42    // (Rule 4) Vertex name – N42 ∈ vertL4
    N43    // (Rule 4) Vertex name – N43 ∈ vertR4
    N44    // (Rule 4) Vertex name – N44 ∈ vertR4
    N51    // (Rule 5) Vertex name – N51 ∈ vertL5
    N52    // (Rule 5) Vertex name – N52 ∈ vertL5
    N53    // (Rule 5) Vertex name – N53 ∈ vertR5
    N54    // (Rule 5) Vertex name – N54 ∈ vertR5
    N55    // (Rule 5) Vertex name – N55 ∈ vertR5
    Tok11    // (Rule 1) Edge name – Tok11 ∈ edgeL1
    Stb11    // (Rule 1) Edge name – Stb11 ∈ edgeL1
    Nxt11    // (Rule 1) Edge name – Nxt11 ∈ edgeL1
    Tok12    // (Rule 1) Edge name – Tok12 ∈ edgeR1
    Nxt12    // (Rule 1) Edge name – Nxt12 ∈ edgeR1
    Act11    // (Rule 1) Edge name – Act11 ∈ edgeR1
    Msg11    // (Rule 1) Edge name – Msg11 ∈ edgeR1
    Tok21    // (Rule 2) Edge name – Tok21 ∈ edgeL2
    Stb21    // (Rule 2) Edge name – Stb21 ∈ edgeL2
```

```
Nxt21     // (Rule 2) Edge name – Nxt21 ∈ edgeL2
Tok22     // (Rule 2) Edge name – Tok22 ∈ edgeR2
Stb22     // (Rule 2) Edge name – Stb22 ∈ edgeR2
Nxt22     // (Rule 2) Edge name – Nxt22 ∈ edgeR2
Stb31     // (Rule 3) Edge name – Stb31 ∈ edgeL3
Nxt31     // (Rule 3) Edge name – Nxt31 ∈ edgeL3
Msg31     // (Rule 3) Edge name – Msg31 ∈ edgeL3
Nxt32     // (Rule 3) Edge name – Nxt32 ∈ edgeR3
Stb32     // (Rule 3) Edge name – Stb32 ∈ edgeR3
Msg32     // (Rule 3) Edge name – Msg32 ∈ edgeR3
Msg41     // (Rule 4) Edge name – Msg41 ∈ edgeL4
Nxt41     // (Rule 4) Edge name – Nxt41 ∈ edgeL4
Act41     // (Rule 4) Edge name – Act41 ∈ edgeL4
Tok41     // (Rule 4) Edge name – Tok41 ∈ edgeL4
Tok42     // (Rule 4) Edge name – Tok42 ∈ edgeR4
Nxt42     // (Rule 4) Edge name – Nxt42 ∈ edgeR4
Stb42     // (Rule 4) Edge name – Stb42 ∈ edgeR4
Stb51     // (Rule 5) Edge name – Stb51 ∈ edgeL5
Nxt51     // (Rule 5) Edge name – Nxt51 ∈ edgeR5
Nxt52     // (Rule 5) Edge name – Nxt52 ∈ edgeR5
Nxt53     // (Rule 5) Edge name – Nxt53 ∈ edgeR5
sourceL1     // (Rule 1) function sourceL1 – returns the source of an edge of L1
targetL1     // (Rule 1) function targetL1 – returns the target of an edge of L1
tL1_V     // (Rule 1) Typing left vertices, tL1_V
tL1_E     // (Rule 1) Typing left edges, tL1_E
sourceR1     // (Rule 1) function sourceR1 – returns the source of an edge of R1
targetR1     // (Rule 1) function targetR1 – returns the target of an edge of R1
tR1_V     // (Rule 1) Typing right vertices, tR1_V
tR1_E     // (Rule 1) Typing right edges, tR1_E
alpha1V    // (Rule 1) Relational Rule alpha1: mapping vertices
alpha1E    // (Rule 1) Relational Rule alpha1: mapping edges
sourceL2     // (Rule 2) function sourceL2 – returns the source of an edge of L2
targetL2     // (Rule 2) function targetL2 – returns the target of an edge of L2
tL2_V     // (Rule 2) Typing left vertices, tL2_V
tL2_E     // (Rule 2) Typing left edges, tL2_E
sourceR2     // (Rule 2) function sourceR2 – returns the source of an edge of R2
targetR2     // (Rule 2) function targetL2 – returns the target of an edge of R2
tR2_V     // (Rule 2) Typing right vertices, tR2_V
tR2_E     // (Rule 2) Typing right edges, tR2_E
alpha2V    // (Rule 2) Relational Rule alpha2: mapping vertices
alpha2E    // (Rule 2) Relational Rule alpha2: mapping edges
sourceL3     // (Rule 3) function sourceL3 – returns the source of an edge of L3
```

targetL3 // (Rule 3) function targetL3 – returns the target of an edge of L3
tL3_V // (Rule 3) Typing left vertices, tL3_V
tL3_E // (Rule 3) Typing left edges, tL3_E
sourceR3 // (Rule 3) function sourceR3 – returns the source of an edge of R3
targetR3 // (Rule 3) function targetR3 – returns the target of an edge of R3
tR3_V // (Rule 3) Typing right vertices, tR3_V
tR3_E // (Rule 3) Typing right edges, tR3_E
alpha3V // (Rule 3) Relational Rule alpha3: mapping vertices
alpha3E // (Rule 3) Relational Rule alpha3: mapping edges
sourceL4 // (Rule 4) function sourceL4 – returns the source of an edge of L4
targetL4 // (Rule 4) function targetL4 – returns the target of an edge of L4
tL4_V // (Rule 4) Typing left vertices, tL4_V
tL4_E // (Rule 4) Typing left edges, tL
sourceR4 // (Rule 4) function sourceR4 – returns the source of an edge of R4
targetR4 // (Rule 4) function targetR4 – returns the target of an edge of R4
tR4_V // (Rule 4) Typing right vertices, tR4_V
tR4_E // (Rule 4) Typing right edges, tR4_E
alpha4V // (Rule 4) Relational Rule alpha4: mapping vertices
alpha4E // (Rule 4) Relational Rule alpha4: mapping edges
sourceL5 // (Rule 5) function sourceL5 – returns the source of an edge of L5
targetL5 // (Rule 5) function targetL5 – returns the target of an edge of L5
tL5_V // (Rule 5) Typing left vertices, tL5_V
tL5_E // (Rule 5) Typing left edges, tL5_E
sourceR5 // (Rule 5) function sourceR5 – returns the source of an edge of R5
targetR5 // (Rule 5) function targetR5 – returns the target of an edge of R5
tR5_V // (Rule 5) Typing right vertices, tR5_V
tR5_E // (Rule 5) Typing right edges, tR5_E
alpha5V // (Rule 5) Relational Rule alpha5: mapping vertices
alpha5E // (Rule 5) Relational Rule alpha5: mapping edges
sourceT // function sourceT – returns the source of an edge of T
targetT // function targetT – returns the target of an edge of T

AXIOMS

