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

ANA PAULA LÜDTKE FERREIRA

Object-Oriented Graph Grammars

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

Prof. Dr. Leila Ribeiro
Advisor

Porto Alegre, September 2005

CIP – CATALOGING-IN-PUBLICATION

Lüdtke Ferreira, Ana Paula

Object-Oriented Graph Grammars / Ana Paula Lüdtke Ferreira. – Porto Alegre: PPGC da UFRGS, 2005.

158 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2005. Advisor: Leila Ribeiro.

1. Graph grammars. 2. Object orientation. 3. Formal verification. I. Ribeiro, Leila. II. Título.

Mr. Worf, fire!

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF DEFINITIONS

LIST OF THEOREMS

NOTATION

$f : A \to B$	f is a (partial) function with domain A and codomain B
$f : A \mapsto B$	f is a total function
$f : A \hookrightarrow B$	f is an injection
$dom(f)$	domain of function f
$cod(f)$	codomain of function f
$im(f)$	image of function f
$f(a)$	value of function f at the point $a \in dom(f)$
$f(A)$	$\{f(a) \mid a \in A\}$
$undef$	undefined value (usually representing $f(a)$ if $a \notin dom(f)$)
$A \times B$	cartesian product of sets A and B
$A \uplus B$	disjoint union of sets A and B
$\mathscr{P}(X)$	powerset of X, i.e., $\{Y \mid Y \subseteq X\}$
$\langle P, \sqsubseteq_P \rangle$	partially ordered set (Definition B.9)
$\uparrow p$	upper set of an element p (Definition B.11)
$\downarrow p$	lower set of an element p (Definition B.11)
$\sqcup A$	supremum of A (Definition B.14)
$\sqcap A$	infimum of A (Definition B.14)
$ub(A)$	set of upper bounds of A (Definition B.12)
$lb(A)$	set of lower bounds of A (Definition B.12)
$lub(A)$	least element of $ub(A)$ (Definition B.14)
$glb(A)$	greatest element of $lb(A)$ (Definition B.14)
Set	Definition C.9
SetP	Definition C.13
Graph	Definition D.14
GraphP	Definition D.15
HGraph	Definition D.16
HGraphP	Definition D.17
LabHGraph	Definition D.18
LabHGraphP	Definition D.19
HGraphP(**T**)	Definition D.20

ABSTRACT

This thesis presents a graph-based formal framework to model and verify object-oriented specifications. More specifically, an extension of the algebraic single-pushout approach to (typed) graph grammars is developed, where the typing morphisms are compatible with the order relations defined over nodes and edges to represent, respectively, inheritance and overriding of classes and methods. This work is divided in three main lines: static specifications, dynamic behaviour, and formal verification of object-oriented systems.

The object-oriented class hierarchy structure is modeled by a graph structure called class-model graph, whose set of nodes and edges have a restricted partial order relation over them, to model inheritance and method overriding. The underlying relations of such sets obey additional restrictions, intended to assure that class-model graphs provide an adequate and faithful model of how object-oriented classes are organized and combined.

Object-oriented graph grammars model the dynamics of object-oriented systems. Object-oriented graphs are hypergraphs typed over a class-model graph, but the typing morphism is more flexible than the traditional one, in the sense that mapped hyperedges need to preserve *relations* between sources and targets. This feature adequately models inheritance, for any object can make use of inherited attributes or messages. Morphisms between object-oriented graphs assure that subclass polymorphism is a built-in feature of the formalism. Object-oriented rules respect the principles of encapsulation and information hiding of the object-oriented paradigm. A direct derivation (or rule application) is shown to be a pushout in the category of object-oriented graphs and their morphisms. An observational semantics for object-oriented graph grammars, based on a labeled transition system, is presented. This semantics is based on a notion of visible entities (objects or messages), which are the elements we are interested in for verification purposes.

Finally, a formal translation from object-oriented graph grammars specifications into Promela programs is defined. Objects in the system graph are translated as Promela processes, and message exchange is implemented with buffered communication channels. The semantics of grammar rule application is preserved by the nondeterminism in the choice of which message to consume. Inheritance, polymorphism and dynamic binding are implemented in the Promela program, which originally does not support it. The translation presented assures that both state and event verification can be performed.

Keywords: Graph grammars, Object orientation, Formal verification.

Gramáticas de Grafos Orientados a Objeto

RESUMO

Esta tese apresenta um modelo conceitual para modelagem e verificação de especificações de sistemas orientados a objeto. Mais especificiamente, uma extensão da abordagem algébrica baseada em *single-pushouts* para gramáticas de grafos tipadas é desenvolvida, onde os morfismos de tipagem são compatíveis com as relações de ordem sobre os nodos e (hiper)arcos de um grafo, e que representam, respectivamente, as relações de herança entre classes e sobrescrita de métodos. O trabalho é dividido em três linhas principais: especificações de sistemas, comportamento dinâmico de programas, e verificação formal de sistemas orientados a objeto.

A hierarquia de classes de um sistema orientado a objeto é modelada por um hipergrafo rotulado chamado *grafo de classes*, cujos conjuntos de nodos e arcos possuem uma relação de ordem parcial restrita, com o objetivo de modelar herança e sobrescrita de métodos. Restrições adicionais garantem que grafos de classes provêm um modelo fiel e adequado da maneira como as classes de um sistema orientado a objetos são efetivamente organizadas e combinadas.

Grafos orientados a objeto são hipergrafos tipados sobre um grafo de classes. O morfismo de tipagem exige que hiperarcos mapeados preservem as *relações* existentes entre os seus nodos de origem e destino. Esta característica modela a herança de forma adequada, visto que qualquer objeto pode fazer uso de atributos ou mensagens herdadas. Morfismos entre grafos orientados a objeto asseguram que o polimorfismo de subclasses seja uma característica intrínseca do formalismo aqui apresentado. Regras orientadas a objeto respeitam os princípios de encapsulamento e oclusão da informação do paradigma. Uma derivação direta (ou aplicação de regra) é uma soma amalgamada (*pushout*) na categoria de grafos orientados a objeto e seus morfismos. Gramáticas de grafos orientados a objeto modelam o comportamento dinâmico de sistemas. Uma semântica observacional para gramáticas de grafos orientados a objeto, baseada em sistemas de transição rotulados, é definida. Tal semântica é baseada na noção de entidades visíveis (objetos ou mensagens), e que representam os elementos importantes no processo de verificação de propriedades do sistema especificado pela gramática.

Finalmente, uma tradução formal de gramáticas de grafos orientados a objeto para programas na linguagem Promela é definida. Objetos são traduzidos como processos em Promela, e a troca de mensagens entre objetos é implementada com canais de comunicação. Herança, polimorfismo e ligação dinâmica são implementados no programa Promela, que originalmente não suporta nenhuma dessas caraterísticas. A verificação de propriedades do programa pode ser efetuada tanto sobre estados como sobre eventos.

Palavras-chave: Gramáticas de grafos, Orientação a objeto, Verificação formal.

1 INTRODUCTION

1.1 Motivation

Computer programs have become a major part of our lives. We rely on them to help us execute most of our activities during the day. As examples of applications we use on a daily basis one can cite cash withdrawal terminals, Internet banking, flight reservation systems, supermarket cashiers, Internet browsers, electronic mail clients, text processors, among many others. More sophisticated services are also computer programmed, such as telecommunication systems, railway and street semaphore control, airplanes autopilots, and even life support systems provided by intensive care units in hospitals.

One of the main factors shared by the mentioned applications is that computer programs are obliged to coordinate a high number of system components that operate simultaneously. For example, a terminal for travel reservations needs to access a shared database to verify if a certain seat is available or not; autopilots should receive information from several parts of the aircraft as well as information concerning weather conditions to guarantee the route to be followed correctly; life support equipments should monitor constantly the vital signals of a patient to assure that, in case of any problem, adequate measures are taken on time.

For the clients of such services, it is of fundamental importance that their operation occurs in an absolutely correct form: bank accounts should reflect exactly what was deposited and withdrawn, planes should keep themselves in the route and artificial breathers should supply oxygen at the correct rate for the patient. If a failure during the execution of an electronic game or a text editor does not generate significant problems, failures in the execution of programs of critical systems can have very serious or even fatal consequences. However, regardless of those requisites, cases where some hardware or software failure caused significant personal or financial damage are often reported. A concrete and relatively recent example was the explosion of the French rocket *Ariane 5*, in June of 1996, less than forty seconds after its launch. The investigation on the accident concluded that there was an error in the program that should calculate the rocket trajectory. The very same error also occurred in the backup computer, which caused the rocket to be destroyed (HUTH; RYAN, 2000).

The massive decrease on hardware costs has also contributed to the dissemination of computers and computational devices within society. The profile of computer users is becoming more and more diverse. Consequently, different domains of application are arising constantly. New software development techniques have emerged over the last years to deal with current developing demands, but the paradigms on which those techniques are based (especially objects, events and concurrency), although making the development process itself easier, make testing and validation of such systems more complex (and consequently, more error prone). This scenario requires specification techniques which can cope with the needs of modern software development. Such techniques must assure that the final product is consistent and complete regarding its specification, be formal, incremental and preferably executable. Also, they must be simple enough to be used by non experts in formal methods.

One of the biggest challenges in the acceptance of formal methods in industry lies in the difficulty of integrate formal specification techniques into the software development process (ARAÚJO JR.; SAWYER, 1998). In spite of the growing evidence that formal methods can offer benefits in terms of quality, strategies to their use appear to be neglected in most organizations. There are a number of reasons why formal methods are not used in industry, and the most commonly cited are the need of a major change on the requisite and resources needed for the process of specification to take place; the perception that those methods are hard to understand if you are not already a specialist; and that the appropriate abstractions are sometimes difficult to identify. This is true for some known formal software specification languages, such as VDM (BJORNES; JONES, 1978), Z (MONIN, 2003), LOTOS, λ-calculi, OBJ (GOGUEN; MALCOM, 1996), Clear (BURSTALL; GOGUEN, 1981), etc., which can only be fully understood and used by people with a reasonable mathematical knowledge. Ideally, people who develop software systems should have this background, but one cannot expect that every client is fluent in the language of mathematics.

Clients — in the sense of people who are the future users of the software system being developed — have to participate in the development process, or at least in the specification process, otherwise there will be no guarantees that the system requirements will be met. Not surprisingly, most informal and semi-formal methods for software specifications are based in some kind of diagram. Diagrams are usually easier to understand than a specification written in some mathematical language. Flowcharts, dataflow diagrams, entity-relationship diagrams, and the Unifying Modeling Language (UML) are familiar examples of such diagrams. Although visually simpler, diagrams are generally built from specific syntactic rules. Therefore, diagrams constitute a visual language of specification, with a formal syntax. If such language is also equipped with a formal semantics, we have a visual formal specification language which can (hopefully) be integrated more easily into the software development process. Diagrams used in the specification process can usually be modeled as *graphs*.

Graphs are mathematical (algebraic) structures which can convey a significant amount of information in a compact, visual, and understandable way. Notably, relations of any order can be easily expressed as graphs. The specification of computational systems using graphs offers two advantages which are, as seen before, in general mutually exclusive: (i) being formal mathematical structures, they have a well defined semantics and, (ii) having a diagrammatical layout, graph specifications can be more easily produced and understood by people outside the field of Computer Science. Therefore, graph-based specification techniques can provide a solid basis for the integration of formal specification techniques into the software development process and to strength the use of formal methods in industry (BARESI; PEZZÈ, 2000).

The rest of this chapter presents the main subjects which are addressed within this work, and upon which the theoretical framework for this thesis is founded. Section 1.2 presents graph transformation systems and graph grammars, which are an extension of transformation systems and Chomsky grammars applied to graphs. Section 1.3 describes object-oriented systems, and how they differ from the more traditional models of computation. Section 1.4 talks about model checking, which is a fully automatic method for verification of finite-state systems. Finally, Section 1.5 lists the main contributions and outline of this thesis.

1.2 Graph transformation systems

Rule–based systems have proven to be useful for describing computations by local transformations: arithmetic, syntactic, and deduction rules are familiar examples. The greatest advantage of rule–based systems lies on the fact that the knowledge of algorithms that most people have, which comes mainly from the study of mathematics in elementary and high school, is based on rules. We know rules to transform mathematical expressions into equivalent ones, to calculate the value of an element belonging to an equation, to extract the square root of a positive number, to find the minimum of a function, and so on and so forth. Areas like language definition (HOPCROFT; MOTWANI; ULLMAN, 2001), logic and functional programming (CLARK; TAERNLUND, 1982), (STERLING; SHAPIRO, 1994), (READE, 1995), algebraic specification (BERGSTRA; HEERING; KLINT, 1989), term rewriting (GADDUCCI, 1996), (KENNAWAY, 1995), theorem proving (FITTING, 1996), concurrent processes (MILNER, 1989a), and expert systems (NIKOLOPOULOS, 1997), (RUSSELL; NORVING, 1995), (KOSKO, 1992) witness the prominent role of rules in computer programming.

Rule-based graph transformation is, for those reasons, a quite natural way to combine graphs, for describing complex structures, with rules, to manipulate those structures. *Graph transformation*, also known as *graph rewriting*, combines the potentials and advantages of both, graphs and rules, into a single computational paradigm (EHRIG et al., 1996).

The theory of graph transformation systems studies a variety of formalisms which expand the theory of formal languages (HOPCROFT, 1969), (HOPCROFT; MOTWANI; ULLMAN, 2001), (LEWIS; PAPADIMITRIOU, 1998), (MARTIN, 1996), to encompass more general structures specified as graphs. All constructions known in the string transformation approach from the formal language framework also exist in graph transformation: rules, derivations and generated languages can be defined for graphs in the very same fashion they are defined for strings. Only naturally, different types of graph rules give rise to different classes of graph languages, with different expressiveness and differences in the decidability of the recognition problem. Similarly to string grammars, graph transformation systems also provide a model of computation. It has been shown that all enumerable graph sets can be generated using very restricted graph grammar models (NAGL, 1986). A graph transformation system allows to describe finitely a collection (finite or infinite) of graphs, which can be obtained from an initial graph through the repeated application of graph productions. The structure of graphs, graph productions, and results of rule applications determine the model of computation provided.

Graph transformation has been studied in a variety of approaches, motivated by application domains such as pattern recognition, semantics of programming languages, compiler description, implementation of functional programming languages, specification of database systems, specification of abstract data types, specification of distributed systems, and many others. This development is documented mainly in conference proceedings (CLAUS; EHRIG; ROZENBERG, 1979), (EHRIG; NAGL; ROZENBERG, 1982), (EHRIG et al., 1986), (EHRIG; KREOWSKI; ROZENBERG, 1990), (SCHNEIDER; EHRIG, 1993), (EHRIG; ENGELS; ROZENBERG, 1994), (EHRIG et al., 1998), (CORRADINI et al., 2002), (EHRIG et al., 2004), and other collections of selected papers, as well as in a series of handbooks (EHRIG et al., 1996), (EHRIG et al., 1997), and (EHRIG et al., 1999).

Graph transformation is generally non-deterministic because of two reasons: (i) there may be more than one rule which can be applied to a certain graph, and (ii) applying a graph transformation rule to a graph means to transform it locally so that there may be various parts in the graph which can be manipulated by rule applications. To regulate the graph transformation process, for example by choosing rules according to a priority, or by prescribing a certain sequence of steps, suitable control conditions can be stated.

A *graph grammar* is a graph transformation system equipped with an *initial graph* (of a suitable kind). The underlying graph data model and the type of transformation rules of a graph grammar form a so-called *graph transformation approach* (ANDRIES et al., 1999). In the literature, there exist a variety of graph transformation approaches like the expression approach (NAGL, 1986), *Progress* (SCHÜRR, 1990), algebraic ones like the double-pushout approach (EHRIG; PFENDER; SCHNEIDER, 1973) or the single-pushout approach (LÖWE, 1991), and more restricted approaches like node replacement (ENGELFRIET; ROZENBERG, 1997), edge replacement (HABEL; KREOWSKI, 1982), hyperedge replacement (HABEL, 1992), probabilistic replacement (MOSBAH, 1996) or mixed approaches such as (BAUDERON; JACQUET, 2001) and (LORETO; RIBEIRO; TOSCANI, 2002). For example, in *Progress*, attributed graphs are transformed, whereas hyperedge replacement applies rules whose left-hand side is composed of a single hyperedge with its corresponding vertices.

The *algebraic approach to graph grammars*, presented for the first time in (EHRIG; PFENDER; SCHNEIDER, 1973) makes use of categorical constructs to define the relevant aspects of the model of computation provided by graph grammars. That approach is currently known as *double-pushout* approach, because derivations are based on two pushout constructions in the category[1] of graphs and total graph morphisms. The *single-pushout* approach (LÖWE, 1991), on the other hand, has derivations characterized as a pushout construction in the category of graphs and partial graph morphisms. It is a proper extension of the double-pushout approach (EHRIG et al., 1996) capable of dealing with addition and deletion of items in unknown contexts, which is an important feature for distributed systems.

The algebraic approach to graph grammars have been used to specify various kinds of software systems, where graphs correspond to the states and graph productions to the operations or transformations of such systems (EHRIG; LÖWE,

[1]Appendix C presents the categorical definitions used along this text. Additional material on Category Theory can be found in (MACLANE, 1998), (PIERCE, 1991), (BORCEUX, 1994) and (ASPERTI; LONGO, 1991).

1993). Concepts of parallel and distributed productions and derivations in the algebraic approach are very useful to model concurrent access, aspects of synchronization, and distributed systems based on local and global graphs (see, for example, (EHRIG; ROSEN, 1980), (LÖWE, 1991), (EHRIG; LÖWE, 1993), (KORFF, 1995), (TAENTZER, 1996a), (HECKEL, 1998), and (MONTANARI; PISTORE; ROSSI, 1999)).

1.3 Object-oriented systems

The principles behind the object-oriented paradigm — *data and code encapsulation*, *information hiding*, *inheritance* and *polymorphism* — fit very well into the needs of modular system development, distributed testing, and reuse of software faced by system designers and engineers, making it perhaps the most popular paradigm of system development in use nowadays. The most distinguished features of object-oriented systems are inheritance and polymorphism, which make them considerably different from other systems in both their architecture and model of execution.

Inheritance is a central mechanism in object-oriented programming. It supports the incremental design of classes (in the *class-based approach* (COOK, 1990)) or objects (in the *object-based approach* (UNGAR et al., 1991)). Inheritance allows that new classes do not need to be written from scratch, but to be built upon already existing ones. This process generates a hierarchy of classes, where a heir class is a refined version of its ancestor or ancestors, depending whether single or multiple inheritance is used. A heir class also inherits all the features (data and operations) of its parent class to which it may add new ones or modify the operations already there. The resulting extensibility of classes support major requirements posed by stepwise development and reuse of code (BREU, 1991).

Along with the fundamental notion of inheritance, comes the other central feature of the object-oriented paradigm, which is *polymorphism*. Polymorphism can be described as the characteristic of being able to assign a different meaning or usage to something in different contexts. There are several different kinds of polymorphism used in the programming language setting, and the one we refer to is the so-called *subclass polymorphism* (CARDELLI; WEGNER, 1985): an object can be viewed as belonging to many different classes that need not to be disjoint. Subclass polymorphism specifies that an instance of a subclass can appear wherever an instance of a superclass is required. Polymorphism, in this context, can be exemplified by the following situation: if two classes x and y are related by the inheritance hierarchy, with x being a descendant (subclass) of y, then in any place an object of the class y is expected, an object of the class x can appear, since an object of the class x is also an object of the class y.

Subclass polymorphism becomes especially important in object-oriented programs when *dynamic binding* is implemented. Dynamic binding is a request-handling mechanism that selects the method to be called based on the type of the target object. So, the code executed to perform a given operation is determined at run time from the message receiver class. This allows the specification of one request that can result in the invocation of different methods depending on the type of the target object, which typically can only be determined at run time. Dynamic binding permits that classes may be created to receive a particular message, without changing (or recompiling) the code which sends the message. Most object-oriented languages support the selection of the appropriate method based on the class of the actual receiver object (CAMPIONE; WALRATH; HUML, 2000), (STROUSTUP, 2000), (THOMAS; WEEDON, 1997).

There is a plethora of formal and semi-formal methods proposed in the literature for the specification of object-oriented systems. Object-oriented programs usually make use of inheritance (which is the most common way of code reuse) and method redefinition to achieve their goals. Therefore, it should be expected that formalisms for the specification of object-oriented architectures or programs reflect those concepts, otherwise the use of such formalisms will neglect concepts that have a major influence in their organization and model of execution.

Object-oriented system modeling and programming approaches should present a number of desired properties, amongst which we cite the following: (i) the existence of a formal specification language which can be easily understood by both software developers and final users; (ii) the possibility of systems static and dynamic aspects be specified in an integrated way (i.e., by the same formalism); (iii) the existence of a formal semantical basis, allowing the composition of modular specifications in a consistent and significant manner; (iv) the possibility of high level specifications be refined into lower ones, or even into actual programs.

It has been known for a long time that the majority of systems under development are not formally specified because most formal specifications are difficult to understand and produce. Software development companies must provide the market with innovations and interesting products, in order to survive financially. This rush naturally demands a high velocity on the development cycle, which is, most of the time, incompatible with detailed formal descriptions which are not understood by programmers anyway. So, one of the more demanding challenges is to produce formal methods which can be understood by all participants on the software development process. We believe graph grammars can help achieving these goals.

Graph grammars have been used to specify various kinds of software systems, where graphs correspond to states and graph productions to operations or transformations of such systems (EHRIG; LÖWE, 1993). System specifications through graphs often rely on labeled graphs or typed graphs to represent different system entities (ANDRIES et al., 1999), (BLOSTEIN; FAHMY; GRBAVEC, 1995), (CORRADINI; MONTANARI; ROSSI, 1996), (DOTTI; RIBEIRO, 2000), (KORFF, 1995), (MONTANARI; PISTORE; ROSSI, 1999), (RIBEIRO, 1996), (TAENTZER, 1996a). However, neither labeling nor typing reflect the inheritance relation among objects, and polymorphism cannot be applied if it is not made explicit. To do so, a class, if represented by a node or edge in a graph, should have a multiplicity of labels or types assigned to it, representing all its ancestors, concerning the inheritance relation on classes. However, this is not compatible with the usual way labeling or typing are defined (as functions).

A very natural way of modeling object-oriented systems through graphs is by representing objects (or classes) as nodes and attributes and messages as arrows. Chapter 2 shows that objects have a natural order relation connecting them, given by the inheritance hierarchy. Additionally, message redefinition (overriding) relation also determines an order relation over the methods of any class organization. It will be shown in Section 2.2 that the reflexive and transitive closure of both inheritance and overriding relations is a partial order. Furthermore, the way messages are allowed to be redefined assures that, if objects are represented as graph nodes and messages as graph edges, the source and target functions must be order-preserving (with respect to those relations), i.e., monotone functions.

Characterizing objects, attributes and methods this way creates a situation where graphs are no longer defined over sets and functions, but over partially ordered sets and monotone functions. The very abstract way that graphs and graph morphisms are dealt with within category theory can be maintained by moving from diagrams in the category **SetP** of sets and partial functions to another suitable category of partially ordered sets and partial monotone functions. The algebraic approach to graph grammars rely on categorical constructs, especially on colimits, to express most of its results. Having graphs expressed in a category other than **SetP** is useful, in the sense that if the constructs used within the theory of graph grammars can be proven to exist in the new setting, the conclusions drawn from the former could be automatically transferred to the latter.

1.4 Model checking

Model checking was first introduced in (CLARKE; EMERSON; SISTLA, 1986), and it is a fully automatic technique to prove that a model of a concurrent program — specified as a finite transition system containing all possible behaviours of that program model — possess a property, specified in some temporal logic (EMERSON, 1998), (STIRLING, 1992). Temporal logics differ from the so-called *classic* logics (propositional, first-, second- and higher order logics) in the sense that a formula is not *statically* true or false within the model it is interpreted upon. Instead, models for temporal logics contain a number of different states and a formula can be true in some states and false in others (although, being a logic, given any state, a formula is either true or false in that state). Hence, the static notion of truth is replaced by a *dynamic* one, in which the formulas may change their truth values as the system evolves from state to state.

It was Amir Pnueli (PNUELI, 1977), in the late seventies, who argued that temporal logics could be a useful formalism for specifying and verifying correctness of computer programs. Temporal logics are specially suitable for reactive systems, i.e., systems that are in general non terminating and continuously interact with an environment. Reactive systems, therefore, differ from systems which behave as a function (i.e., compute a result from a specific input), and must terminate for all possible inputs to be considered correct. Many systems can be modeled as finite state machines governed by transition relations. We can then express properties about the state space as formulae in a temporal logic.

Since the beginning of the research in model verification, the theory of automated verification, and the design and construction of efficient verification algorithms and tools has evolved dramatically. Seminal steps for model checking techniques that are today available in tools such as SPIN (HOLZMANN, 1991), (HOLZMANN, 1997) were the introduction of logic model checking techniques in 1983, independently by Emerson and Clarke (CLARKE; EMERSON, 1981) and Jean-Pierre Quielle and Joseph Sifakis (QUIELLE; SIFAKIS, 1981), the development of the automata theoretic framework for verification, jointly by Moshe Vardi and Pierre Wolper (VARDI; WOLPER, 1986), and great improvements in efficient graph-search algorithms and memory-management techniques by many others.

The truth value of a formula could in principle be determined by exploring the reachability graph of the state space. The term *model checking* means to explore a finite state space to establish properties of the system. However, the state spaces arising from practical problems are often huge (typically exponential in the number of variables), so exhaustively exploring the state space is not generally feasible. What is done is to use an implicit representation of the state space, where the reachability relation is represented as a collection of boolean formulae. The truth values of temporal formulae are tested by a series of operations on the boolean formulae as opposed to an explicit search. This is referred to as symbolic model checking (ANDERSON et al., 1996). Symbolic model checking has been implemented by a large number of tools. The more recent introduction of binary decision diagrams (BDDs) (HUTH; RYAN, 2000) in hardware model checking, and of aggressive partial-order reduction techniques in software verification has helped still further to make more general applicability of automated verification techniques a distinct reality.

A wide range of sophisticated verification tools is now available. Those tools typically have an input language, and a specification written in that input language is transformed into a model suitable for verification. There are several variants of logic model checking tools that are traditionally focused on hardware verification tasks. These tools are powerful enough to perform full CTL (computational tree logic) or LTL (linear temporal logic) model checking for very substantial industrial applications, such as communications protocol design and distributed systems design. Recently, however, a new focus has been given to model check actual programs (VISSER et al., 2003).

There are two different approaches to verify formally described systems using model checking: the first one is to build a model checker to generate models written in the specification language, and the other one is to make use of existing model checkers by translating the language used to describe the system to the chosen model checker input language. Naturally, the translation must be correct, in the sense that the behaviour (semantics) of the systems must be preserved by the translation (at least concerning the properties being specified). The development of (correct) model checkers is a task which requires a large amount of time and effort. Furthermore, good model checkers not only perform model generation and property verification: techniques such abstraction (SCHMIDT, 2002), program slicing (HATCLIFF; DWYER; ZHENG, 2000), (KRINKE, 2003) and partial order reductions (ALUR et al., 2001) are necessary to verify systems with an excessive state space size. Hence, the second approach is usually simpler.

1.5 Thesis contributions and outline

Object-oriented systems play a major role on the way software systems are built nowadays. Recalling what was said in Section 1.1, there is a large number of formal and semi-formal methods proposed in the literature aiming the specification of object-oriented systems. Our purpose is neither to present a survey of them nor to indicate a "better" solution for such problem. It is generally believed that a method for the specification of software should be chosen based on the problem at hand, and that any method has almost always advantages and disadvantages when compared to any other one. Hence, the contribution of this thesis is focused on the area of graph grammars, and in their use for the specification and verification of object-oriented systems. The general goal of this work can be summarized as

> *To present a graph grammar-based formalism for the specification of object-oriented systems which provides adequate support to the verification of properties and reasoning about models written in that formalism.*

This goal has been achieved through (i) the development of an extension of the single-pushout approach to graph grammars which encompass the idiosyncrasies of the object-oriented paradigm for system development and programming, and (ii) a formally defined translation of models written on that formalism to programs written in Promela (which is the input language for the model checker SPIN), which allows the verification of (LTL) properties of system states (graphs) and events (application of graph productions).

The main contributions of this thesis are listed below:

1. the development of an extension of the single-pushout approach to graph grammars to encompass the idiosyncrasies of the object-oriented paradigm for system development and programming. This new graph transformation approach, called "object-oriented graph grammar" presents the following characteristics:
   - object-oriented systems static and dynamic aspects can be described in an integrated way;
   - the semantics of computations given by the graph grammar is compatible with the one of object-oriented systems;
   - static specifications of systems can be consistently combined and extended;
   - specifications can be refined into lower level specifications or even into actual programs;
   - the resulting formalism can be easily used/understood by system developers, programmers and final users.
2. a formally defined translation from object-oriented graph grammars into Promela source code. The translation aims at providing a means for the verification of properties of object-oriented systems expressed by object-oriented graph grammars. The translation and verification procedures account for:
   - the semantics of the object-oriented graph grammar is preserved by the translation, in the sense that no behaviour is introduced or removed concerning the original model;
   - the final user do not need to understand the translation itself, the Promela language, or how the final model is built by the model checker;
   - linear temporal logics (LTL) formulae can be expressed in terms of the elements belonging to the initial graph;
   - temporal properties can be stated in terms both of states (graphs) and events (rule applications) occurring during the computation.

Given a system specification, we can be interested in two things: the system itself and its behaviour. The first part of this work provide the means to build (executable) system specifications; the second one provides the tools for the analysis of systems built through them.

The rest of this text is structured as follows:

- Chapter 2 relates the most fundamental characteristics of the object-oriented programming paradigm with partial orders, and introduces the algebraic graph structure over which the graphs and grammars presented in this thesis will be typed. Those structures — class-model graphs — model object-oriented specifications. System composition is defined in terms of special colimits in the category of class-model graphs and class-model graph morphisms. Object-oriented features such as class inheritance and aggregation are shown to be special kinds of system composition within this framework.

- Chapter 3 presents object-oriented graphs and object-oriented graph grammars. Object-oriented graphs are (hyper)graphs typed over class-model graphs, which preserve the underlying relations on nodes (inheritance) and edges (overriding). Object-oriented rules are constructed following the principles of data encapsulation and information hiding from the object-oriented paradigm. Object-oriented matches ensure that subclass polymorphism do exist. It is shown that object-oriented graphs and their morphisms form a category, and that direct derivations on object-oriented graph grammars are pushouts in that category. It is also shown that an object-oriented derivation reflects the semantics of dynamic binding accurately.

- Chapter 4 formally defines a translation between an object-oriented graph grammars and a Promela (the input language for the model checker SPIN) program. Objects are translated to asynchronous processes, and method invocation to messages shared between processes. It is also shown how the semantics of the grammar computations is preserved by the translation. As a case study, Dijkstra's *Dining Philosophers problem* is programmed as an object-oriented graph grammar, and properties are checked over the translated Promela program.

- Chapter 5 presents a survey of the related work found in the literature.

- Chapter 6 draws some conclusions about the results achieved by this thesis, and lists possible developments for the work presented here.

- The main chapters only present the mathematical proofs of the main results achieved, for the sake of readability. The proofs of lemmas and theorems not shown in the text can be found in Appendix A. Appendices B, C, and D present, respectively, the definitions and theorems from order theory, category theory and graph theory used in this text. Appendix E describes the languages used in the model checker SPIN: it portrays the formal syntax of Promela and the syntax and semantics of the temporal language LTL. The complete translated program source code for the Dining Philosophers problem are also listed there. Finally, Appendix F presents a summary of the main results achieved by this thesis (in portuguese).