axm_vertT : $partition(vertT, \{Node\})$
// (Type Graph T) vertT = { Node}

axm_edgeT : $partition(edgeT, \{Nxt\}, \{Tok\}, \{Msg\}, \{Stb\}, \{Act\})$
// (Type Graph T) edgeT = { Nxt, Tok, Msg, Stb, Act} inc

axm_srcTtype : $sourceT \in edgeT \rightarrow vertT$
// (Type Graph T) function sourceT

axn_srcTdef : $partition(sourceT, \{Nxt \mapsto Node\}, \{Tok \mapsto Node\}, \{Msg \mapsto Node\}, \{Stb \mapsto Node\}, \{Act \mapsto Node\})$
// (Type Graph T) function sourceT

axm_tgtTtype : $targetT \in edgeT \rightarrow vertT$
// (Type Graph T) function targetT

axm_tgtTdef : $partition(targetT, \{Nxt \mapsto Node\}, \{Tok \mapsto Node\}, \{Msg \mapsto Node\}, \{Stb \mapsto Node\}, \{Act \mapsto Node\})$
// (Type Graph T) function targetT

axm_vertL1 : $partition(vertL1, \{N11\}, \{N12\})$
// (Rule 1) Left Graph L1 – Vertices

axm_edgeL1 : $partition(edgeL1, \{Tok11\}, \{Stb11\}, \{Nxt11\})$
// (Rule 1) Left Graph L1 – Edges names

axm_srcL1type : $sourceL1 \in edgeL1 \rightarrow vertL1$
// (Rule 1) function sourceL1

axm_srcL1def : $partition(sourceL1, \{Tok11 \mapsto N11\}, \{Stb11 \mapsto N11\}, \{Nxt11 \mapsto N11\})$
// (Rule 1) function sourceL1

axm_tgtL1type : $targetL1 \in edgeL1 \rightarrow vertL1$
// (Rule 1) function targetL1

axm_tgtL1def : $partition(targetL1, \{Tok11 \mapsto N11\}, \{Stb11 \mapsto N11\}, \{Nxt11 \mapsto N12\})$
// (Rule 1) function targetL1

axm_tL1_V : $tL1\_V \in vertL1 \rightarrow vertT$
// (Rule 1) Typing left vertices, tL1_V

axm_tL1_V_def : $partition(tL1\_V, \{N11 \mapsto Node\}, \{N12 \mapsto Node\})$
// (Rule 1) Typing left vertices, tL1_V

axm_tL1_E : $tL1\_E \in edgeL1 \rightarrow edgeT$
// (Rule 1) Typing left edges, tL1_E

axm_tL1_E_def : $partition(tL1\_E, \{Tok11 \mapsto Tok\}, \{Stb11 \mapsto Stb\}, \{Nxt11 \mapsto Nxt\})$
// (Rule 1) Typing left edges, tL1_E

axm_vertR1 : $partition(vertR1, \{N13\}, \{N14\})$
// (Rule 1) Right Graph R1 – Vertices

axm_edgeR1 : $partition(edgeR1, \{Tok12\}, \{Act11\}, \{Nxt12\}, \{Msg11\})$
(Rule 1) Right Graph R1 – Edges names

axm_srcR1type : $sourceR1 \in edgeR1 \rightarrow vertR1$
// (Rule 1) function sourceR1

axm_srcR1def : $partition(sourceR1, \{Tok12 \mapsto N13\}, \{Act11 \mapsto N13\}, \{Nxt12 \mapsto N13\}, \{Msg11 \mapsto N14\})$
// (Rule 1) function sourceR1

axm_tgtR1type : $targetR1 \in edgeR1 \rightarrow vertR1$
// (Rule 1) function targetR1

axm_tgtR1def : $partition(targetR1, \{Tok12 \mapsto N13\}, \{Act11 \mapsto N13\}, \{Nxt12 \mapsto N14\}, \{Msg11 \mapsto N14\})$
// (Rule 1) function targetR1

axm_tR1_V : $tR1\_V \in vertR1 \rightarrow vertT$
// (Rule 1) Typing right vertices, tR1_V

axm_tR1_V_def : $partition(tR1\_V, \{N13 \mapsto Node\}, \{N14 \mapsto Node\})$
// (Rule 1) Typing right vertices, tR1_V

axm_tR1_E : $tR1\_E \in edgeR1 \rightarrow edgeT$
// (Rule 1) Typing right edges, tR1_E

axm_tR1_E_def : $partition(tR1\_E, \{Tok12 \mapsto Tok\}, \{Act11 \mapsto Act\}, \{Nxt12 \mapsto Nxt\}, \{Msg11 \mapsto Msg\})$
// (Rule 1) Typing right edges, tR1_E

axm_alpha1V : $alpha1V \in vertL1 \twoheadrightarrow vertR1$
// (Rule 1) Relational Rule alpha1: mapping vertices

axm_alpha1V_def : $partition(alpha1V, \{N11 \mapsto N13\}, \{N12 \mapsto N14\})$
// (Rule 1) Relational Rule alpha1: mapping vertices

axm_alpha1E : $alpha1E \in edgeL1 \twoheadrightarrow edgeR1$
// (Rule 1) Relational Rule alpha1: mapping edges

axm_alpha1E_def : $partition(alpha1E, \{Tok11 \mapsto Tok12\}, \{Nxt11 \mapsto Nxt12\})$

// (Rule 1) Relational Rule alpha1: mapping edges

axm_vertL2 : $partition(vertL2, \{N21\}, \{N22\})$
// (Rule 2) Left Graph L2 – Vertices

axm_edgeL2 : $partition(edgeL2, \{Tok21\}, \{Stb21\}, \{Nxt21\})$
// (Rule 2) Left Graph L2 – Edges names

axm_srcL2type : $sourceL2 \in edgeL2 \rightarrow vertL2$
// (Rule 2) function sourceL2

axn_srcL2def : $partition(sourceL2, \{Tok21 \mapsto N21\}, \{Stb21 \mapsto N21\}, \{Nxt21 \mapsto N21\})$
// (Rule 2) function sourceL2

axm_tgtL2type : $targetL2 \in edgeL2 \rightarrow vertL2$
// (Rule 2) function targetL2

axn_tgtL2def : $partition(targetL2, \{Tok21 \mapsto N21\}, \{Stb21 \mapsto N21\}, \{Nxt21 \mapsto N22\})$
// (Rule 2) function targetL2

axm_tL2_V : $tL2\_V \in vertL2 \rightarrow vertT$
// (Rule 2) Typing left vertices, tL2_V

axm_tL2_V_def : $partition(tL2\_V, \{N21 \mapsto Node\}, \{N22 \mapsto Node\})$
// (Rule 2) Typing left vertices, tL2_V

axm_tL2_E : $tL2\_E \in edgeL2 \rightarrow edgeT$
// (Rule 2) Typing left edges, tL2_E

axm_tL2_E_def : $partition(tL2\_E, \{Tok21 \mapsto Tok\}, \{Stb21 \mapsto Stb\}, \{Nxt21 \mapsto Nxt\})$
// (Rule 2) Typing left edges, tL2_E

axm_vertR2 : $partition(vertR2, \{N23\}, \{N24\})$
// (Rule 2) Right Graph R2 – Vertices

axm_edgeR2 : $partition(edgeR2, \{Tok22\}, \{Stb22\}, \{Nxt22\})$
// (Rule 2) Right Graph R2 – Edges names

axm_srcR2type : $sourceR2 \in edgeR2 \rightarrow vertR2$
// (Rule 2) function sourceR2

axn_srcR2def : $partition(sourceR2, \{Tok22 \mapsto N24\}, \{Stb22 \mapsto N23\}, \{Nxt22 \mapsto N23\})$
// (Rule 2) function sourceL2

axm_tgtR2type : $targetR2 \in edgeR2 \rightarrow vertR2$
// (Rule 2) function targetR2

axn_tgtR2def : $partition(targetR2, \{Tok22 \mapsto N24\}, \{Stb22 \mapsto N23\}, \{Nxt22 \mapsto N24\})$
// (Rule 2) function targetR2

axm_tR2_V : $tR2\_V \in vertR2 \rightarrow vertT$
// (Rule 2) Typing right vertices, tR2_V

axm_tR2_V_def : $partition(tR2\_V, \{N23 \mapsto Node\}, \{N24 \mapsto Node\})$
// (Rule 2) Typing right vertices, tR2_V

axm_tR2_E : $tR2\_E \in edgeR2 \rightarrow edgeT$
// (Rule 2) Typing right edges, tR2_E

axm_tR2_E_def : $partition(tR2\_E, \{Stb22 \mapsto Stb\}, \{Tok22 \mapsto Tok\}, \{Nxt22 \mapsto Nxt\})$
// (Rule 2) Typing right edges, tR2_E

axm_alpha2V : $alpha2V \in vertL2 \rightarrowtail\!\!\!\!\!\twoheadrightarrow vertR2$
// (Rule 2) Relational Rule alpha2: mapping vertices

axm_alpha2V_def : $partition(alpha2V, \{N21 \mapsto N23\}, \{N22 \mapsto N24\})$
// (Rule 2) Relational Rule alpha2: mapping vertices

axm_alpha2E : $alpha2E \in edgeL2 \rightarrowtail\!\!\!\!\!\twoheadrightarrow edgeR2$
// (Rule 2) Relational Rule alpha2: mapping edges

axm_alpha2E_def : $partition(alpha2E, \{Stb21 \mapsto Stb22\}, \{Nxt21 \mapsto Nxt22\})$

// (Rule 2) Relational Rule alpha2: mapping edges

axm_vertL3 : $partition(vertL3, \{N31\}, \{N32\})$
// (Rule 3) Left Graph L3 – Vertices

axm_edgeL3 : $partition(edgeL3, \{Stb31\}, \{Msg31\}, \{Nxt31\})$
// (Rule 3) Left Graph L3 – Edges names

axm_srcL3type : $sourceL3 \in edgeL3 \rightarrow vertL3$
// (Rule 3) function sourceL3

axn_srcL3def : $partition(sourceL3, \{Msg31 \mapsto N31\}, \{Stb31 \mapsto N31\}, \{Nxt31 \mapsto N31\})$
// (Rule 3) function sourceL3

axm_tgtL3type : $targetL3 \in edgeL3 \rightarrow vertL3$
// (Rule 3) function targetL3

axn_tgtL3def : $partition(targetL3, \{Msg31 \mapsto N31\}, \{Stb31 \mapsto N31\}, \{Nxt31 \mapsto N32\})$
// (Rule 3) function targetL3

axm_tL3_V : $tL3\_V \in vertL3 \rightarrow vertT$
// (Rule 3) Typing left vertices, tL3_V

axm_tL3_V_def : $partition(tL3\_V, \{N31 \mapsto Node\}, \{N32 \mapsto Node\})$
// (Rule 3) Typing left vertices, tL3_V

axm_tL3_E : $tL3\_E \in edgeL3 \rightarrow edgeT$
// (Rule 3) Typing left edges, tL3_E

axm_tL3_E_def : $partition(tL3\_E, \{Stb31 \mapsto Stb\}, \{Msg31 \mapsto Msg\}, \{Nxt31 \mapsto Nxt\})$
// (Rule 3) Typing left edges, tL3_E

axm_vertR3 : $partition(vertR3, \{N33\}, \{N34\})$
// (Rule 3) Right Graph R3 – Vertices

axm_edgeR3 : $partition(edgeR3, \{Stb32\}, \{Nxt32\}, \{Msg32\})$
// (Rule 3) Right Graph R3 – Edges names

axm_srcR3type : $sourceR3 \in edgeR3 \rightarrow vertR3$
// (Rule 3) function sourceR3

axn_srcR3def : $partition(sourceR3, \{Msg32 \mapsto N34\}, \{Stb32 \mapsto N33\}, \{Nxt32 \mapsto N33\})$
// (Rule 3) function sourceR3

axm_tgtR3type : $targetR3 \in edgeR3 \rightarrow vertR3$
// (Rule 3) function targetR3

axn_tgtR3def : $partition(targetR3, \{Msg32 \mapsto N34\}, \{Stb32 \mapsto N33\}, \{Nxt32 \mapsto N34\})$
// (Rule 3) function targetR3

axm_tR3_V : $tR3\_V \in vertR3 \rightarrow vertT$
// (Rule 3) Typing right vertices, tR3_V

axm_tR3_V_def : $partition(tR3\_V, \{N33 \mapsto Node\}, \{N34 \mapsto Node\})$
// (Rule 3) Typing right vertices, tR3_V

axm_tR3_E : $tR3\_E \in edgeR3 \rightarrow edgeT$
// (Rule 3) Typing right edges, tR3_E

axm_tR3_E_def : $partition(tR3\_E, \{Stb32 \mapsto Stb\}, \{Msg32 \mapsto Msg\}, \{Nxt32 \mapsto Nxt\})$
(Rule 3) Typing right edges, tR3_E

axm_alpha3V : $alpha3V \in vertL3 \twoheadrightarrow vertR3$
(Rule 3) Relational Rule alpha3: mapping vertices

axm_alpha3V_def : $partition(alpha3V, \{N31 \mapsto N33\}, \{N32 \mapsto N34\})$
// (Rule 3) Relational Rule alpha3: mapping vertices

axm_alpha3E : $alpha3E \in edgeL3 \twoheadrightarrow edgeR3$
// (Rule 3) Relational Rule alpha3: mapping edges

axm_alpha3E_def : $partition(alpha3E, \{Stb31 \mapsto Stb32\}, \{Nxt31 \mapsto Nxt32\})$

// (Rule 3) Relational Rule alpha3: mapping edges

axm_vertL4 : $partition(vertL4, \{N41\}, \{N42\})$
// (Rule 4) Left Graph L4 – Vertices

axm_edgeL4 : $partition(edgeL4, \{Tok41\}, \{Act41\}, \{Msg41\}, \{Nxt41\})$
// (Rule 4) Left Graph L4 – Edges names

axm_srcL4type : $sourceL4 \in edgeL4 \rightarrow vertL4$
// (Rule 4) function sourceL4

axn_srcL4def : $partition(sourceL4, \{Tok41 \mapsto N41\}, \{Msg41 \mapsto N41\}, \{Act41 \mapsto N41\}, \{Nxt41 \mapsto N41\})$
// (Rule 4) function sourceL4

axm_tgtL4type : $targetL4 \in edgeL4 \rightarrow vertL4$
// (Rule 4) function targetL4

axn_tgtL4def : $partition(targetL4, \{Tok41 \mapsto N41\}, \{Msg41 \mapsto N41\}, \{Act41 \mapsto N41\}, \{Nxt41 \mapsto N42\})$
// (Rule 4) function targetL4

axm_tL4_V : $tL4\_V \in vertL4 \rightarrow vertT$
// (Rule 4) Typing left vertices, tL4_V

axm_tL4_V_def : $partition(tL4\_V, \{N41 \mapsto Node\}, \{N42 \mapsto Node\})$
  // (Rule 4) Typing left vertices, tL4_V

axm_tL4_E : $tL4\_E \in edgeL4 \rightarrow edgeT$
  // (Rule 4) Typing left edges, tL4_E

axm_tL4_E_def : $partition(tL4\_E, \{Tok41 \mapsto Tok\}, \{Msg41 \mapsto Msg\}, \{Act41 \mapsto Act\}, \{Nxt41 \mapsto Nxt\})$
  // (Rule 4) Typing left edges, tL4_E

axm_vertR4 : $partition(vertR4, \{N43\}, \{N44\})$
  // (Rule 4) Right Graph R4 – Vertices

axm_edgeR4 : $partition(edgeR4, \{Tok42\}, \{Stb42\}, \{Nxt42\})$
  // (Rule 4) Right Graph R4 – Edges names

axm_srcR4type : $sourceR4 \in edgeR4 \rightarrow vertR4$