2 OBJECT-ORIENTED SPECIFICATIONS

2.1 Introduction

Order relations[1] in general and partial order relations in particular arise throughout the field of Computer Science. As examples one can cite causal dependencies of process actions in parallel computing, subtyping relations among primitive and user-built types in programming languages, prefix relations on elements of any kind, approximations, information orderings, and so on. Other examples can be drawn from the areas of information theory, abstract interpretation, real arithmetic, semantics, domain theory, among others (DAVEY; PRIESTLEY, 2002).

The same occurs with object-oriented programming. To see the existing relationship between object orientation and partial order relations, let us examine the main features of the paradigm:

Inheritance — Inheritance is the construction which permits a class (in the *class-based approach* (COOK, 1990)) or an object (in the *object-based approach* (UNGAR et al., 1991)) to be specialized from an already existing one. This newly created object carries (or "inherits") all data and actions belonging to its primitive object, in addition to its own data and actions. If this new object is further extended using inheritance, then all the new information will also be carried along. The relation "inherits from" induces a hierarchical relationship among the defined classes of a system, which can be viewed as a set of trees (single inheritance) or as an acyclic graph (multiple inheritance). The inheritance hierarchy also possesses a transitive nature: if a class c is an extension of another class c', which is yet an extension of a third class c'', it means that c is also an extension of class c''. It is also antisymmetric, for it is necessary that a class have already been defined *before* it is extended. Although the idea of a relation "inherits from" is not reflexive per se, it makes sense to allow such relation to be reflexive, in the sense that inheritance means that a class owns all attributes and methods their primitives do, and naturally it possess its own atttributes and methods. So, the (single or multiple) inheritance hierarchy can be formally characterized as a partial order relation.

Polymorphism — Polymorphism arises when we can assign a different meaning or usage to something in different contexts. There are several different types of polymorphism implemented in actual object-oriented programming languages. Subclass polymorphism (CARDELLI; WEGNER, 1985) is perhaps one of the

[1]All formal definitions concerning order relations can be found in Appendix B.

main features of the paradigm, and it is specially relevant within this work. Subclass polymorphism specifies that an instance of a subclass can appear wherever an instance of a superclass is required. In order to apply it, however, the inheritance relationship must be made explicit by any formalism aiming to model object-oriented systems. One of the most useful ways inheritance and polymorphism can be used is through method overriding, also known as method redefinition. Subtyping and subclassing relations were somewhat mixed together in the past, but now they are seen as having similar (although not identical) structures (COOK; HILL; CANNING, 1989), (BRUCE; FIECH; PETERSEN, 1997), (MITCHELL; VISWANATHAN, 1996), (POLL, 1997). Both relations, however, have a clear partial order structure, although in an opposite direction: it is easy to see that both subtyping and subclassing are reflexive, transitive and antisymmetric relations.

Method overriding — The purpose of method overriding is to take advantage of the polymorphism concept through the mechanism known as *dynamic binding* (MITCHELL; APT, 2001), which is an execution time routine to determine what piece of code should be called when a message is sent to an object. Notice that a class or an object can only override a method if it exists somewhere along the chain (respecting the inheritance relation) from itself to the one which has the method being redefined. Additionally, the signature of both methods must be exactly the same. One can easily see that method overriding also obeys an order (transitive) relation, which naturally follows the inheritance relation.

Inheritance, *polymorphism*, and *method overriding*, together with data and code *encapsulation* and *information hiding* form the very core of the object orientation paradigm to software development. As seen in the latter paragraphs, three of them can be expressed as order relations. It is only natural that specification languages to model such systems make order relations the formal basis for its syntactic and semantic definitions, especially considering that order relations are a well studied domain of mathematics (ABRAMSKY; JUNG, 1994), (STOLTENBERG-HANSEN; LINDSTRÖM; GRIFFOR, 1994), (FIECH, 1996), (DAVEY; PRIESTLEY, 2002). Having order — and specially partial order — relations as a semantic foundation for object-oriented constructs allows the reuse of all knowledge of this mathematical field, which can aid the development of new theories.

This chapter is structured as follows: Section 2.2 presents *strict relations*, which intend to reflect both the hierarchic structure of classes and method overriding as defined in most object-oriented programming languages. It is also shown that the reflexive and transitive closure of such relations is a partial order. Section 2.3 develops the theory regarding the graphs used to portray object-oriented systems structure of classes. It is shown that those graphs, called *class-model graphs*, and their morphisms constitute a category. Finally, Section 2.4 shows how system composition, inheritance and aggregation of objects can be seen as particular cases of colimits in that category.

2.2 Strict relations

Different programming languages implement different object hierarchic structures. Languages which only allow single inheritance, differ in the sense that all classes have a common primitive class (such as the class *Object*, in Java (CAMPIONE; WALRATH; HUML, 2000) and Delphi (LISCHNER, 2000)) or not (such as Ada95 (BARNES, 1998), or Glass (MUHAMMAD; FERREIRA, 2003)). The same happens to languages that allow multiple inheritance, with a top element (Eiffel (MEYER, 1991)) or without it (C++ (STROUSTUP, 2000)).

Sets of trees or acyclic graphs, with or without a top element, can formally characterize the inheritance relation of classes/objects created in such programming languages. The definition of a *strict relation*, given below, formalizes what should be the fundamental object-oriented hierarchic structure of classes, when only single inheritance is allowed.

Definition 2.1 (Strict relation) *A binary relation $R \subseteq A \times A$ is said a* strict relation *if and only if it has the following properties:*

1. *if $(a, a') \in R$ then $a \neq a'$ (R is irreflexive);*
2. *if $(a, a_1), (a_1, a_2), \ldots, (a_{n-1}, a_n), (a_n, a') \in R$, $n \geqslant 0$, then $(a', a) \notin R$ (R is acyclic);*
3. *for any $a, a', a'' \in A$, if $(a, a'), (a, a'') \in R$ then $a' = a''$ (R is a function);*
4. *for each $a \in A$, either $(a, b) \notin R$ for all $b \in A$ or there exists $n \geqslant 0$ such that if $(a, a_1), (a_1, a_2), \ldots, (a_{n-1}, a_n) \in R$, then $(a_n, b) \notin R$ for all $b \in A$ (all chains in R are finite).*

Notice that the requirement concerning the absence of cycles and reflexive pairs on strict relations is consistent with both the creation of classes and redefinition of methods (overriding). A class, to be defined as a specialization of one or more classes, requires that the primitive classes exist prior to its creation. Similarly, a method can only redefine another method (with the same signature) if it exists in an already existing primitive class. Hence, neither a class can ever be created nor a method can be redefined in a circular or reflexive way. The third condition guarantees that single inheritance is implemented, i.e., a class is a specialization of at most another one. Dropping that condition suffices to represent the fundamental inheritance hierarchy of object-oriented languages which implement multiple inheritance. The fourth condition has a theoretical significance, but it is meaningful in the sense that cannot exist an infinite chain of classes from which one class is derived. Object-oriented specifications are always finite (as all real objects are), but finiteness of set A is a too strong assumption, and we want to be as less restrictive as possible. For our purposes, it suffices that there are no infinite chains in R.

The choice of representing only single inheritance in this work has to do with the fact that most object-oriented programming languages only implement single inheritance, which usually makes object-oriented specifications follow the same lines. It should not be hard, however, to extend this framework to multiple inheritance. The only difference would be that some restrictions could be waved out, making the proofs and definitions slightly different. Figure 2.1 presents an example of an strict relation. The names constitute the base set A and the dashed arrows represent the relation R itself.

If strict relations represent the hierarchic structure of inheritance and method redefinition relations, it would be of interest to investigate some of its properties,

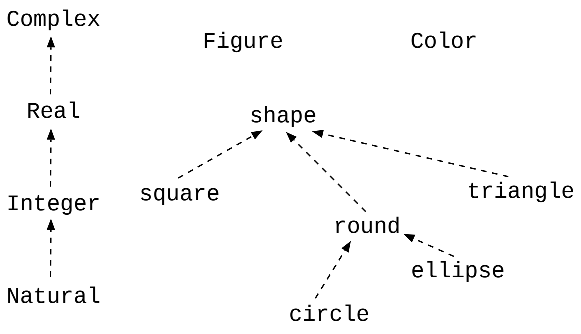

Figure 2.1: Example of a strict relation

which will be important in what follows. The first and more important one is given by Lemma 2.1, stated below.

Lemma 2.1 *The reflexive and transitive closure of a strict relation is a partial order relation.*

Proof: On page 125. □

Strict relations will be used to represent two distinct hierarchic structures: inheritance and overriding. Since they represent fundamental characteristics of the object-oriented paradigm, mappings between them must be compatible with mappings of pertinent parts of object-oriented specifications. Those mappings, which are needed to define a number of things such as extension and composition, are called *strict ordered functions*, whose domain and codomain of definition are *strict ordered sets.*

Definition 2.2 (Strict ordered set) *A* strict ordered set $P_\sqsubseteq$ *is a pair* $\langle P, \sqsubseteq_P^* \rangle$ *where* P *is a set,* $\sqsubseteq_P$ *is a strict relation, and* $\sqsubseteq_P^*$ *is its reflexive and transitive closure.*

Strict ordered sets are a special case of partially ordered sets, where the underlying partial order relation was generated as the reflexive and transitive closure of a strict relation.

Notation: In what follows, notation from order theory is used: for any partially ordered set $\langle P, \sqsubseteq_P \rangle$ (Definition B.9), and all $p \in P$ and $A \subseteq P$, $\uparrow p$ denotes the *upper set* of p (Definition B.11), $\downarrow p$ denotes the *lower set* of p (Definition B.11), $\sqcup A$ denotes the supremum of A (Definition B.14), and $\sqcap A$ denotes the infimum of A (Definition B.14). Also, a little abuse of notation is carried out: a function $f : D \to I$ applied to a set $S \subseteq D$ is defined as $f(S) = \{f(s) \in I \mid s \in S\}$.

Definition 2.3 (Strict ordered function) *Let* $P_\sqsubseteq = \langle P, \sqsubseteq_P^* \rangle$ *and* $Q_\sqsubseteq = \langle Q, \sqsubseteq_Q^* \rangle$ *be two strict ordered sets. A partial monotonic function* $f : P_\sqsubseteq \to Q_\sqsubseteq$ *is a* strict ordered function *if and only if for all elements* $x \in dom(f)$*, we have that* $\uparrow x \subseteq dom(f)$ *and* $f(\uparrow x) = \uparrow f(x) \cap \downarrow f(\sqcup \uparrow x)$*.*

The restrictions imposed to a strict ordered function are related to the mapping coherence between the strict ordered sets underlying relations. Specifically, if an element is mapped by a strict ordered function, then *all* elements from the chain to

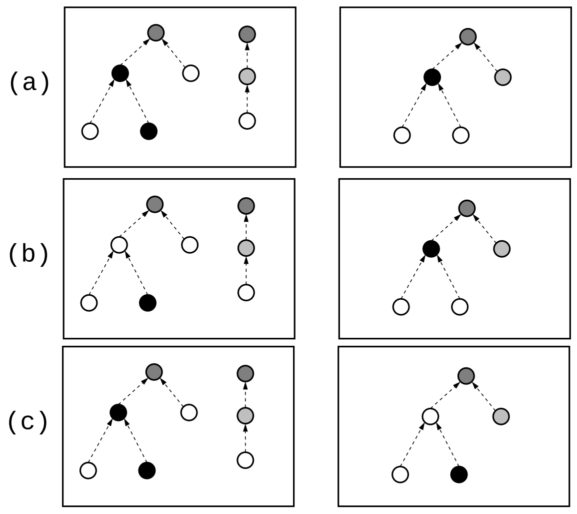

Figure 2.2: Example of strict and non strict ordered functions

which it belongs (respecting the strict relation on its base set) must also be mapped accordingly. Moreover, they must be an exact match to the image set. It means that the mapping must be some sort of "glue" of structures, as shown in Example 2.1.

Example 2.1 (Strict ordered function) *The idea of a strict ordered function is to map two strict ordered sets in such a way that the mapping constitute a gluing of the structures. Since a strict ordered function is partial, if an element is mapped, all elements which are above it in the domain relation must be mapped in such a way that no "holes" appear on the mapped domain. Formally, this means that the function must be surjective regarding the image elements of the function between the bottom and top elements of the domain chain.*

Since strict relations have a functional structure, the upper set of any element is a chain (Definition B.10), which must be completely mapped onto the image chain which contains all elements between the image of the bottom and the top elements of upper set already mentioned. Figure 2.2 shows three examples of functions defined on the same domain and image, and denoted respectively by the (a), (b) and (c) letters in the figure. In each set (the one on the left being the domain and the one on the right the image of the function), the mapping provided by each function is represented by the colors on the set elements. All white elements do not belong to the domain of the function, while the ones colored in black or shades of gray are. Functions (b) and (c) are not strict ordered functions. The function in (b) has an element on the upper set of the element colored as black on the left not mapped by the function (the one right above it, colored in white). Function (c) is not a strict ordered function because one element is missed on the image of the chain formed between the image of the bottom and the top elements of the black element previously referred.

The upper set of any element is indeed a (finite) chain, as Lemmas A.1 and A.2 assure, and therefore has both a least (infimum) — the element itself — and a greatest (supremum) element — the primitive class from where all its ancestors derive. This restriction is needed to assure that the strict relation structure is maintained when the sets are combined. Before showing how this combination can be performed, however, some properties of strict ordered functions will be shown, together with the proof that strict ordered sets and strict ordered functions constitute a category.

Property 2.1 *Strict ordered functions are closed under composition.*

Proof: Let $f : P_{1\sqsubseteq} \rightarrow P_{2\sqsubseteq}$ and $g : P_{2\sqsubseteq} \rightarrow P_{3\sqsubseteq}$ be two strict ordered functions. Let $x \in dom(g \circ f)$. Then $x \in dom(f)$ and $f(x) \in dom(g)$. Since both f and g are strict ordered functions, $\uparrow x \subseteq dom(f)$ and $\uparrow f(x) \subseteq dom(g)$. According to Lemma A.1, $\uparrow x$ is chain, and so there exists a chain of elements $x = x_1 \sqsubseteq_1 x_2 \sqsubseteq_1 \ldots \sqsubseteq_1 x_n = \sqcup \uparrow x$ all belonging to the domain of f. The image, under f of such set is given by the chain of elements (since f is monotonic) $f(x) = f(x_1) \sqsubseteq_2 f(x_2) \sqsubseteq_2 \ldots \sqsubseteq_2 f(x_n) = f(\sqcup \uparrow x)$. But all those elements belong to the upper set of that chain bottom element, $\uparrow f(x)$, which means (since g is a strict ordered function) that $f(\uparrow x) = \{f(x) = f(x_1), f(x_2), \ldots, f(x_n) = f(\sqcup \uparrow x)\} \subseteq dom(g)$. Hence, $g(f(\uparrow x))$ is defined, and therefore $\uparrow x \subseteq dom(g \circ f)$.

By definition, $(g \circ f)(\uparrow x) = g(f(\uparrow x) = \{g(f(a)) \mid a \in \uparrow x\}$, which is the set $\{g(f(x)) = g(f(x_1)), g(f(x_2)), \ldots, g(f(x_n)) = g(f(\sqcup \uparrow x))\}$ with $g(f(x)) = g(f(x_1)) \sqsubseteq_3 g(f(x_2)) \sqsubseteq_3 \ldots \sqsubseteq_3 g(f(x_n)) = g(f(\sqcup \uparrow x))$, which is trivially contained in the set $\uparrow g(f(x))$. Since $g(f(\sqcup \uparrow x)) \sqsubseteq_3 \sqcup \uparrow g(f(x))$ (the latter exists because of Lemma A.2), then the image set $g(f(\uparrow x))$ is also contained in the set $\downarrow (g \circ f)(\sqcup \uparrow x)$, and so $g(f(\uparrow x)) \subseteq \uparrow (g \circ f)(x) \cap \downarrow (g \circ f)(\sqcup \uparrow x)$. But Lemma A.4 guarantees that there are no elements between $g(f(x))$ and $g(f(\sqcup \uparrow x))$ which do not belong to $g(f(\uparrow x))$. Hence, $(g \circ f)(\uparrow x) = \uparrow (g \circ f)(x) \cap \downarrow (g \circ f)(\sqcup \uparrow x)$. □

Property 2.2 *The identity function is a strict ordered function.*

Proof: The identity function, for any domain set D is defined as $id(x) = x$ for all $x \in D$. Since it is a total function (defined for all elements $x \in D$, then trivially $\uparrow x \subseteq dom(f)$.