  // (Rule 4) function sourceR4

axn_srcR4def : $partition(sourceR4, \{Tok42 \mapsto N44\}, \{Stb42 \mapsto N43\}, \{Nxt42 \mapsto N43\})$
  // (Rule 4) function sourceR4

axm_tgtR4type : $targetR4 \in edgeR4 \rightarrow vertR4$
  // (Rule 4) function targetR4

axn_tgtR4def : $partition(targetR4, \{Tok42 \mapsto N44\}, \{Stb42 \mapsto N43\}, \{Nxt42 \mapsto N44\})$
  // (Rule 4) function targetR4

axm_tR4_V : $tR4\_V \in vertR4 \rightarrow vertT$
  // (Rule 4) Typing right vertices, tR4_V

axm_tR4_V_def : $partition(tR4\_V, \{N43 \mapsto Node\}, \{N44 \mapsto Node\})$
  // (Rule 4) Typing right vertices, tR4_V

axm_tR4_E : $tR4\_E \in edgeR4 \rightarrow edgeT$
  // (Rule 4) Typing right edges, tR4_E

axm_tR4_E_def : $partition(tR4\_E, \{Stb42 \mapsto Stb\}, \{Tok42 \mapsto Tok\}, \{Nxt42 \mapsto Nxt\})$
  // (Rule 4) Typing right edges, tR4_E

axm_alpha4V : $alpha4V \in vertL4 \twoheadrightarrow vertR4$
  // (Rule 4) Relational Rule alpha4: mapping vertices

axm_alpha4V_def : $partition(alpha4V, \{N41 \mapsto N43\}, \{N42 \mapsto N44\})$
  // (Rule 4) Relational Rule alpha4: mapping vertices

axm_alpha4E : $alpha4E \in edgeL4 \twoheadrightarrow edgeR4$
  // (Rule 4) Relational Rule alpha4: mapping edges

axm_alpha4E_def : $partition(alpha4E, \{Nxt41 \mapsto Nxt42\})$
  // (Rule 4) Relational Rule alpha4: mapping edges

axm_vertL5 : $partition(vertL5, \{N51\}, \{N52\})$
  // (Rule 5) Left Graph L5 – Vertices

axm_edgeL5 : $partition(edgeL5, \{Nxt51\})$
  // (Rule 5) Left Graph L5 – Edges names

axm_tL5_V : $tL5\_V \in vertL5 \rightarrow vertT$
  // (Rule 5) Typing left vertices, tL5_V

axm_srcL5type : $sourceL5 \in edgeL5 \rightarrow vertL5$
  // (Rule 5) function sourceL5

axn_srcL5def : $partition(sourceL5, \{Nxt51 \mapsto N51\})$
  // (Rule 5) function sourceL5

axm_tgtL5type : $targetL5 \in edgeL5 \rightarrow vertL5$
  // (Rule 5) function targetL5

axn_tgtL5def : $partition(targetL5, \{Nxt51 \mapsto N52\})$
  // (Rule 5) function targetL5

axm_tL5_V_def : $partition(tL5\_V, \{N51 \mapsto Node\}, \{N52 \mapsto Node\})$
  // (Rule 5) Typing left vertices, tL5_V

axm_tL5_E : $tL5\_E \in edgeL5 \rightarrow edgeT$
  // (Rule 5) Typing left edges, tL5_E

axm_tL5_E_def : $partition(tL5\_E, \{Nxt51 \mapsto Nxt\})$
  // (Rule 5) Typing left edges, tL5_E

axm_vertR5 : $partition(vertR5, \{N53\}, \{N54\}, \{N55\})$
  // (Rule 5) Right Graph R5 – Vertices

axm_edgeR5 : $partition(edgeR5, \{Stb51\}, \{Nxt52\}, \{Nxt53\})$
  // (Rule 5) Right Graph R5 – Edges names

axm_srcR5type : $sourceR5 \in edgeR5 \rightarrow vertR5$
  // (Rule 5) function sourceR5

axn_srcR5def : $partition(sourceR5, \{Stb51 \mapsto N55\}, \{Nxt52 \mapsto N53\}, \{Nxt53 \mapsto N55\})$
  // (Rule 5) function sourceR5

axm_tgtR5type : $targetR5 \in edgeR5 \rightarrow vertR5$
  // (Rule 5) function targetR5

axn_tgtR5def : $partition(targetR5, \{Stb51 \mapsto N55\}, \{Nxt52 \mapsto N55\}, \{Nxt53 \mapsto N54\})$
  // (Rule 5) function targetR5

axm_tR5_V : $tR5\_V \in vertR5 \rightarrow vertT$
  // (Rule 5) Typing right vertices, tR5_V

axm_tR5_V_def : $partition(tR5\_V, \{N53 \mapsto Node\}, \{N54 \mapsto Node\}, \{N55 \mapsto Node\})$
  // (Rule 5) Typing right vertices, tR5_V

axm_tR5_E : $tR5\_E \in edgeR5 \rightarrow edgeT$
  // (Rule 5) Typing right vertices, tR5_V

axm_tR5_E_def : $partition(tR5\_E, \{Stb51 \mapsto Stb\}, \{Nxt52 \mapsto Nxt\}, \{Nxt53 \mapsto Nxt\})$
  // (Rule 5) Typing right edges, tR5_E

axm_alpha5V : $alpha5V \in vertL5 \rightarrowtail vertR5$
  // (Rule 5) Relational Rule alpha5: mapping vertices

axm_alpha5V_def : $partition(alpha5V, \{N51 \mapsto N53\}, \{N52 \mapsto N54\})$
  // (Rule 5) Relational Rule alpha5: mapping vertices

axm_alpha5E : $alpha5E \in edgeL5 \rightarrowtail edgeR5$
  // (Rule 5) Relational Rule alpha5: mapping edges

axm_alpha5E_def : $alpha5E = \varnothing$
  // (Rule 5) Relational Rule alpha5: mapping edges

axmNxtTok : $Nxt \neq Tok$

axmNxtMsg : $Nxt \neq Msg$
axmNxtStb : $Nxt \neq Stb$
axmNxtAct : $Nxt \neq Act$
axmTokMsg : $Tok \neq Msg$
axmTokStb : $Tok \neq Stb$
axmTokAct : $Tok \neq Act$
axmMsgStb : $Msg \neq Stb$
axmMsgAct : $Msg \neq Act$
axmStbAct : $Stb \neq Act$

END

B.2 Event-B Machine of Token Ring

An Event-B Specification of mch_trAll
Creation Date: 8 Mar 2010 @ 09:59:48 PM

MACHINE mch_trAll
SEES ctx_trAll
VARIABLES

vertG // (Graph) Vertices
edgeG // (Graph) Edges
sourceG // (Graph) function sourceG
targetG // (Graph) function targetG
tG_V // (Graph) Typing vertices, tG_V
tG_E // (Graph) Typing edges, tG_E

INVARIANTS

inv_vertG : $vertG \in \mathbb{P}(\mathbb{N})$
// (Graph) Vertices are natural numbers.
inv_edgeG : $edgeG \in \mathbb{P}(\mathbb{N})$
// (Graph) Edges are natural numbers.
inv_srcGtype : $sourceG \in edgeG \rightarrow vertG$
// (Graph) function sourceG
inv_tgtGtype : $targetG \in edgeG \rightarrow vertG$
// (Graph) function targetG
inv_tG_V : $tG\_V \in vertG \rightarrow vertT$
// (Graph) function tG_V
inv_tG_E : $tG\_E \in edgeG \rightarrow edgeT$
// (Graph) function tG_E
prop1fin : $finite(dom(tG\_E \rhd \{Tok\}))$
// Property 0: The set of edges of type Tok of a reachable graph is finite.
prop1 : $card(dom(tG\_E \rhd \{Tok\})) = 1$
// Property 1: Any reachable graph has exactly one edge of type Tok.

EVENTS

Initialisation

begin

act_vertG : $vertG := \{1, 2, 3\}$
// (G = G0) Vertices

act_edgeG : $edgeG := \{1, 2, 3, 4, 5, 6, 7\}$
// (G = G0) Edges

act_srcG : $sourceG := \{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1, 4 \mapsto 2, 5 \mapsto 2, 6 \mapsto 3, 7 \mapsto 3\}$
// (G = G0) function sourceG

act_tgtG : $targetG := \{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 2, 4 \mapsto 2, 5 \mapsto 3, 6 \mapsto 3, 7 \mapsto 1\}$
// (G = G0) function targetG

act_tG_V : $tG\_V := \{1 \mapsto Node, 2 \mapsto Node, 3 \mapsto Node\}$
// (G = G0) Typing vertices

act_tG_E : $tG\_E := \{1 \mapsto Tok, 2 \mapsto Stb, 3 \mapsto Nxt, 4 \mapsto Stb, 5 \mapsto Nxt, 6 \mapsto Stb, 7 \mapsto Nxt\}$
// (G = G0) Typing edges

end

Event *rule1* $\widehat{=}$

any

mV // mV component of a match
mE // mE component of a match
$newEmsg$ // new fresh name for an edge
$newEact$ // new fresh name for an edge

where

grd_mV : $mV \in vertL1 \rightarrow vertG$
// mV is total

grd_mE : $mE \in edgeL1 \rightarrowtail edgeG$
// mE is total and injective

grd_newEmsg : $newEmsg \in \mathbb{N} \setminus edgeG$
// newEmsg is a fresh name

grd_newEact : $newEact \in \mathbb{N} \setminus edgeG$
// newEact is a fresh name

grd_E1E2 : $newEmsg \neq newEact$

grd_vertices : $\forall v \cdot v \in vertL1 \Rightarrow tL1\_V(v) = tG\_V(mV(v))$
vertex compatibility

grd_edges : $\forall e \cdot e \in edgeL1 \Rightarrow tL1\_E(e) = tG\_E(mE(e))$
edge compatibility

grd_srctgt : $\forall e \cdot e \in edgeL1 \Rightarrow mV(sourceL1(e)) = sourceG(mE(e)) \land mV(targetL1(e)) = targetG(mE(e))$
source/target compatibility

then

act_E : $edgeG := (edgeG \setminus \{mE(Stb11)\}) \cup \{newEmsg, newEact\}$

act_src : $sourceG := (\{mE(Stb11)\} \lhd sourceG) \cup \{newEact \mapsto mV(N11), newEmsg \mapsto mV(N12)\}$

act_tgt : $targetG := (\{mE(Stb11)\} \lhd targetG) \cup \{newEact \mapsto mV(N11), newEmsg \mapsto mV(N12)\}$

```
        act_tE : $tG\_E := (\{mE(Stb11)\} \lhd tG\_E) \cup \{newEact \mapsto Act, newEmsg \mapsto Msg\}$
    end
Event  $rule2 \mathrel{\widehat{=}}$
    any
        $mV$    // mV component of a match
        $mE$    // mE component of a match
        $newEtok$    // new fresh name for an edge
    where
        grd_mV : $mV \in vertL2 \to vertG$
            // mV is total
        grd_mE : $mE \in edgeL2 \rightarrowtail edgeG$
            // mE is total and injective
        grd_newE1 : $newEtok \in \mathbb{N} \setminus edgeG$
            // newEtok is a fresh name
        grd_vertices : $\forall v \cdot v \in vertL2 \Rightarrow tL2\_V(v) = tG\_V(mV(v))$
            vertex compatibility
        grd_edges : $\forall e \cdot e \in edgeL2 \Rightarrow tL2\_E(e) = tG\_E(mE(e))$
            edge compatibility
        grd_srctgt : $\forall e \cdot e \in edgeL2 \Rightarrow mV(sourceL2(e)) = sourceG(mE(e)) \land mV(targetL2(e)) = targetG(mE(e))$
            source/target compatibility
    then
        act_E : $edgeG := (edgeG \setminus \{mE(Tok21)\}) \cup \{newEtok\}$
        act_src : $sourceG := (\{mE(Tok21)\} \lhd sourceG) \cup \{newEtok \mapsto mV(N22)\}$
        act_tgt : $targetG := (\{mE(Tok21)\} \lhd targetG) \cup \{newEtok \mapsto mV(N22)\}$
        act_tE : $tG\_E := (\{mE(Tok21)\} \lhd tG\_E) \cup \{newEtok \mapsto Tok\}$
    end
Event  $rule3 \mathrel{\widehat{=}}$
    any
        $mV$    // mV component of a match
        $mE$    // mE component of a match
        $newEmsg$    // new fresh name for an edge
    where
        grd_mV : $mV \in vertL3 \to vertG$
            // mV is total
        grd_mE : $mE \in edgeL3 \rightarrowtail edgeG$
            // mE is total and injective
        grd_newE : $newEmsg \in \mathbb{N} \setminus edgeG$
            // newEmsg is a fresh name
        grd_vertices : $\forall v \cdot v \in vertL3 \Rightarrow tL3\_V(v) = tG\_V(mV(v))$
            vertex compatibility
        grd_edges : $\forall e \cdot e \in edgeL3 \Rightarrow tL3\_E(e) = tG\_E(mE(e))$
            edge compatibility
        grd_srctgt : $\forall e \cdot e \in edgeL3 \Rightarrow mV(sourceL3(e)) = sourceG(mE(e)) \land mV(targetL3(e)) = targetG(mE(e))$
            source/target compatibility
```

then

`act_E` : $edgeG := (edgeG \setminus \{mE(Msg31)\}) \cup \{newEmsg\}$
`act_src` : $sourceG := (\{mE(Msg31)\} \lhd sourceG) \cup \{newEmsg \mapsto mV(N32)\}$
`act_tgt` : $targetG := (\{mE(Msg31)\} \lhd targetG) \cup \{newEmsg \mapsto mV(N32)\}$
`act_tE` : $tG\_E := (\{mE(Msg31)\} \lhd tG\_E) \cup \{newEmsg \mapsto Msg\}$

end

Event *rule4* $\widehat{=}$

any

mV // mV component of a match
mE // mE component of a match
$newEstb$ // new fresh name for an edge
$newEtok$ // new fresh name for an edge

where

`grd_mV` : $mV \in vertL4 \rightarrow vertG$
// mV is total
`grd_mE` : $mE \in edgeL4 \rightarrowtail edgeG$
// mE is total and injective
`grd_newEstb` : $newEstb \in \mathbb{N} \setminus edgeG$
// newEstb is a fresh name
`grd_newEtok` : $newEtok \in \mathbb{N} \setminus edgeG$
// newEtok is a fresh name
`grd_newE1E2` : $newEstb \neq newEtok$