For id to be a strict ordered function, then for all $x \in D$ we need that $id(\uparrow x) = \uparrow id(x) \cap \downarrow id(\sqcup \uparrow x)$. But $id(\uparrow x) = \{id(a) \mid a \in \uparrow x\} = \{a \mid a \in \uparrow x\} = \uparrow x$; $\uparrow id(x) = \uparrow x$; and $\downarrow id(\sqcup \uparrow x) = \downarrow (\sqcup \uparrow x)$. Since $\uparrow x \subseteq \downarrow (\sqcup \uparrow x)$, then $\uparrow x \cap \downarrow (\sqcup \uparrow x) = \uparrow x = id(\uparrow x)$. □

Theorem 2.1 (Category SOSet) *There is a category* **SOSet** *which has strict ordered sets as objects and strict ordered functions as arrows.*

Proof: Strict ordered functions are closed under composition (Property 2.1), and their composition is associative, since monotonic partial functions composition is associative, and strict ordered functions are just a a special case. For any strict ordered set $\langle P, \sqsubseteq_P^* \rangle$, let $id_P : P \rightarrow P$ be the trivial identity morphism $id_P(x) = x$ for all $x \in P$, which is itself a strict ordered function by Property 2.2. Then, for all monotonic partial function $f : P \rightarrow Q$ we have that $(f \circ id_P)(x) = f(id_P(x)) = f(x) = y = id_Q(y) = id_Q(f(x)) = (id_Q \circ f)(x)$ if $x \in dom(f)$ and $(f \circ id_P)(x) = f(id_P(x)) = f(x) = undef = id_Q(f(x)) = (id_Q \circ f)(x)$ otherwise. □

Two important categorial constructions do exist in **SOSet**. The following theorems state that the category has an initial objects and arbitrary coproducts.

Theorem 2.2 (Initial object in SOSet) *The category* **SOSet** *has an initial object.*

Proof: On page 125. □

Theorem 2.3 (Coproducts in SOSet) *The category* **SOSet** *has all finite and infinite coproducts.*

Proof: On page 126. □

Unfortunately, the category of strict relations and strict ordered functions is not cocomplete. Although it has an initial object (Theorem 2.2) and generalized coproducts (Theorem 2.3), category **SOSet** does not have all coequalizers. It is easy to show a counterexample: let $A_{\sqsubseteq} = \langle A, \sqsubseteq_A^* \rangle$ and $B_{\sqsubseteq} = \langle B, \sqsubseteq_B^* \rangle$ be two strict ordered sets, where $A = \{a\}$, $\sqsubseteq_A = \emptyset$, $B = \{x, x', y, y'\}$ and $\sqsubseteq_B = \{(x, x'), (y, y')\}$; let $f : A \to B$ and $g : A \to B$ be two strict ordered functions with $f(a) = x$ and $g(a) = y$. A coequalizer of two arrows in categories of sets and (partial) functions is built as the quotient set by the least equivalence class which contains all pairs $(f(x), g(x))$, excluding the elements which are mapped by only one function (Definition C.15). Therefore, the resulting coequalizer object would contain three equivalence classes — namely $[x']$, $[y']$ and $[x]$ (which equals $[y]$) — with relation $\sqsubseteq = \{([x], [x']), ([x], [y'])\}$. This is clearly not a strict ordered set, since the order relation is not functional, and therefore not a strict relation.

System composition can usually be defined categorically in terms of a colimit construction. Colimits are constructed from coproducts and coequalizers, and if coequalizers cannot be always built this could present a problem: the definition of system composition will not be as elegant and general as one should expect. However, general colimits are indeed a too strong assumption, since there are restrictions in the way object-oriented systems can be meaningfully combined.

We will show next that a restricted form of diagram in **SOSet** has always a colimit. That will suffice for our purposes.

Theorem 2.4 (Special colimits in SOSet) *Let* **D** *be a diagram in* **SOSet** *containing two objects $R_{1\sqsubseteq}$ and $R_{2\sqsubseteq}$, and one morphism $f : R_{1\sqsubseteq} \to R_{2\sqsubseteq}$. Then* **D** *has a colimit, represented by object $C_{\sqsubseteq}$ and morphisms $c_i : R_{i\sqsubseteq} \to C_{\sqsubseteq}$ for $i = 1, 2$.*

Proof: Let $R_{\uplus\sqsubseteq} = R_{1\sqsubseteq} \uplus R_{2\sqsubseteq}$ be the disjoint union of the strict ordered sets $R_{1\sqsubseteq}$ and $R_{2\sqsubseteq}$, where $R_{\uplus\sqsubseteq} = \langle R_1 \uplus R_2, \sqsubseteq_{R_1} \uplus \sqsubseteq_{R_2} \rangle$. Let $C \subseteq \mathscr{P}(R_{\uplus})$ be the quotient set of $R_{\uplus}$ by the least equivalence relation which contains all pairs $(x, f(x)) \in R_{\uplus} \times R_{\uplus}$, and let $c : R_{\uplus} \to C$ be the projection of $R_{\uplus}$ onto C. For sets R_i, $i = 1, 2$, let $\iota_i : R_i \to R_{\uplus}$ be the inclusion functions $\iota_i(x) = x$ for all $x \in R_i$. Let also $\sqsubseteq_C = \{([x], [x']) \in C \times C \mid y \in [x], y' \in [x'], y \sqsubseteq_{R_{\uplus}} y'\}$. Then the tuple $\langle C_{\sqsubseteq} = \langle C, \sqsubseteq_C \rangle, \{c_i = c \circ \iota_i : R_{i\sqsubseteq} \to C_{\sqsubseteq}\} \rangle$ is a colimit of diagram D in **SOSet**.

In order to prove that this is indeed a colimit, we have to prove that (i) $C_{\sqsubseteq}$ is a strict ordered set, (ii) $c : R_{\uplus} \to C$ is a strict ordered function, (iii) $c_1 = c \circ \iota_1$ and $c_1 = c \circ \iota_1$ are also strict ordered functions, and (iv) for any other strict ordered set $D_{\sqsubseteq}$ and strict ordered functions $d_1 : R_{1\sqsubseteq} \to D_{\sqsubseteq}$ and $d_2 : R_{2\sqsubseteq} \to D_{\sqsubseteq}$ such that $d_1 = d_2 \circ f$ there is a unique strict ordered function $h : C_{\sqsubseteq} \to D_{\sqsubseteq}$ such that $d_1 = h \circ c_1$ and $d_2 = h \circ c_2$.

(i) We will prove that $C_{\sqsubseteq}$ is a strict ordered set by proving that $\sqsubseteq_C$ is a strict relation. Since it is (by construction) reflexive and transitive, it remains to show that it is functional and antisymmetric.

Suppose $\sqsubseteq_C$ is not functional. Then there must exist equivalent classes $[x], [x'], [x''] \in C$ such that $([x], [x']) \in \sqsubseteq_C$, $([x], [x'']) \in \sqsubseteq_C$, and neither $([x'], [x'']) \in \sqsubseteq_C$ nor $([x''], [x']) \in \sqsubseteq_C$ (so, we are inspecting the generative pairs, excluding the transitive ones). Therefore (and by construction), it must be the case that there are $a, a', b, b' \in R_{\uplus}$ such that $a, b \in [x]$, $a' \in [x']$, $b' \in [x'']$, and $(a, a'), (b, b') \in \sqsubseteq_{R_{\uplus}}$.

If $(a, a'), (b, b') \in \sqsubseteq_{R_\uplus}$, then $a, a' \in R_i$ and $b, b' \in R_j$, for some $i, j \in \{1, 2\}$, because relation $\sqsubseteq_{R_\uplus}$ is a disjoint union. Assume, without loss of generality, that $(a, a') \in \sqsubseteq_{R_1}$ and $(b, b') \in \sqsubseteq_{R_2}$ (if they are in the same set, their equivalence classes will be all related in the same chain). If both a and b end up in the same equivalence class, then $f(a) = b$. But, if $a \in dom(f)$ and $a' \in \uparrow a$, then $a' \in dom(f)$. Since $f(\uparrow a) = \uparrow f(a) \cap \downarrow f(\sqcup \uparrow a)$, and given (by Lemma A.3) that we know that $\uparrow f(a) = \uparrow b$ is a chain, then either $(f(a'), b') \in \sqsubseteq_{R_2}$ or $(b', f(a')) \in \sqsubseteq_{R_2}$. Therefore either $([x'], [x'']) \in \sqsubseteq_C$ or $([x''], [x']) \in \sqsubseteq_C$, and $\sqsubseteq_C$ is functional.

Suppose $\sqsubseteq_C$ is not antisymmetric. Then there must exist two pairs $([x], [x']), ([x'], [x]) \in \sqsubseteq_C$ such that $[x] \neq [x']$. Therefore there must be elements $a, a', b,' b \in \sqsubseteq_{R_\uplus}$ such that $(a, a'), (b, b') \in \sqsubseteq_{R_\uplus}$, $a, b' \in [x]$ and $b, a' \in [x']$. But then again if $a, b' \in [x]$ then $f(a) = b'$, and since a strict ordered function is a partial order function (i.e., it preserves relations), then it is not possible that $f(a') = b$. Therefore, $\sqsubseteq_C$ is antisymmetric.

$\sqsubseteq_C$ is reflexive, transitive, functional and antisymmetric, therefore it is a strict ordered function, and $C_\sqsubseteq$ is a strict ordered set.

(ii) Before proving that function $c : R_{\uplus\sqsubseteq} \to \sqsubseteq_C$ is a strict ordered function, we will show that it is a partial order function, which preserves the underlying relation: for all $x \notin dom(f)$, the equivalence class of x, denoted by $[x]$ is the singleton $\{x\}$. For any other equivalence class $[y]$ in $C_\sqsubseteq$, $([x], [y]) \in \sqsubseteq_C$ if and only if $(x, y') \in \sqsubseteq_{R_\uplus}$ for some $y' \in [y]$; for all $x \in dom(f)$, then $x, f(x) \in [x]$, and for all y such that $(x, y) \in \sqsubseteq_{R_\uplus}$, $y, f(y) \in [y]$ and $([x], [y]) \in \sqsubseteq_C$.

c is (by construction) a total function, and therefore $\uparrow x \subseteq dom(c)$ for all $x \in R_{\uplus\sqsubseteq}$. Now, suppose that $c(\uparrow x) \neq \uparrow c(x) \cap \downarrow c(\sqcup \uparrow x)$. Then either $\exists y \in c(\uparrow x) : y \notin \uparrow c(x) \cap \downarrow c(\sqcup \uparrow x)$ or $\exists y \in \uparrow c(x) \cap \downarrow c(\sqcup \uparrow x) : y \notin c(\uparrow x)$. If $[y] \in c(\uparrow x)$ there must exist an x' such that $(x, x') \in \sqsubseteq_{R_\uplus}$ and $c(x') \in [y]$. And, if $[y] \notin (\uparrow c(x) \cap \downarrow c(\sqcup \uparrow x))$ then either $[y] \notin \uparrow c(x)$ or $[y] \notin \downarrow c(\sqcup \uparrow x)$. But $[x]$ is the bottom element of the chain $\uparrow c(x)$ ($C_\sqsubseteq$ is a strict ordered set, and therefore $\uparrow c(x)$ is a chain by Lemma A.1). Since $(x, x') \in \sqsubseteq_{R_\uplus}$, and because c is a order preserving function, $c(x') \in \uparrow c(x)$, and therefore (by construction) $[y] \in \uparrow c(x)$. Now, the top element of chain $\uparrow x$ is $\sqcup \uparrow x$. So, since $(x, x') \in \sqsubseteq_{R_\uplus}$, by definition $(x', \sqcup \uparrow x) \in \sqsubseteq_{R_\uplus}$. Again, because c is order-preserving, $(c(x'), c(\sqcup \uparrow x)) \in \sqsubseteq_C$, and therefore $[y] \in \downarrow c(\sqcup \uparrow x)$.

If $[y] \in (\uparrow c(x) \cap \downarrow c(\sqcup \uparrow x))$ but $[y] \notin c(\uparrow x)$, it means that there is an equivalence class $[z]$ in $\uparrow c(x) \cap \downarrow c(\sqcup \uparrow x)$ that is not in the image of c for its domain subset $\uparrow c(x)$. But c is total and surjective (by construction), therefore there must be an element $z \in R_{\uplus\sqsubseteq}$ such that $c(z) = [z]$. But if $[z] \in (\uparrow c(x) \cap \downarrow c(\sqcup \uparrow x))$, then $(c(x), [z]), ([z], c(\sqcup \uparrow x)) \in \sqsubseteq_C$. Then, there must be elements $z', z'' \in R_{\uplus\sqsubseteq}$, with $(z', z), (z, z'') \in \sqsubseteq_{R_\uplus}$ such that $c(z') \in [z']$, $c(z'') \in [z'']$, with $[z'] \neq [z] \neq [z'']$ such that $[z'], [z''] \in (\uparrow c(x) \cap \downarrow c(\sqcup \uparrow x))$. But, if two elements end up in the same equivalence class, it is because they have been mapped by f. Therefore, $z', z'' \in dom(f)$. If $[z] \notin c(\uparrow x)$, then $z \notin dom(f)$, which is impossible because f is a strict ordered function. Hence, $c(\uparrow x) = \uparrow c(x) \cap \downarrow c(\sqcup \uparrow x)$ for all $x \in R_{\uplus\sqsubseteq}$ and c is a strict ordered function.

(iii) Since c is a strict ordered function, and the inclusion functions are also strict ordered functions (Lemma 2.2), and because strict ordered functions are close under composition (Lemma 2.1), it can be concluded that $c_1 = c \circ \iota_1$ and $c_2 = c \circ \iota_2$ are both also strict ordered functions.

(iv) $c_1 = c_2 \circ f$ by construction. Let $D_\sqsubseteq$ be a strict ordered set. Let $d_1 : R_{1\sqsubseteq} \to D_\sqsubseteq$ and $d_2 : R_{2\sqsubseteq} \to D_\sqsubseteq$ be strict ordered functions such that $d_1 = d_2 \circ f$. Let $h : C_\sqsubseteq \to D_\sqsubseteq$ be the function defined as follows: for each $x \in dom(d_i)$, $h(c_i(x)) = d_i(x)$, and h is undefined for all other equivalence classes in $C_\sqsubseteq$. $d_1 = h \circ c_1$ and $d_2 = h \circ c_2$ by construction of h. Now, assume that there is another strict ordered function $h' : C_\sqsubseteq \to D_\sqsubseteq$ such that $d_1 = h' \circ c_1$ and $d_2 = h' \circ c_2$ but $h \neq h'$. c_1 and c_2 are jointly surjective and therefore if there is an equivalence class $[e] \in C_{\sqsubseteq}$ such that $h(e) \neq h'(e)$, $d_1 = h' \circ c_1$ and $d_2 = h' \circ c_2$ would not hold. $\square$

2.3 Class-model graphs

A very natural way of modeling object-oriented systems through graphs is by representing objects (or classes) as nodes and attributes and messages as arrows. However, the usual definition of graphs is altered in this work to deal with the object-oriented aspects of program specification. This modification is meant to reflect more precisely the underlying structure of the object-oriented paradigm, and so improve the compactness and understandability of specifications, as explained next.

Object-oriented systems consist of instances of previously defined classes which have an internal structure defined by attributes and communicate among themselves solely through message passing. That approach underlies the structure of the graphs used to model those systems. Each graph node is a class identifier, hyperarcs departing from it correspond to its internal attributes, and hyperarcs targeting at it consist of the services it provides to the exterior (i.e., its methods). Notice that the restrictions put to the structure of the hyperarcs assure, as expected, that messages target and attributes belong to a single object.

The inheritance hierarchy is also portrayed, by imposing a strict relation (Definition 2.1) among the graph nodes. Hyperarcs also possess an order structure, which reflects the possibility of a derived object to redefine the methods inherited from its ancestors. This feature will be used in Section 3.3 to define a formal semantics for dynamic binding based on graph computations.

Such structure is called a *class-model graph* and its formal definition is given below.

Definition 2.4 (Class-model graph) *A class-model graph is a tuple $\langle V_{\sqsubseteq}, E_{\sqsubseteq}, L, src, tar, lab\rangle$ where $V_{\sqsubseteq} = \langle V, \sqsubseteq_V^*\rangle$ is a strict ordered set of vertices, $E_{\sqsubseteq} = \langle E, \sqsubseteq_E^*\rangle$ is a strict ordered set of (hyper)edges, $L = \{\mathrm{attr}, \mathrm{msg}\}$ is an unordered set of two edge labels, $src, tar : E \to V^*$ are monotonic order-preserving functions, called respectively* source *and* target *functions, $lab : E \to L$ is the edge* labeling *function, such that the following constraints hold:*

Structural constraints: [2]

for all $e \in E$, the following holds:

- *if $lab(e) = \mathrm{attr}$ then $src(e) \in V$ and $tar(e) \in V^*$, and*
- *if $lab(e) = \mathrm{msg}$ then $src(e) \in V^*$ and $tar(e) \in V$.*

Sets $\{e \in E \mid lab(e) = \mathrm{attr}\}$ and $\{e \in E \mid lab(e) = \mathrm{msg}\}$ are denoted by $E|_{\mathrm{attr}}$ and $E|_{\mathrm{msg}}$, respectively.

[2]In the following, a little abuse of notation is used: for any set S, S^* denotes its reflexive and transitive closure with respect to concatenation (i.e., all finite lists formed by elements of S). In what follows, an element $s \in S^*$ means a finite list with (possibly repeated) elements of S, while $s \in S$ means either an element of S or a list containing a single element of S. This abuse is used in several texts on formal language theory (such as (LEWIS; PAPADIMITRIOU, 1998), (MARTIN, 1996), and (HOPCROFT; MOTWANI; ULLMAN, 2001)).

Order relations constraints: *for all $e \in E$, the following holds:*

1. *if $(e, e') \in \sqsubseteq_E$ then $lab(e) = lab(e') = \mathrm{msg}$,*
2. *if $(e, e') \in \sqsubseteq_E$ then $src(e) = src(e')$,*
3. *if $(e, e') \in \sqsubseteq_E$ then $(tar(e), tar(e')) \in \sqsubseteq_V^+$, and*
4. *if $(e', e) \in \sqsubseteq_E$ and $(e'', e) \in \sqsubseteq_E$, with $e' \neq e''$, then $(tar(e'), tar(e'')) \notin \sqsubseteq_V^*$ and $(tar(e''), tar(e')) \notin \sqsubseteq_V^*$.*

Class-model graphs are the graph structures used to model object-oriented class organizations. Each graph node is a class identifier, hyperarcs departing from it correspond to its internal attributes, and messages addressed to it are the services it provides to the exterior (i.e., its methods). Notice that the restrictions put to the structure of the hyperarcs assure, as expected, that messages target and attributes belong to a single object.

Inheritance and overriding hierarchies are made explicit by imposing that graph nodes (i.e., the objects) and message edges (i.e., the methods) are strict ordered sets (Definition 2.2). Notice that only single inheritance is allowed, since $\sqsubseteq_V$ is required to be a function. The relation between message arcs, $\sqsubseteq_E$, establishes which methods are overridden within the derived object, by mapping them. The restrictions applied to $\sqsubseteq_E$ ensure that methods are redefined consistently, i.e., only message arcs can be mapped (1), their parameters are the same (2), the method being redefined is located somewhere (strictly) above in the class-model graph (under $\sqsubseteq_V^+$) (3), and only the closest message with respect to relations $\sqsubseteq_V$ and $\sqsubseteq_E$ can be redefined (4).

The requirement concerning the absence of cycles and reflexive pairs on $\sqsubseteq_V$ and $\sqsubseteq_E$ is consistent, respectively, with the creation of classes and method redefinition within the object-oriented paradigm. A class is defined as a specialization of at most one other class (single inheritance), which must exist prior to the creation of the new class. A method can only redefine another method, which must always exist in a prior primitive class. Hence, neither a class can ever be created nor a method can be redefined in a circular or reflexive way.

Example 2.2 (Class-model graph) *Figure 2.3 presents a (naive) class-model graph for geometric shapes and figures. The nodes in the graph denote objects (shape, round, circle, ellipse, Figure, Drawing, Color and Integer), while object attributes and messages are represented by hyperarcs. The inheritance relation (actually, its underlying strict relation) is represented by dotted arrows and the redefinition function is represented by solid thin ones.*

Since class-model graphs are algebraic structures, morphisms between them can be defined. A class-model graph morphism formalizes the relationship between elements used by two different applications.

Definition 2.5 (Class-model graph morphism) *Given two class-model graphs, $\mathcal{C}_1 = \langle V_{1\sqsubseteq}, E_{1\sqsubseteq}, L, src_1, tar_1, lab_1 \rangle$ and $\mathcal{C}_2 = \langle V_{2\sqsubseteq}, E_{2\sqsubseteq}, L, src_2, tar_2, lab_2 \rangle$, with $L = \{\mathrm{attr}, \mathrm{msg}\}$, the tuple of order-preserving monotonic functions $t = \langle t_V : V_1 \to V_2, t_E : E_1 \to E_2, id_L : L \to L \rangle : \mathcal{C}_1 \to \mathcal{C}_2$ is a* class-model graph morphism *if and only if t_V and t_E are strict ordered functions and t is a partial labeled hypergraph morphism (Definition D.11).*

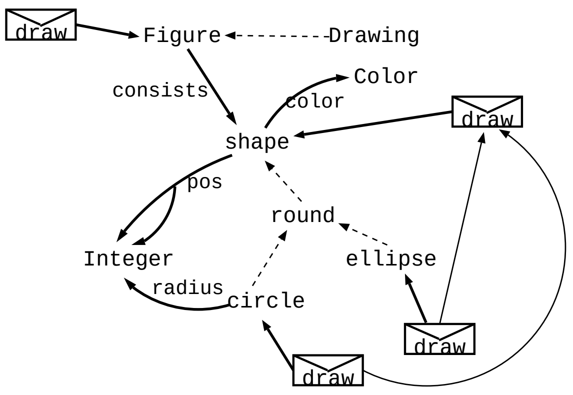

Figure 2.3: Class-model graph for geometric figures

A class-model graph morphism is a restricted partial labeled hypergraph morphism. The restrictions are related to the mapping coherence regarding the order relations on nodes and edges. Specifically, if a vertex (or arc) is mapped, then *all* elements from the chain to which it belongs regarding to the inheritance (or redefinition) relation in the first graph must also be mapped accordingly. This restriction is required to assure that single inheritance is maintained when the structures are combined. Before showing how this combinations can be performed, however, some properties of class-model graphs morphisms will be shown, together with the proof that class-model graphs and their morphisms constitute a category.

Lemma 2.2 *Class-model graph morphisms are closed under composition.*

Proof: Let $L = \{\text{attr}, \text{msg}\}$, $f = \langle f_V, f_E, id_L \rangle : \mathcal{C}_1 \to \mathcal{C}_2$ and $g = \langle g_V, g_E, id_L \rangle : \mathcal{C}_2 \to \mathcal{C}_3$ be two class-model graph morphisms. Composition of f and g is done componentwise, i.e., $g \circ f = \langle g_V \circ f_V, g_E \circ f_E, id_L \circ id_L \rangle$. Since strict ordered functions are closed under composition (Lemma 2.1), then the morphism component functions $g_V \circ f_V$ and $g_E \circ f_E$ are strict ordered functions. Additionally, partial labeled hypergraph morphisms are also closed under composition, which proves that $g \circ f$ is a class-model graph morphism. □

Lemma 2.3 *Composition of class-model graphs morphisms is associative.*

Proof: Again, composition of class-model graphs morphisms is done componentwise. Since strict ordered sets and strict ordered functions constitute a category (Theorem 2.1), composition of strict ordered functions is associative. Category **LabHGraphP** (Definition D.19) assures that composition of partial labeled hypergraph morphisms is also associative. □

Theorem 2.5 (Category CGraph) *There is a category* **CGraph** *which has class-model graphs as objects and class-model graph morphisms as arrows.*

Proof: Lemma 2.2 shows that class-model graph morphisms are closed under composition, whereas Lemma 2.3 proves the associativity property for class-model graph morphisms. Let $\mathcal{C} = \langle V_{\sqsubseteq}, E_{\sqsubseteq}, L, src, tar, lab \rangle$ be a class-model graph and $L = \{\text{attr}, \text{msg}\}$. The trivial morphism $id_{\mathcal{C}} = \langle id_V, id_E, id_L \rangle : \mathcal{C} \to \mathcal{C}$ is the identity morphism, i.e., $id_V(v) = v$ for each $v \in V$, $id_V(e) = e$ for each $e \in E$ and $id_L(l) = l$ for each $l \in L$. Notice that, besides being a partial labeled hypergraph morphism, the identity function is also a strict ordered function

(Lemma 2.2). Hence, for any class-model graph morphism $f = \langle f_V, f_E, id_L\rangle : \mathcal{C}_1 \to \mathcal{C}_2$, with $\mathcal{C}_1 = \langle V_{1\sqsubseteq}, E_{1\sqsubseteq}, L, src_1, tar_1, lab_1\rangle$ and $\mathcal{C}_2 = \langle V_{2\sqsubseteq}, E_{2\sqsubseteq}, L, src_2, tar_2, lab_2\rangle$, $f \circ id_{\mathcal{C}_1} = \langle f_V \circ id_{V_1}, f_E \circ id_{E_1}, id_L \circ id_L\rangle$ we have that for each $v \in V_1$, $(f_V \circ id_{V_1})(v) = f_V(id_{V_1}(v)) = f_V(v) = id_{V_2}(f_V(v)) = (id_{V_2} \circ f_V)(v)$ holds; for each $e \in E_1$, $(f_E \circ id_{E_1})(e) = f_E(id_{E_1}(e)) = f_E(e) = id_{E_2}(f_E(e)) = (id_{E_2} \circ f_E)(e)$ holds; and $id_L \circ id_L = id_L$.

The existence of a identity morphism for each object, the existence of an arrow composition operation and the associativity of composition prove that **CGraph** is a category.

□

Characterizing objects, attributes and methods this way creates a situation where graphs are no longer defined over sets and functions, but over strict ordered sets and strict ordered functions. The very abstract way that graphs and graph morphisms are dealt with within category theory can be maintained by moving from diagrams from the category **SetP** of sets and partial functions, used by the single-pushout approach to graph transformation, to the category **SOSet** of strict ordered sets and strict ordered functions.

The algebraic approach to graph grammars rely on categorical constructs to express most of its results. Having graphs and graph morphisms expressed in any category is useful, in the sense that if the needed categorical constructs used within the theory of graph grammars can be proven to exist in the new setting, the conclusions drawn from the former could be automatically transferred to the latter.

Again, although **CGraph** has a initial object (Theorem 2.6) and arbitrary coproducts (Theorem 2.7), it is not cocomplete, because it does not have coequalizers for each pair of arrows. However, special colimits can be proven to exist. Those colimits, similar to the ones shown in Theorem 2.4, are the ones we are interested in for the purposes of this work, so we conclude this section with the following results:

Theorem 2.6 (Initial object in CGraph) *Category* **CGraph** *has an initial object.*

Proof: On page 126. □

Theorem 2.7 (Coproducts in CGraph) *Category* **CGraph** *has coproducts.*

Proof: On page 127. □

Theorem 2.8 (Special colimits in CGraph) *Let* **D** *be a diagram in* **CGraph** *containing the objects class-model graphs $\mathcal{C}_1 = \langle V_{1\sqsubseteq}, E_{1\sqsubseteq}, L, src_1, tar_1, lab_1\rangle$ and $\mathcal{C}_2 = \langle V_{2\sqsubseteq}, E_{2\sqsubseteq}, L, src_2, tar_2, lab_2\rangle$, and morphism $t = \langle t_V, t_E, id_L\rangle : \mathcal{C}_1 \to \mathcal{C}_2$. Then* **D** *has a colimit, represented by object $\mathcal{C}_C$ and morphisms $c_i : \mathcal{C}_i \to \mathcal{C}_C$ for $i = 1, 2$.*

Proof:

Given two class-model graphs $\mathcal{C}_1 = \langle V_{1\sqsubseteq}, E_{1\sqsubseteq}, L, src_1, tar_1, lab_1\rangle$ and $\mathcal{C}_2 = \langle V_{2\sqsubseteq}, E_{2\sqsubseteq}, L, src_2, tar_2, lab_2\rangle$ together with a class-model graph morphism $t = \langle t_V, t_E, id_L\rangle : \mathcal{C}_1 \to \mathcal{C}_2$, let $\langle V_{C\sqsubseteq}, E_{C\sqsubseteq}, L, src_C, tar_C, lab_C\rangle$ be the structure generated as follows:

- V_C is the colimit object of the colimit $\langle\langle V_C, \sqsubseteq_{V_C}\rangle, v_1 : V_1 \to V_C, v_2 : V_2 \to V_C\rangle$ in the category **SOSet** of the diagram containing objects $V_{1\sqsubseteq} = \langle V_1, \sqsubseteq_{V_1}\rangle$, $V_{2\sqsubseteq} = \langle V_2, \sqsubseteq_{V_2}\rangle$, and arrow $t_V : V_{1\sqsubseteq} \to V_{2\sqsubseteq}$ (according to Theorem 2.4);

- E_C is the colimit object of the colimit $\langle\langle E_C, \sqsubseteq_{E_C}\rangle, e_1 : E_1 \to E_C, e_2 : E_2 \to E_C\rangle$ in the category **SOSet** of the diagram containing objects $E_{1\sqsubseteq} = \langle E_1, \sqsubseteq_{E_1}\rangle$, $E_{2\sqsubseteq} = \langle E_2, \sqsubseteq_{E_2}\rangle$, and arrow $t_E : E_{1\sqsubseteq} \to E_{2\sqsubseteq}$ (according to Theorem 2.4);
- functions $src_C, tar_C : E_C \to V_C$ are the ones induced by the construction in such a way that the diagram (in category **SOSet**)

$$\begin{tikzcd}
& L \arrow[d, leftrightarrow, "id_L"] \arrow[ddl, bend right, "id_L"'] & & E_1 \arrow[ll, "lab_1"'] \arrow[rr, "{src_1,tar_1}"] \arrow[d, "t_E"] \arrow[ddl, bend right, "e_1"] & & V_1 \arrow[d, "t_V"] \arrow[ddl, bend right, "v_1"] \\
& L \arrow[dl, "id_L"] & & E_2 \arrow[ll, "lab_2"'] \arrow[rr, "{src_2,tar_2}"] \arrow[dl, "e_2"] & & V_2 \arrow[dl, "v_2"] \\
L & & E_C \arrow[ll, "lab_C"'] \arrow[rr, "{src_C,tar_C}"] & & V_C &
\end{tikzcd}$$

  commutes (i.e., $src_C(e) = v_1 \dots v_n$ if and only if $src_i(e') = v'_1 \dots v'_n$ with $e_i(e') = e$ and $v_i^*(v'_1 \dots v'_n) = v_1 \dots v_n$, $lab_C(e) = lab_i(e')$ for some $e' \in E_i$, $v'_1 \dots v'_n \in V_i$, and $tar_C(e) = v$ if and only if $tar_i(e') = v'$ with $e_i(e') = e$ and $v_i(v') = e$ for some $e' \in E_i$, $v' \in V_i$, with $i \in \{1, 2\}$).

The tuple $\langle \mathcal{C}_C, c_1 = \langle v_1, e_1, id_L\rangle : \mathcal{C}_1 \to \mathcal{C}_C, c_2 = \langle v_2, e_2, id_L\rangle : \mathcal{C}_1 \to \mathcal{C}_C\rangle$, is a colimit of the diagram containing $\mathcal{C}_1$, $\mathcal{C}_2$, and $t = \langle t_V, t_E, id_L\rangle : \mathcal{C}_1 \to \mathcal{C}_2$.