`grd_vertices` : $\forall v \cdot v \in vertL4 \Rightarrow tL4\_V(v) = tG\_V(mV(v))$
vertex compatibility
`grd_edges` : $\forall e \cdot e \in edgeL4 \Rightarrow tL4\_E(e) = tG\_E(mE(e))$
edge compatibility
`grd_srctgt` : $\forall e \cdot e \in edgeL4 \Rightarrow mV(sourceL4(e)) = sourceG(mE(e)) \land$
$mV(targetL4(e)) = targetG(mE(e))$
source/target compatibility

then

`act_E` : $edgeG := (edgeG \setminus \{mE(Tok41), mE(Act41), mE(Msg41)\}) \cup$
$\{newEstb, newEtok\}$
`act_src` : $sourceG := (\{mE(Tok41), mE(Act41), mE(Msg41)\} \lhd sourceG) \cup$
$\{newEstb \mapsto mV(N41), newEtok \mapsto mV(N42)\}$
`act_tgt` : $targetG := (\{mE(Tok41), mE(Act41), mE(Msg41)\} \lhd targetG) \cup$
$\{newEstb \mapsto mV(N41), newEtok \mapsto mV(N42)\}$
`act_tE` : $tG\_E := (\{mE(Tok41), mE(Act41), mE(Msg41)\} \lhd tG\_E) \cup$
$\{newEstb \mapsto Stb, newEtok \mapsto Tok\}$

end

Event *rule5* $\widehat{=}$

any

mV // mV component of a match
mE // mE component of a match
$newV$ // new fresh name for a vertex
$newE1$ // new fresh name for an edge
$newE2$ // new fresh name for an edge
$newEstb$ // new fresh name for an edge

where

grd_mV : $mV \in vertL5 \rightarrow vertG$
// mV is total
grd_mE : $mE \in edgeL5 \rightarrowtail edgeG$
// mE is total and injective
grd_newV : $newV \in \mathbb{N} \setminus vertG$
// newV is a fresh name
grd_newEstb : $newEstb \in \mathbb{N} \setminus edgeG$
// newEstb is a fresh name
grd_newE1 : $newE1 \in \mathbb{N} \setminus edgeG$
// newE1 is a fresh name
grd_newE2 : $newE2 \in \mathbb{N} \setminus edgeG$
// newE2 is a fresh name
grd_newE1E2 : $newE1 \neq newE2$
grd_newE2stb : $newE2 \neq newEstb$
grd_newE1stb : $newE1 \neq newEstb$
grd_vertices : $\forall v \cdot v \in vertL5 \Rightarrow tL5\_V(v) = tG\_V(mV(v))$
vertex compatibility
grd_edges : $\forall e \cdot e \in edgeL5 \Rightarrow tL5\_E(e) = tG\_E(mE(e))$
edge compatibility
grd_srctgt : $\forall e \cdot e \in edgeL5 \Rightarrow mV(sourceL5(e)) = sourceG(mE(e)) \wedge mV(targetL5(e)) = targetG(mE(e))$
source/target compatibility

then

act_vertG : $vertG := vertG \cup \{newV\}$
act_tG_V : $tG\_V := tG\_V \cup \{newV \mapsto Node\}$
act_E : $edgeG := (edgeG \setminus \{mE(Nxt51)\}) \cup \{newE1, newE2, newEstb\}$
act_src : $sourceG := (\{mE(Nxt51)\} \lhd sourceG) \cup \{newE1 \mapsto mV(N51), newE2 \mapsto newV, newEstb \mapsto newV\}$
act_tgt : $targetG := (\{mE(Nxt51)\} \lhd targetG) \cup \{newE1 \mapsto newV, newE2 \mapsto mV(N52), newEstb \mapsto newV\}$
act_tE : $tG\_E := (\{mE(Nxt51)\} \lhd tG\_E) \cup \{newE1 \mapsto Nxt, newE2 \mapsto Nxt, newEstb \mapsto Stb\}$

end

END

APPENDIX C RESUMO ESTENDIDO DA TESE

Nesta tese, introduziu-se uma abordagem lógica e relacional de gramática de grafos para permitir a análise de sistemas distribuídos e assíncronos com espaço de estados infinito. Utilizou-se estruturas relacionais para caracterizar gramática de grafos e definiu-se aplicações de regras como transduções definíveis. Primeiro considerou-se gramática de grafos definidas sobre grafos (tipados) simples, e então se estendeu a representação para grafos com atributos e para gramáticas com condições negativas de aplicação. Mostrou-se que a abordagem proposta oferece uma codificação adequada para a definição *single pushout* (SPO) de gramática de grafos, podendo ser utilizada como base para o uso de técnicas de prova de teoremas para prova de propriedades, complementando as abordagens existentes baseadas em técnicas de verificação automática de modelos. A maior contribuição deste trabalho não deve ser vista como uma nova abordagem para descrever gramática de grafos, mas como uma forma de permitir o uso de técnicas (e ferramentas) de prova de teoremas para as abordagens existentes (modelou-se aqui a abordagem SPO, mas a teoria proposta pode também ser utilizada como base para manipular outras abordagens). Este é um resultado relevante desde que gramática de grafos oferece uma técnica de especificação interessante para diversas áreas de aplicações e, até o momento, técnicas de prova de teoremas não podiam ser utilizadas para analisar propriedades de gramáticas de grafos. As principais contribuições deste trabalho são:

- A *representação lógica e relacional de gramática de grafos* (Capítulo 3) estabelece as fundamentações teóricas para a análise de gramáticas de grafos através de prova de teoremas. Representou-se gramática de grafos e seu comportamento utilizando estruturas lógicas e relacionais porque elas constituem a base de provadores de teoremas. Trabalhos relacionados (STRECKER, 2008; BARESI; SPOLETINI, 2006) que adotam uma descrição de gramática de grafos baseada em representações lógicas ou em teoria dos conjuntos, ou não estão verificando propriedades de gramáticas de grafos ou estão limitadas para analisar sistemas dentro de um escopo finito, cujo tamanho é definido pelo usuário. Abordagens para analisar gramática de grafos com número infinito de estados (BALDAN; CORRADINI; KÖNIG, 2008; BALDAN; KÖNIG; RENSINK, 2005) derivam o modelo como aproximações, as quais podem resultar em relatórios inconclusivos de verificação.