In order to prove that the proposed structure is actually a colimit of the diagram formed by class-model graphs $\mathcal{C}_1$ and $\mathcal{C}_2$, along with the class-model graph morphism t, we must prove 3 (three) things: (i) $\mathcal{C}_C$ is indeed a class-model graph, (ii) $c_1 : \mathcal{C}_1 \to \mathcal{C}_C$ and $c_2 : \mathcal{C}_1 \to \mathcal{C}_C$ are class-model graph morphisms, and (iii) for any other class-model graphs $\mathcal{C}_D$ and class-model graphs morphisms $d_1 : \mathcal{C}_1 \to \mathcal{C}_C$ and $d_2 : \mathcal{C}_1 \to \mathcal{C}_C$ such that $d_1 = d_2 \circ t$ there is a unique class-model graph morphism $h : \mathcal{C}_C \to \mathcal{C}_D$ such that $d_1 = h \circ c_1$ and $d_2 = h \circ c_2$.

(i) We begin noticing that t is a labeled hypergraph morphism (Definition 2.5), and therefore it preserves edges labels, sources and targets. Therefore the first all structural constraints and the three first order relation constraints (Definition 2.4) are fulfilled. The last one is met by noticing that t_E is a strict ordered function, which means that all elements belonging to the domain of t_E must have their upper set elements mapped to. Similarly to the proof of Theorem 2.4, the desired structure is maintained.

(ii) in order to prove that $c_1 = \langle v_1, e_1, id_L\rangle : \mathcal{C}_1 \to \mathcal{C}_C$ and $c_2 = \langle v_2, e_2, id_L\rangle : \mathcal{C}_1 \to \mathcal{C}_C$ are class-model graph morphisms, we have to show that they are labeled hypergraph morphisms, because v_1, v_2, e_1 and e_2 are (by construction) strict ordered functions. So, it remains to prove that $c_{1V} \circ src_1 = src_C \circ c_{1E}$ and $c_{1V} \circ tar_1 = tar_C \circ c_{1E}$, which is true by construction.

(iii) let $\mathcal{C}_D = \langle V_{D\sqsubseteq}$, $E_{D\sqsubseteq}$, L, src_D, tar_D, $lab_D\rangle$ be a class-model graph, $d_1 : \mathcal{C}_1 \to \mathcal{C}_D$ and $d_2 : \mathcal{C}_1 \to \mathcal{C}_D$ be two class-model graph morphisms such that $d_1 = d_2 \circ t$. Let $h : \mathcal{C}_C \to \mathcal{C}_D$ be a class-model graph morphism constructed as follows: for each $x \in dom(d_i)$, let $h(c_i(x)) = d_i(x)$; for all $x \notin dom(d_i)$, let $h(c_i(x))$ be undefined. For all $x \in \mathcal{C}_i$, if $x \in dom(d_i)$ then $\uparrow x \subseteq dom(d_i)$ (because d_i is a class-model graph morphism). So $c_i(\uparrow x) \subseteq dom(h)$ (by definition of h). Since both d_1 and d_2 are class-model graph morphisms, we have that, for all $x \in dom(d_1)$, $y \in dom(d_2)$, $d_1(\uparrow x) = \uparrow d_1(x) \cap \downarrow d_1(\sqcup\uparrow x)$ and $d_2(\uparrow y) = \uparrow d_2(y) \cap \downarrow d_2(\sqcup\uparrow y)$. Therefore, for all $x \in dom(c_i)$ we have that $(h \circ c_i)(\uparrow x) = \uparrow(h \circ c_i)(x) \cap \downarrow(h \circ c_i)(\sqcup\uparrow x)$, or that $h(c_i(\uparrow x)) = \uparrow h(c_i(x)) \cap \downarrow h(\sqcup\uparrow c_i(x))$. Hence, h is a strict ordered function. Now, suppose there is another strict ordered function $h' : \mathcal{C}_C \to \mathcal{C}_D$ such that $d_i = h' \circ c_i$ but $h \neq h'$. Then, there must be an element $e \in \mathcal{C}_C$ such that

$h(e) \neq h'(e)$. But for all $x \in dom(d_i)$, $d_i(x) = h'(c_i(x)) = h(c_i(x))$; and for all $x \notin dom(d_i)$, $d_i(x) = h'(c_i(x)) = h(c_i(x)) = undef$, and therefore $h = h'$. $\square$

The existence of certain kinds of colimits is a desirable feature of categories having objects intended to represent specifications: it means that specifications can be combined or merged in a meaningful and correct way. Moreover, since colimits are unique up to isomorphism, the resulting composite specification is always well defined. In the next section we will explore the use of those colimits in the category **CGraph** to formally define system composition and extension.

2.4 System extension

Software systems are generally built from previously constructed subsystems, which are later combined. The object-oriented paradigm favors that approach: existent objects can be aggregated or derived to form new ones. Composition is one of the most fundamental operations over systems, and it must be formalized in such a way that its result is compatible with the way systems are in fact combined. So, modularity plays a key role in software development, allowing a complete specification to be constructed from different, smaller ones. The need for integration tools also is a key issue in software development (REPS, 1991), since that task is considerably demanding in terms of effort if it is not automatized. Hence, specification formalisms should allow composition which can be performed systematically, guaranteeing a meaning for the operations in terms of system composition. Composition of class-model graphs is described next.

Definition 2.6 (Binary class-model graph composition) *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be class-model graphs, and $t : \mathcal{C}_1 \to \mathcal{C}_2$ be a class-model graph morphism. The* composition *of $\mathcal{C}_1$ and $\mathcal{C}_2$ under f is the colimit object (within category* **CGraph***) of the diagram containing $\mathcal{C}_1$, $\mathcal{C}_2$, and f.*

The class-model graph morphism f from Definition 2.6 represents the mapping between elements (objects, attributes, or methods) which are considered to be *the same* in two different subsystems. This interpretation makes sense when we think about the specification of object-oriented systems: it is fairly common, when programming a class, to make use of objects defined elsewhere. It is not required to have knowledge of the complete specification of a class to use it as an attribute or to call on some of its methods. However, when the object files are linked together, the whole system must be completely specified. The morphisms used to perform composition of specifications play the role of identifying which elements are shared by different subsystems (the mapped elements) and which ones belong to just one of them (the ones neither in the domain of definition nor in the image of the morphism used to relate sub-specifications).

Example 2.3 (Class-model graph composition) *Consider the class-model graph presented in Figure 2.3: it shows a set of geometric figures having a procedure to draw each of them. Figure 2.4 presents a very similar class-model graph for geometric shapes, being the only difference a parameter* Environment *for the drawing messages.*

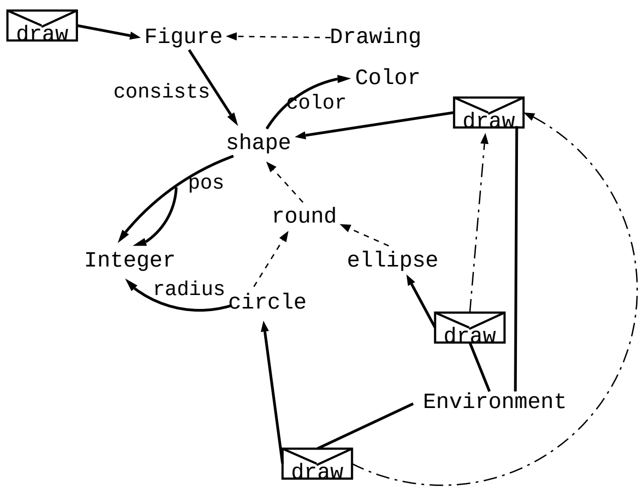

Figure 2.4: Class-model graph for geometric shapes

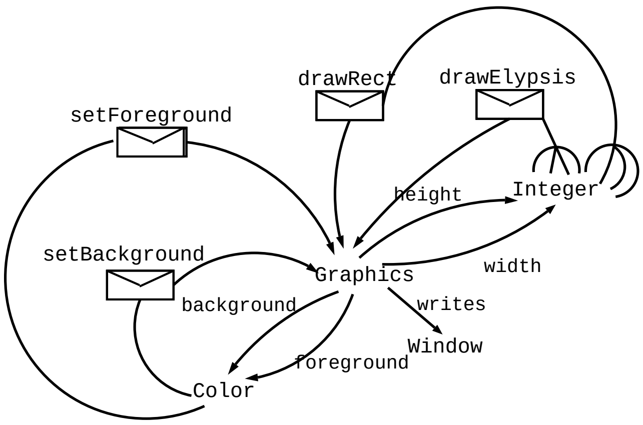

Figure 2.5: Class-model graph for graphic primitives

Drawing is commonly performed by some graphic primitives belonging to some function library, mainly because graphic functions are hardware dependent. For that reason, programs making use of graphic primitives usually call them through imported functions, whose object code are linked together in the final step of the compilation process. Figure 2.5 shows a class-model graph composed by four objects: Integer, Color, Window *and* Graphics. *The last one has five attributes (*foreground *and* background*, of type* Color*;* width *and* height*, of type* Integer*; and* writes*, of type* Window*) and receives four messages (*setForeground *and* setBackground*, which have both one parameter of type* Color*; and* drawRect *and* drawEllipse*, which have four Integer parameters each).*

This class-model graph can be seen as the definition of a class Graphics*, with all its attributes and method signatures, which can be later combined to other class-model graphs to form a complete specification description. This merge of class-model graphs can be achieved by a colimit operation, as described in Definition 2.6. Notice that it requires a set of class-model graphs and a set of class-model graph morphisms, to connect elements in different class-model graphs which are meant to be* the same *(even if their* names *are actually different). The colimit object is built by collapsing the mapped elements, and by leaving all the other elements unaltered.*

The morphism mapping the common elements within the class-model graphs

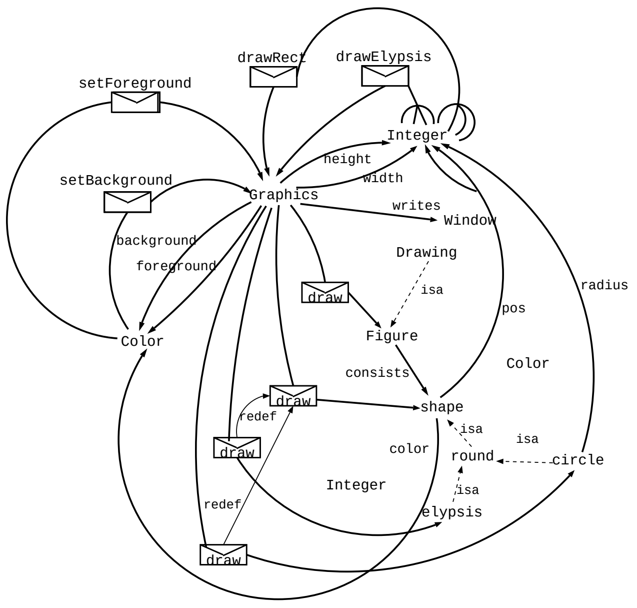

Figure 2.6: Composition of specifications as class-model graph composition

shown in Figures 2.4 and 2.5, is as follows:

Class-model graph (Figure 2.4)	*Class-model graph (Figure 2.5)*
Environment	*Graphics*
Integer	*Integer*
Color	*Color*

Notice that only nodes are mapped, although arcs could have been mapped too. Figure 2.6 shows the resulting system. Notice how the internal structure of the class Window*, for instance, is not shown in the picture. This class can be defined in yet another class-model graph, which can be combined in the same way the class-model graphs presented in this example.*

Composition of class-model graphs can also be used to perform different tasks other than plain system composition. Namely, specialization through inheritance and object aggregation, which are the most common way of extending a specification, can be understood in terms of class-model graphs composition. This is important for it generates an uniform treatment on the way systems are combined and augmented. So, all results applied to system composition (as colimits in the category **SOSet**) are also applied to object creation using inheritance or aggregation. Example 2.4 shows how specialization through inheritance can be achieved by class-model graphs composition. Object construction by aggregation can be defined in a similar manner, as explained in Example 2.5.

Example 2.4 (Specialization through inheritance) *The most common way of code reuse in the object-oriented paradigm is done through inheritance. This operation can be formalized by class-model graph composition (notice that the creation of a new object always alters a system specification, so it is coherent to formalize it by composition). To do so, it is necessary to create the specification of the new*

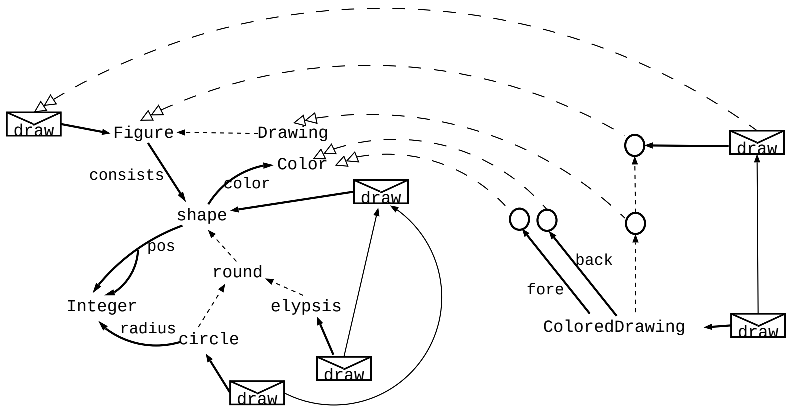

Figure 2.7: Specialization through inheritance as class-model graph composition

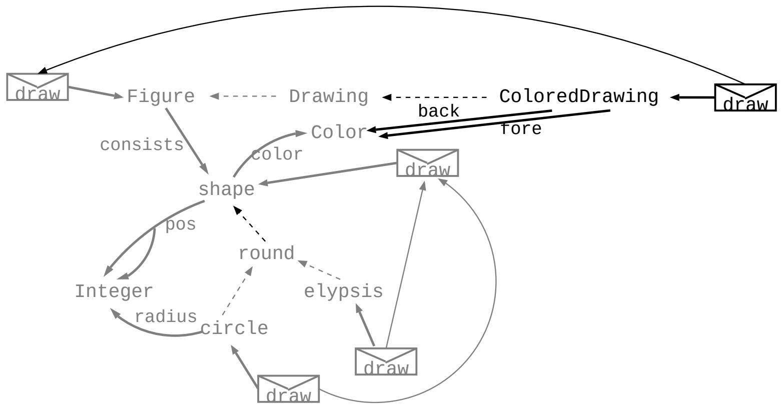

Figure 2.8: Specialization through inheritance as class-model graph composition

object (all attributes and messages included) and connect it to a chain of nodes as long as needed. How many are "as needed" will be made clear in what follows.

For instance, suppose we want to specialize an object of type Drawing *from the class-model graph portrayed in Figure 2.3 to add to it a background and a foreground color. Besides these attributes, it should also redefine the method* Draw, *which was initially defined at the level of object* Figure. *It is necessary to portray a node to be related to the Figure node at the original class-model graph, in order to map the message which is being overridden in the morphism. However, the mapping must be a class-model graph morphism, which means that the mapping on nodes need to be a strict ordered function. Hence,* all *nodes between the new class node and the highest node having a message to be overridden have to be made explicit. The resulting class-model graph, along with the class-model graph morphism (represented as dashed double arrows) which relates the corresponding elements on both class-model graphs is shown in Figure 2.7.*

The final composite system is shown in Figure 2.8. Notice how the whole system structure was maintained, with the exception of the new added class ColoredDrawing *which is derived from the class* Drawing, *as intended.*

The class-model graph built to perform specialization through inheritance must possess at least one node to which the new element must be connected via the order relation (i.e., at least one primitive class). This particular node must be connected to the primitive object on the other class-model graph. The number of such objects on the constructed hierarchy depends on the methods the derived object is intended to redefine. There must be as many objects on the node chain as there are elements between the primitive object and the one to which the method to be redefined belongs.

The process described in Example 2.4 can be formally stated as follows.

Definition 2.7 (Inheritance as class-model graph composition) *Let $\mathcal{C} = \langle V_{\sqsubseteq}, E_{\sqsubseteq}, L, src, tar, lab\rangle$ be a class-model graph, $p \in V$ the object one wants to derive and $M = \{m_1, \ldots, m_n\}$, $n \geqslant 0$, a collection of messages to be redefined by the newly created class. Let $u = \sqcup\{tar(m) \mid m \in M\}$ if $M \neq \emptyset$ and $u = p$ otherwise. Now let U be a strict ordered set isomorphic to $\uparrow p \cap \downarrow u$ (with isomorphism ι_U).*

Let $\mathcal{C}_O = \langle V_{O\sqsubseteq}, E_{O\sqsubseteq}, L, src_O, tar_O, lab_O\rangle$ be a class-model graph with the following characteristics:

- *$V_O = U \cup \{o\} \cup A_o \cup A_m \cup A_r$, where o is called the* object-vertex*, $A_o = \{{t^a}_1, \ldots, {t^a}_k\}$ its set of attributes types, A_m is the set of the new messages parameter's types, A_r is the set of the to be redefined messages parameter's types which is isomorphic (with isomorphism ι_V) to the set $\{src(m) \mid m \in M\}$; $\sqsubseteq_V^* = \sqsubseteq_U^* \cup \{(o,u) \mid u \in U\}$;*
- *$E_O = M_N \cup M' \cup M_R \cup \overline{K}_a$, where M' is a set isomorphic to M (with isomorphism ι'_E), M_R is also isomorphic to M (with isomorphism ι_E) and contains the messages that will be redefined within the new object, M_N is a set of hyperarcs representing the methods belonging to the new object o, and $\overline{K}_a = \{1_a, 2_a, \ldots, k_a\}$ contains the new object attribute arcs; $lab_O(x_a) = \mathrm{attr}$, $src_O(x_a) = o$ and $tar_O(x_a) = t^a_x \in A_o$ for all $x_a \in \overline{K}_a$; $lab_O(x) = \mathrm{msg}$, $src_O(x) \subseteq A_m^*$ and $tar_O(x) = o$ for all $x \in M_N$; $lab_O(x) = \mathrm{msg}$, $src_O(x) \subseteq A_r^*$ and $tar_O(x) = o$ for all $x \in M_R$, with $(\iota_V \circ src_O)(x) = (src \circ i_E)(x)$; $\sqsubseteq_E = \{(m,m') \mid m \in M_R \wedge m' \in M' \wedge \iota_E(m) = \iota'_E(m')\}$, relating the messages which redefine and which are redefined within the new object;*

Now, let t be a class-model graph morphism defined as follows:

- *$dom(t_V) = V \setminus \{o\}$, where o is the object-vertex of O; $t_V(u) = \iota_U(u)$, for all $u \in U$, $t_V(a) = \iota_V(a)$, for all $a \in A_r$, and for all vertices $v \in (A_o \cup A_m)$, $t_V(v)$ is the mapping which connects the vertex v to its actual type in the existing class-model graph $\mathcal{C}$;*
- *$dom(t_E) = M'$ which coincides with ι'_E, i.e., $t_E(m) = \iota'_E(m)$ for all $m \in M'$.*

The object of the colimit given by the diagram containing objects $\mathcal{C}$ and $\mathcal{C}_O$, and morphism $t = \langle t_V, t_E, id_L\rangle$ as described above, correspond to the object-oriented system specification defined by $\mathcal{C}$ augmented with one object o which was formed by the derivation (via inheritance) of the existing object p in the class-model graph $\mathcal{C}$.

Example 2.5 (Aggregation) *Aggregation is the operation used to combine two or more existent objects to construct a new one, allowing the new object to use all their constituents functionalities in a transparent way. Given a class-model graph which contains the objects we want to aggregate, it is easy to augment it using the composition operation, in such a way that the resulting class-model graph contains*

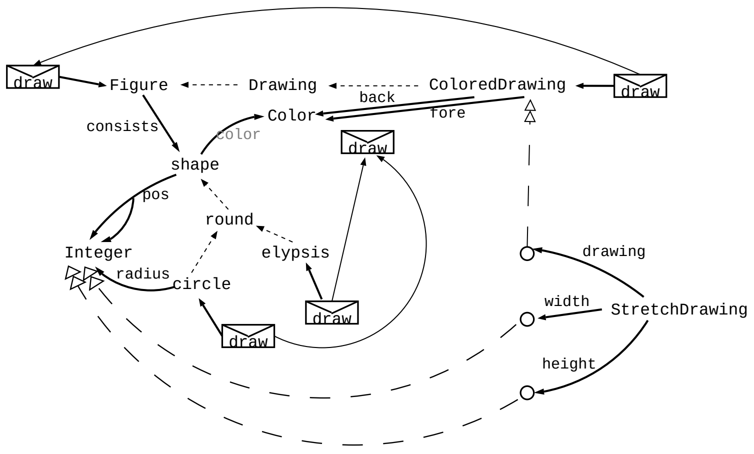

Figure 2.9: Aggregation as class-model graph composition

the new object. Notice that, if we want to aggregate classes belonging to more than one class-model graph, it is enough to combine them first, using empty morphisms between them, and then apply the aggregation procedure to the resulting class-model graph. Class-model graph composition was defined as a binary colimit, which is an associative (up to isomorphism) operation. Additionally, a colimit of two class-model graphs under an empty morphism is equivalent (up to isomorphism) to a coproduct operation.

For example, suppose we want an object called StretchDrawing, *which embraces the functionality of the object* ColoredDrawing *(specialized from* Drawing *in the Example 2.4), plus two attributes of type* Integer, *called* width *and* height. *The class-model graph that should be constructed to aggregate existing objects belonging to another class-model graph has as constituents a node together with its own attributes and messages, and which is additionally connected to as many nodes as the classes one wants to aggregate. The morphisms should then identify the latter ones with the actual classes to be aggregated. Figure 2.9 presents this new class-model graph, together with the morphism represented by dashed lines with double hollow triangles at their ends.*

Figure 2.10 portrays the resulting class-model graph (generated as the colimit of the diagram presented in Figure 2.9), which contains the new node with the required attributes.

Definition 2.8 (Aggregation as class-model graph composition) *Let $\mathcal{C} = \langle V_{\sqsubseteq}, E_{\sqsubseteq}, L, src, tar, lab\rangle$ be an arbitrary class-model graph and $\mathcal{C}_O = \langle V_{O\sqsubseteq}, E_{O\sqsubseteq}, L, src_O, tar_O, lab_O\rangle$ be a class-model graph where*

- *$V_O = \{o\} \cup A_o \cup A_m$ where o is the new object to be created, $A_o = \{t^a{}_1, \ldots, t^a{}_n\}$ its set of attributes types, $A_m = \{t^m{}_1, \ldots, t^m{}_l\}$ is the set of message parameter's types; $\sqsubseteq_{V_O} = \emptyset$;*
- *$E_O = \overline{N}_a \cup M_o$, where $\overline{N}_a = \{1_a, 2_a, \ldots, n_a\}$ and $M_o = \{m_1, \ldots, m_k\}$ is a set of k hyperarcs representing the methods belonging to the new object o; $lab_O(x_a) = \mathrm{attr}$, $src_O(x_a) = o$ and $tar_O(x_a) = t^a_x \in A_o$ for all $x_a \in \overline{N}_a$; $lab_O(x) = \mathrm{msg}$, $src_O(x) \subseteq A^*_m$ and $tar_O(x) = o$ for all $x \in M_o$; $\sqsubseteq_{E_O} = \emptyset$.*

Let $t : \mathcal{C}_O \to \mathcal{C}$ be the following class-model graph morphism: $dom(t_V) = V_O \setminus \{o\}$,

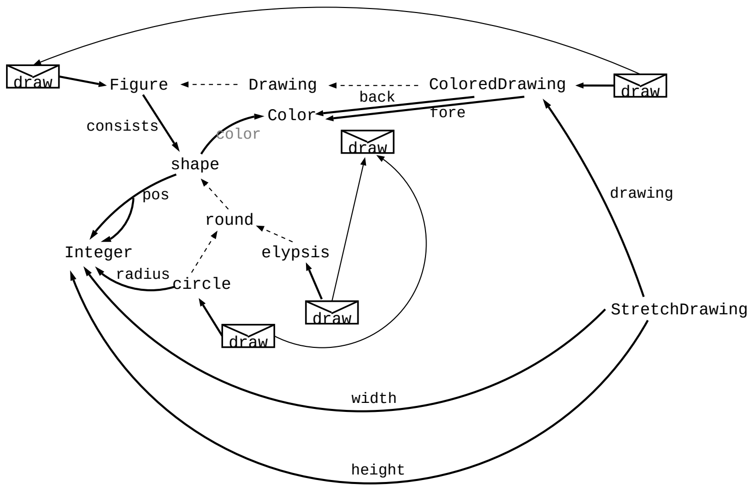

Figure 2.10: Aggregation as class-model graph composition

and for all $x \in (A_o \cup A_m)$, $t_V(x) = v$ for some $v \in V$ which reflects the actual type of the attribute x into $\mathcal{C}$; $t_E = \emptyset$.

The object of the colimit given by the diagram containing objects $\mathcal{C}$ and $\mathcal{C}_O$, and morphism t as described above, correspond to the object-oriented system specification defined by $\mathcal{C}$ augmented with one object o which was formed by aggregation of existing objects in $\mathcal{C}$.

2.5 Summary

This chapter begins by relating the most fundamental characteristics of the object-oriented programming paradigm with partial orders. Namely, inheritance, polymorphism and method overriding are shown to have a nature which resembles the structure of partial orders.

Strict relations are defined as the base "skeleton" relation which models the hierarchy of inheritance and method overriding for an object-oriented specification. It is shown that the reflexive and transitive closure of such relation is a partial order (i.e., a reflexive, transitive, and antisymmetric relation). Next, strict ordered sets and strict ordered functions are defined, and it is shown that they form a category.

The most important structure of this chapter is the *class-model graph*, which is a labeled hypergraph with a restricted edge structure, whose sets of nodes and hyperedges are strict ordered sets. The underlying relations of such sets obey additional restrictions, intended to assure that class-model graphs provide an adequate and faithful model of how object-oriented classes are actually organized. A class-model graph reflects the structure of classes (nodes), with their attributes and methods (hyperedges) in any object-oriented system.

Class-model graphs are algebraic structures, and so they can be related by the notion of morphism. A morphism between two class-model graphs is a labeled hypergraph morphism whose node and hyperedge mappings preserve their underlying strict relation, in such a way that, when combined, the two structures are "glued" together to form a new one having the desired structure.

A class-model graph morphism identifies elements from two distinct class-model graphs which are meant to be the *same*. Having established that information, one can compose two class-model graphs (i.e., object-oriented class specifications) by creating a third one, where the mapped elements are identified as being a single one. Composition of systems is then formally defined as the colimit object of the diagram formed by the systems being composed (with a suitable morphism connecting them). It assures that composition of systems is unique (up to isomorphism) and well defined.

Finally, it is shown that system extension through inheritance — achieved when a class is created by deriving some other already existing class, and aggregation — which occurs when a new class is composed as a collection of other classes, are both special cases of class-model graph composition. It means that the existing ways of augmenting an object-oriented system can be all formalized by the same categorical construction.

3 OBJECT-ORIENTED COMPUTATIONS

3.1 Introduction

Rule–based systems have proven to be useful for describing computations by local transformations: arithmetic, syntactic, and deduction rules are familiar examples. Language definition, semantics of programming languages, logic and functional programming, algebraic specification, term rewriting, theorem proving, expert systems, concurrent and mobile processes, are some examples of areas which witness the prominent role of rules in computer programming. Rule-based graph transformation also constitutes a quite natural way to combine graphs, for describing complex structures, with rules, to manipulate them. Graph transformation, also known as graph rewriting, combines the potentials and advantages of both, graphs and rules, into a single computational paradigm.

The theory of graph transformation systems, as previously mentioned in Chapter 1, studies a variety of formalisms which expand the theory of formal languages (HOPCROFT, 1969), (HOPCROFT; MOTWANI; ULLMAN, 2001), (LEWIS; PAPADIMITRIOU, 1998), (MARTIN, 1996), to encompass more general structures specified as graphs. All constructions known from the string transformation approach in the formal language framework also exist in graph transformations systems: rules, derivations and generated languages can be defined for graphs in the very same manner they are defined for strings. Only naturally, different types of graph rules give rise to different classes of graph languages, with different expressiveness and differences in the decidability of the associated problems. It has been shown that all enumerable graph sets can be generated using very restricted graph transformation rules (NAGL, 1986). Similarly to string grammars, graph grammars (graph transformation systems equipped with an initial graph) also provide a model of computation. The structure of graphs, graph productions and results of rule applications determine the model of computation provided.

A graph transformation system allows to describe finitely a collection (finite or infinite) of graphs, which can be obtained from an initial graph through the repeated application of graph productions.

This chapter is structured as follows: Section 3.2 presents $\mathcal{C}$-typed graphs and $\mathcal{C}$-typed graph morphisms, which are respectively hypergraphs typed over class-model graphs, introduced in Section 2.3, and morphisms between those structures. Object-oriented graphs, which intend to represent object-oriented systems, are special cases of $\mathcal{C}$-typed graphs, and are defined at the end of this section. Section 3.2 also contains the proof that there are categories $\mathbf{CGraphP}(\mathcal{C})$ and $\mathbf{OOGraphP}(\mathcal{C})$, having respectively, $\mathcal{C}$-typed graphs as objects and $\mathcal{C}$-typed graph morphisms as arrows, and object-oriented graphs as objects and $\mathcal{C}$-typed graph morphisms as arrows. The existence of such categories is essential to make the theory developed in this work adhere to the algebraic single-pushout approach theory of graph grammars. Section 3.3 defines object-oriented rules, matches and direct derivations. Object-oriented rules are constraint in order to implement the concepts of data and code *encapsulation*, *information hiding*, as well as to restrict their grammars computations to reflect the behaviour of object-oriented programs. The most important result of this chapter is provided within this section, which is the proof that an object-oriented direct derivation is a pushout in the category $\mathbf{OOGraphP}(\mathcal{C})$. This result makes applicable a number of results from the traditional single-pushout approach, such as sequential and parallel independence of derivations (EHRIG; ROSEN, 1980), (EHRIG; LÖWE, 1993), abstract and concrete derivations (CORRADINI et al., 1993), and a number of different approaches to graph grammar semantics (CORRADINI et al., 1994), (RIBEIRO, 1996), (HECKEL et al., 2001). This section ends with some properties regarding different types of object-oriented rules, and with the formal definition of an object-oriented graph grammar. Section 3.4 presents some definitions concerning object behaviour, and an observation semantics for object-oriented graph grammars is defined. Section 3.5 presents some final remarks.

3.2 Object-oriented graphs

Chapter 2 presented class-model graphs, which can adequately model object-oriented class signatures. The focus of this chapter lies on object-oriented *systems*, or *programs*, i.e., a collection of instances from the classes previously defined — called *objects* — which can interact with each other through *message passing* (the object-oriented jargon for method invocation or procedure call).

Classes are represented as nodes in a class-model graph, and their methods and attributes are represented as hyperedges. A collection of objects, with their attributes and addressed messages can be characterized as a graph *typed* over a class-model graph. Typed graphs ((CORRADINI; MONTANARI; ROSSI, 1996), and Definition D.12) present a powerful formalism to describe system specifications. However, typed graph morphisms are not suitable for the specification of object-oriented systems, since features such as inheritance and polymorphism cannot be dealt by them appropriately (FERREIRA; RIBEIRO, 2003). The typing we propose here can be considered more "flexible", in the sense that the typing morphism must preserve the hyperarcs sources and targets in a less restricted way. The definition of a $\mathcal{C}$-typed graph below shows how this flexibility is achieved and how it is related to object-oriented systems. Method implementation will be described in Section 3.3, where object-oriented *rules* are defined.