  A definição de gramática de grafos como estruturas relacionais (Def. 13) permite a associação de uma gramática de grafos com uma tupla composta de um conjunto e uma coleção de relações sobre este conjunto. O conjunto descreve o domínio da estrutura (o conjunto de vértices e arcos da gramática de grafos) e as relações definem o grafo tipo, o grafo inicial e as regras. Uma série de condições lógicas impõe restrições aos elementos destas relações para garantir que elas realmente representem os componentes de uma gramática de grafos (grafos, grafos tipados, morfismos de grafos e regras). A aplicação de uma regra é descrita por uma transdução definível (Def. 16), que pode ser vista como uma regra de inferência na estrutura relacional associada a gramática de grafos. O resultado da transdução é outra gramática de grafos cujo estado inicial corresponde ao resultado da aplicação de uma regra a um dado *match* ao estado inicial da gramática original. Os outros componentes da gramática permanece inalterados (isto é, a gramática resultante tem o mesmo grafo tipo e regras da gramática original). Proposições 7 e 9 garantem que a codificação adotada está bem-definida. Para uso em verificação, as relações da estrutura relacional definem axiomas que podem ser utilizados nas provas e propriedades sobre estados alcançáveis são provadas por indução: primeiro (caso base) a propriedade é verificada para o grafo inicial e então, no passo indutivo, a propriedade é verificada para cada regra da gramática aplicável a um grafo alcançável G, considerando que a propriedade é válida para G.

- A *abordagem relacional para gramáticas de grafos com atributos* (Capítulo 4) é uma extensão do formalismo básico que integra o uso de tipos de dados em grafos. Gramática de grafos com atributos é bastante interessante do ponto de vista prático, desde que é possível utilizar variáveis e termos quando se especifica o comportamento expresso por regras. Estes valores (ou termos) vêm de álgebras especificadas como tipos abstratos de dados. O uso de grafos com atributos fornece ao especificador uma linguagem que é mais adequada para especificação, combinando as vantagens da representação gráfica com uma representação padrão para tipos de dados clássicos. Partindo de uma perspectiva prática, grafos com atributos são necessários desde que não é viável codificar tipos de dados como números naturais ou strings, etc. em grafos. Para verificação formal, a presença de atributos insere problemas adicionais, desde que tipos de dados são frequentemente conjuntos infinitos. Na verdade, mesmo restringindo apenas para conjuntos finitos, especificações que usam grafos com atributos frequentemente levam a sistemas não verificáveis devido a explosão de estados. Existem algumas abordagens para verificar gramáticas de grafos com atributos, como (KASTENBERG, 2006) e (KÖNIG; KOZIOURA, 2008) e elas funcionam para classes limitadas de gramáticas. Mostrou-se que atributos podem ser integrados de forma adequada na representação proposta de gramática de grafos. A abordagem proposta provê uma base para uma ferramenta para argumentar sobre uma classe maior de gramática de grafos, incluindo gramáticas que especificam sistemas com espaço de estados infinito, sem utilizar nenhum tipo de aproximação.

  As Definições 29 e 31 expressam a representação relacional de uma gramática de grafos com atributos. As Proposições 16 e 18 garantem que a extensão relacional está bem-definida. A estratégia de prova aplicada na etapa de verificação é a mesma descrita anteriormente: utilizou-se indução matemática, considerando que as relações da estrutura relacional definem axiomas a serem utilizados nas provas. A diferença é que agora utilizou-se variáveis como atributos no lado direito e esquerdo das regras, e então, em diversas situações, no passo indutivo o desenvolvimento de provas envolve variáveis. Neste caso, para estabelecer a propriedade, devem-se considerar as equações da regra que está sendo aplicada como axiomas.

- A *extensão para gramática de grafos com condições negativas de aplicação* (Capítulo 5) permite a especificação de que uma certa estrutura é proibida ao se executar uma aplicação de regra, aumentando a expressividade da transformação. Particularmente, condições negativas de aplicação (NACs) restringem a aplicação de uma regra expressando que uma estrutura específica (por exemplo nodos, arcos ou subgrafos) não devem estar presentes num grafo-estado antes de se aplicar uma regra. Condições de aplicação são comumente utilizadas em especificações não triviais. Como enfatizado em (HABEL; HECKEL; TAENTZER, 1996) elas são expressas frequentemente de maneira informal assumindo algum tipo de mecanismo de controle que não é especificado. No entanto, tal estratégia impede especificação e verificação formal. A expressão de NACs é atualmente possível em ferramentas (ERMEL; RUDOLF; TAENTZER, 1999; SCHüRR; WINTER; ZüNDORF, 1999) de gramática de grafos que focam em análise de conflitos e comportamento funcional. NACs também podem ser especificadas em GROOVE (KASTENBERG; RENSINK, 2006b) para a análise de gramáticas de grafos com estados infinitos, no caso em que o espaço de estados possa ser representado dentro de um fragmento finito.

  A Definição 36 associa uma estrutura relacional a uma gramática de grafos com condições negativas de aplicação. A Proposição 20 mostra que a definição relacional está bem-definida. Nesta abordagem, condições extras devem ser checadas antes de uma aplicação de regra para garantir que os elementos proibidos não estão no grafo-estado. Na etapa de verificação, a existência de NACs determina condições extras que podem ser utilizadas durante as provas.

- Os *padrões de propriedades* (Capítulo 6) propostos contêm 15 classes de padrões, dentro das quais requisitos funcionais e estruturais de estados alcançáveis podem ser formulados. Os padrões tem o objetivo de auxiliar e simplificar a tarefa de descrever requisitos precisos a serem verificados. Eles devem prover o auxílio suficiente para a especificação de propriedades sobre estados alcançáveis de gramáticas de grafos. Acredita-se que os padrões propostos representam o primeiro passo na direção de um padrão de especificação para propriedades sobre estados no contexto de gramáticas de grafos. Diferentemente da maioria das abordagens propostas (DWYER; AVRUNIN; CORBETT, 1999; CHECHIK; PAUN, 1999; SALAMAH et al., 2007), o foco foi dado em propriedades sobre estados alcançáveis para verificação de estados (infinitos). A maioria dos padrões existentes para especificação de propriedades descrevem propriedades sobre traços para ferramentas de verificação de estados finitos. Estas duas abordagens são complementares.

  As Tabelas 6.1, 6.2 e 6.3 descrevem uma biblioteca padrão de funções que descrevem características típicas ou elementos de grafos (como vértices de determinado tipo, o conjunto de todos os arcos de algum tipo, a cardinalidade de vértices, etc.). Estas funções foram definidas dentro do escopo de gramáticas de grafos relacionais. A Tabela 6.4 propõe uma taxonomia de padrões enquanto as Tabelas 6.6 e 6.7 listam uma coleção de padrões para especificação de propriedades.

- A *modelagem de especificações de gramática de grafos em estruturas de event-B* (Capítulo 7) permitiu o uso de provadores de event-B (através da plataforma Rodin) para demonstrar propriedades de uma gramática de grafos. Event-B (DEPLOY, 2010) tem sido utilizado com sucesso em diversas outras aplicações e possui ferramentas de suporte disponíveis tanto para especificação quanto para análise. Event-B foi escolhida devido a similaridade entre modelos event-B e especificações em gramáticas de grafos, especialmente o comportamento baseado em regras. Diversos outros trabalhos (ZEYDA; CAVALCANTI, 2009; ISOBE; ROGGENBACH, 2008a; LEHMANN; LEUSCHEL, 2003) têm focado na prova de teoremas de sistemas concorrentes, mas para sistemas assíncronos, gramática de grafos tem vantagem devido ao seu estilo visual e modular.

Para definir um modelo event-B, utilizou-se a definição relacional de gramática de grafos. O grafo tipo é definido em um contexto de um modelo event-B, onde tipos de vértices, arcos e relações de incidência relacionando eles são definidos como constantes. Um conjunto de axiomas define estes conjuntos explicitamente. Um grafo tipado sobre um grafo tipo é modelado por um conjunto de variáveis descrevendo seu conjunto de vértices, relação de incidência e funções de tipagem. As condições de compatibilidade de tipos e origem e destino de arcos podem ser declaradas como invariantes. O evento de inicialização é utilizado para criar o grafo inicial. A estrutura de uma regra é definida por conjuntos, constantes e invariantes. O comportamento de uma regra é descrito por um evento com condições de guarda. Um conjunto de ações atualiza o grafo estado de acordo com a regra.

Finalmente, é possível dizer que o campo de pesquisa sobre prova de teoremas para gramática de grafos está nos seus primeiros estágios. Existem diversas questões em aberto que devem ser objeto de trabalhos futuros.

- Além de implementação, estudos de casos são necessários para avaliar e melhorar a abordagem proposta. Até o momento, as extensões do formalismo básico de gramática de grafos não foram especificados na plataforma Rodin. É possível também investigar até que ponto a teoria do refinamento, que é bem desenvolvida em event-B, pode ser utilizada para validar um desenvolvimento passo-a-passo baseado em gramática de grafos. Outro objetivo é a implementação do tipo de dado grafo alcançável a ser usado na especificação e verificação de modelos de gramática de grafos. Esta estratégia deve ser comparada e avaliada com a implementação adotada.

- Outras classes de gramáticas de grafos não consideradas nesta tese englobam diversas aplicações práticas. Em particular, gramática de grafos baseada em objetos (DOTTI et al., 2003), gramática de grafos baseada em objetos temporizadas (MICHELON; COSTA; RIBEIRO, 2007, 2006), gramática de grafos orientada a objetos (FERREIRA; FOSS; RIBEIRO, 2007) e muitas outras (SCHFüRR, 1997) possivelmente com outros tipos de estrutura de grafos, como hiper-grafos, hipergrafos atribuídos e rotulados, têm seu próprio campo de aplicação. Desta forma, seria interessante investigar uma descrição geral da abordagem relacional de forma que diversos tipos de grafos e/ou gramáticas se tornem instâncias desta ferramenta mais geral.

- A abordagem aqui proposta deve ser definida para gramáticas na abordagem *double-pushout* (DPO) sem maiores problemas. Na abordagem SPO é apenas necessário encontrar uma imagem do lado esquerdo da regra num grafo alcançável para que a regra possa ser aplicada. Na abordagem DPO algumas restrições extras devem ser verificadas, denominada *gluing condition*, antes que uma regra seja aplicada.

Isto significa que algumas fórmulas lógicas adicionas (ou condições de guarda adicionais no caso de estruturas event-B) devem ser incluídas para ser checadas antes de uma aplicação de regra.

- Os padrões de propriedades também devem ser incorporados na ferramenta de prova. Seria de grande auxílio detalhar para cada requisito, tanto quanto possível, as propriedades ou lemas que devem ser exigidos para a conclusão da prova, incluindo estratégias de provas que podem ser adotadas em cada caso. Simultaneamente, uma gramática estruturada pode ser desenvolvida para auxiliar na formulação de propriedades. Além disso, uma extensão natural dos padrões declarados seria a investigação dos requisitos descritos com lógica de alta-ordem. Deve-se, por fim, complementar e avaliar o sistema de padrões proposto analisando um número apropriado de especificações do mundo real.

- Outro tópico de trabalho futuro é o uso da técnica de prova de teoremas para analisar outros tipos de propriedades, como propriedades de segurança e *liveness*. Em particular, controlabilidade, ou a propriedade de se alcançar um particular (conjunto de) estado(s) do sistema qualquer que seja o atual, é um importante tópico de análise. Tal propriedade não pode ser verificada por indução matemática desde que não é finitária. Ela deve ser definida sobre todos os comportamentos futuros do sistema.