Notation: For any partially ordered set $\langle P, \sqsubseteq_P \rangle$, an induced partially ordered set $\langle P^*, \sqsubseteq_{P^*} \rangle$ can be constructed, such that for any two strings $u = u_1 \ldots u_n, v = v_1 \ldots v_n \in P^*$, we have that $u \sqsubseteq_{P^*} v$ if and only if $|u| = |v|$ and $u_i \sqsubseteq_P v_i$, $i = 1, \ldots, |u|$. If $P_\sqsubseteq$ and $Q_\sqsubseteq$ are partially ordered sets, any monotonic function $f : P \to Q$ can be extended to a monotonic function $f^* : P^* \to Q^*$, with $f^*(p_1 \ldots p_n) = f(p_1)f(p_2) \ldots f(p_n) = q_1 \ldots q_n$ where $p_1 \ldots p_n \in P^*$ and $q_1 \ldots q_n \in Q^*$.

Definition 3.1 ($\mathcal{C}$-typed graph) *A $\mathcal{C}$-typed graph $G^{\mathcal{C}}$ is a tuple $\langle G, t, \mathcal{C} \rangle$, where $\mathcal{C} = \langle V_{\mathcal{C}_\sqsubseteq}, E_{\mathcal{C}_\sqsubseteq}, L, src_{\mathcal{C}}, tar_{\mathcal{C}}, lab_{\mathcal{C}} \rangle$ is a class-model graph, $G = \langle V_G, E_G, src_G, tar_G \rangle$*

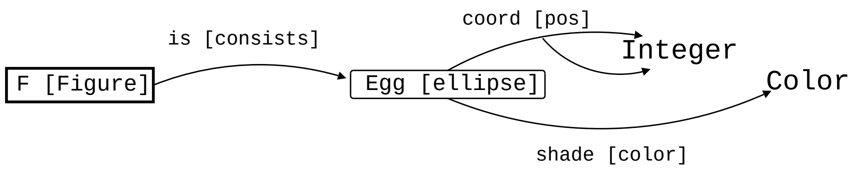

Figure 3.1: Example of a $\mathcal{C}$-typed graph

is a hypergraph, and t is a pair of total functions $\langle t_V : V_G \to V_{\mathcal{C}}, t_E : E_G \to E_{\mathcal{C}} \rangle$ *such that* $(t_V^* \circ src_G) \sqsubseteq_{V_{\mathcal{C}}^*} (src_{\mathcal{C}} \circ t_E)$*, and* $(t_V^* \circ tar_G) \sqsubseteq_{V_{\mathcal{C}}^*} (tar_{\mathcal{C}} \circ t_E)$*.*

$\mathcal{C}$-typed graphs reflect the inheritance of attributes and methods from the object-oriented paradigm. Notice that they are ordinary hypergraphs typed over a class-model graph. However, the typing morphism is more flexible than the traditional one (CORRADINI; MONTANARI; ROSSI, 1996), (RIBEIRO, 1996): a $\mathcal{C}$-typed graph edge e can be incident to any $\mathcal{C}$-typed graph node v as long as its typing edge $t_E(e)$ (in $\mathcal{C}$) is incident to a node type v' (also in $\mathcal{C}$), such that $t_V(v)$ and v' are connected by the underlying order relation (i.e., $t_V(v) \sqsubseteq_{V_{\mathcal{C}}}^* v'$). This definition reflects the fact that an object can use any attribute belonging to one of its primitive classes, since it was inherited when the class was specialized. Example 3.1 illustrates this idea.

Example 3.1 ($\mathcal{C}$-typed graph) *Figure 3.1 shows a $\mathcal{C}$-typed graph* $G = \langle \{F, Egg, Integer, Color\}, \{is, coord, shade\}, src, tar \rangle$ *over the class-model graph $\mathcal{C}$ portrayed in Figure 2.3. The typing morphism is revealed by the names between brackets, for the sake of clarity. Notice that an ellipse has no attribute directly connected to it in the class-model graph $\mathcal{C}$. However, since an* ellipse *is a specialized* shape*, it inherits all its attributes, and so the graph is well typed.*

All attributes belonging to the $\mathcal{C}$-typed graph on Figure 3.1 are allowed by Definition 3.1: the referred graph has three edges, namely is *(typed as* consists*),* coord *(typed as* pos*), and* shade *(typed as* color*); for* coord *we have (the same can be done to the other two):*

$$\begin{aligned}(t_V^* \circ src)(coord) = ellipse \;&\sqsubseteq_{V_{\mathcal{C}}^*}\; shape = (src_{\mathcal{C}} \circ t_E)(coord)\\ (t_V^* \circ tar)(coord) = Integer\,Integer \;&\sqsubseteq_{V_{\mathcal{C}}^*}\; Integer\,Integer = (tar_{\mathcal{C}} \circ t_E)(coord)\end{aligned}$$

Relations between $\mathcal{C}$-typed graphs can be defined as morphisms between such structures. The basic difference between ordinary typed graph morphisms and $\mathcal{C}$-typed graph morphism is that sources and targets do not need to be preserved *per se*, but only be compatible with the inheritance/overriding relation in the underlying class-model graph.

Notation: For all diagrams presented in the rest of this text, $\mapsto$-arrows denote total morphisms, $\hookrightarrow$-arrows denote injections, whereas $\rightarrow$-arrows denote arbitrary morphisms (possibly partial). For a partial function $f : A \to B$, $dom(f) \subseteq A$ represents its domain of definition, $f? : dom(f) \hookrightarrow A$ and $f! : dom(f) \mapsto B$ denote the corresponding domain inclusion and domain restriction. Each partial function f can be factorized into components $f?$ and $f!$.

Definition 3.2 ($\mathcal{C}$-typed graph morphism) *Let* $G_1^{\mathcal{C}} = \langle G_1, t_1, \mathcal{C} \rangle$ *and* $G_2^{\mathcal{C}} = \langle G_2, t_2, \mathcal{C} \rangle$ *be two $\mathcal{C}$-typed graphs typed over the same class-model graph* $\mathcal{C} = \langle V_{\sqsubseteq},$

$E_{\sqsubseteq}$, L, src, tar, $lab\rangle$. *A* $\mathcal{C}$-typed graph morphism $h : G_1^{\mathcal{C}} \to G_2^{\mathcal{C}}$ *between* $G_1^{\mathcal{C}}$ *and* $G_2^{\mathcal{C}}$*, is a pair of partial functions* $h = \langle h_V : V_{G_1} \to V_{G_2}, h_E : E_{G_1} \to E_{G_2}\rangle$ *such that the diagram (in category* **SetP***)*

$$\begin{tikzcd}
E_{G_1} \arrow[d, mapsto, "{src_{G_1}, tar_{G_1}}"'] & dom(h_E) \arrow[l, hook', "h_E?"'] \arrow[r, mapsto, "h_E!"] & E_{G_2} \arrow[d, mapsto, "{src_{G_2}, tar_{G_2}}"] \\
V_{G_1}^{*} \arrow[rr, "h_V^{*}"] & & V_{G_2}^{*}
\end{tikzcd}$$

commutes, for all elements $v \in dom(h_V)$*,* $(t_{2V} \circ h_V)(v) \sqsubseteq_{V_{\mathcal{C}}} t_{1V}(v)$*, and for all elements* $e \in dom(h_E)$*,* $(t_{2E} \circ h_E)(e) \sqsubseteq_{E_{\mathcal{C}}} t_{1E}(e)$*. If* $(t_{2E} \circ h_E)(e) = t_{1E}(e)$ *for all elements* $e \in dom(h_E)$*, the morphism is said to be* strict.

A graph morphism is a mapping which preserves hyperarcs sources and targets. A typed graph morphism also preserves (node and edge) types. Ordinary typed graph morphisms, however, cannot describe correctly morphisms on object-oriented systems because the existing inheritance relation among objects causes that actions available for objects of a certain kind are valid to *all* objects derived from it. So, an object can be viewed as not being uniquely typed, but having a type *set* (namely, the set of all types it is connected via the inheritance relation). Defining a graph morphism compatible with the underlying order relations assures that polymorphism can be applied consistently.

The meaning of the connection of two elements $x \dashrightarrow y$ by the inheritance relation is the usual: in any place that an object of type[1] y is expected, an object of type x can appear, since an object of type x is also an object of type y. A $\mathcal{C}$-typed graph morphism reflects this concept, since two nodes can be identified through the morphism as long as they are connected by the inheritance relation within the class-model graph. Similarly, two arcs can be identified by a $\mathcal{C}$-typed graph morphism if their types are related through the overriding relation. Since attribute arcs types can only be related (under the reflexive closure of that relation) to themselves, two of them can only be identified if they have the *same* type in the underlying class-model graph. A message, however, can be identified with any other message which redefines it. The reason for that will be clear in Section 3.3.

It should be noticed that the arity of methods is preserved by the morphism, since two hyperarcs can only be mapped if they have the same number of parameters with compatible types.

Example 3.2 ($\mathcal{C}$-typed graph morphism) *Figure 3.2 shows a possible morphism between the* $\mathcal{C}$*-typed graphs* G_1 *and* G_2 *(in dashed lines), both typed over the class-model graph portrayed in Figure 2.3. Notice that the nodes in graphs* G_1 *and* G_2 *have the same names as in the class-model graph, so the typing morphisms between graphs* G_1 *and* G_2 *and the graph on Figure 2.3 need not to be shown. Notice that since a* Drawing *is a* Figure*, and a* circle *is a* round *shape, the morphism is allowed.*

[1]The word *type* is used here in a less strict sense than it is used in programming language design texts. Although the literature makes a difference on subtyping and inheritance relationships between objects (COOK; HILL; CANNING, 1989), such distinction will not be made here, since this work is being developed in an higher level of abstraction. It is hoped that this will not cause any confusion to the reader.

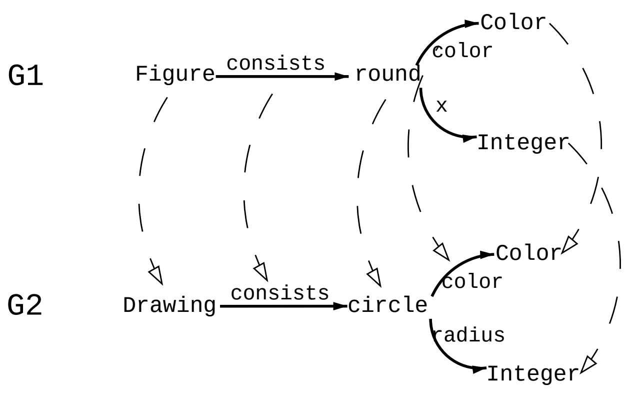

Figure 3.2: $\mathcal{C}$-typed graph morphism

The algebraic approach to graph grammars rely on categorical constructs, especially on colimits, to express most of its results. Having graphs expressed in a category other than **GraphP** (graphs as objects, and partial graph morphisms as arrows, Definition D.15) or **GraphP(T)** (graphs typed over a type graph T as objects, and partial typed graph morphisms as arrows, Definition D.20) — frequently used by the algebraic single-pushout approach (LÖWE, 1991), (EHRIG et al., 1996) — is useful, in the sense that if the same constructs used by the theory of graph grammars can be proven to exist in the new setting, the conclusions drawn from the former could be automatically transferred to the latter. The following results are needed to prove that $\mathcal{C}$-typed graphs over a fixed class-model graph $\mathcal{C}$ and their morphisms constitute a category.

Lemma 3.1 *$\mathcal{C}$-typed graph morphisms are closed under composition.*

Proof: Let $\mathcal{C} = \langle V_{\sqsubseteq}, E_{\sqsubseteq}, L, src, tar, lab\rangle$ be a class-model graph, $G_1^{\mathcal{C}} = \langle G_1, t_1, \mathcal{C}\rangle$, $G_2^{\mathcal{C}} = \langle G_2, t_2, \mathcal{C}\rangle$, and $G_3^{\mathcal{C}} = \langle G_3, t_3, \mathcal{C}\rangle$ be $\mathcal{C}$-typed graphs, and $f = \langle f_V, f_E\rangle : G_1^{\mathcal{C}} \to G_2^{\mathcal{C}}$ and $g = \langle g_V, g_E\rangle : G_2^{\mathcal{C}} \to G_3^{\mathcal{C}}$ be two $\mathcal{C}$-typed graph morphisms. The compound morphism $g \circ f$ can be constructed componentwise: $g \circ f = \langle g_V \circ f_V, g_E \circ f_E\rangle : G_1^{\mathcal{C}} \to G_3^{\mathcal{C}}$. It results in a $\mathcal{C}$-typed graph morphism: for each $f_E(e) \in dom(g_E)$, $(g_V^* \circ src_{G_2})(f_E(e)) = (src_{G_3} \circ g_E)(f_E(e))$, and since partial function composition is associative, we have that $(g_V^* \circ (src_{G_2} \circ f_E))(e) = (src_{G_3} \circ (g_E \circ f_E))(e)$. But for each $e \in dom(f_E)$, $(f_V^* \circ src_{G_1})(e) = (src_{G_2} \circ f_E)(e)$, and then $(g_V^* \circ (f_V^* \circ src_{G_1}))(e) = (src_{G_3} \circ (g_E \circ f_E))(e)$, or $((g_V^* \circ f_V^*) \circ src_{G_1})(e) = (src_{G_3} \circ (g_E \circ f_E))(e)$, for all $e \in dom(g \circ f)$. Equally, for each $f_E(e) \in dom(g_E)$, $(g_V^* \circ tar_{G_2})(f_E(e)) = (tar_{G_3} \circ g_E)(f_E(e))$, and since partial function composition is associative, we have that $(g_V^* \circ (tar_{G_2} \circ f_E))(e) = (tar_{G_3} \circ (g_E \circ f_E))(e)$. But for each $e \in dom(f_E)$, $(f_V^* \circ tar_{G_1})(e) = (tar_{G_2} \circ f_E)(e)$, and then $(g_V^* \circ (f_V^* \circ tar_{G_1}))(e) = (tar_{G_3} \circ (g_E \circ f_E))(e)$, or $((g_V^* \circ f_V^*) \circ tar_{G_1})(e) = (tar_{G_3} \circ (g_E \circ f_E))(e)$, for all $e \in dom(g \circ f)$. Additionally, for all elements $v \in dom(f_V)$, $(t_{2V} \circ f_V)(v) \sqsubseteq_{V_{\mathcal{C}}} t_{1V}(v)$, and for all elements $v \in dom(g_V)$, $(t_{3V} \circ g_V)(v) \sqsubseteq_{V_{\mathcal{C}}} t_{2V}(v)$ (f and g are both $\mathcal{C}$-typed graph morphisms). Then, for all $v \in V_{G_1}$, $(t_{2V} \circ f_V)(v) \sqsubseteq_{V_{\mathcal{C}}} t_{1V}(v)$, and for all $f_V(v) \in dom(g_V)$, $(t_{3V} \circ g_V)(f_V(v)) \sqsubseteq_{V_{\mathcal{C}}} t_{2V}(f_V(v))$, and so (since partial function composition is associative) $(t_{3V} \circ g_V \circ f_V)(v) \sqsubseteq_{V_{\mathcal{C}}} (t_{2V} \circ f_V)(v)$. Since $\sqsubseteq_{V_{\mathcal{C}}}$ is (by definition) a transitive relation, $(t_{3V} \circ g_V \circ f_V)(v) \sqsubseteq_{V_{\mathcal{C}}} t_{1V}(v)$. Likewise, for all $f_E(e) \in dom(g_E)$, $(t_{3E} \circ g_E \circ f_E)(e) \sqsubseteq_{V_{\mathcal{C}}} (t_{2E} \circ f_E)(e)$, $(t_{2E} \circ f_E)(e) \sqsubseteq_{E_{\mathcal{C}}} t_{1E}(e)$, and since $\sqsubseteq_{E_{\mathcal{C}}}$ is transitive, $(t_{3E} \circ g_E \circ f_E)(e) \sqsubseteq_{E_{\mathcal{C}}} t_{1E}(e)$. Thus, $h \circ f$ is a $\mathcal{C}$-typed graph morphism. □

Lemma 3.2 *Composition of $\mathcal{C}$-typed graph morphisms is associative.*

Proof: Let $\mathcal{C} = \langle V_{\sqsubseteq}, E_{\sqsubseteq}, L, src, tar, lab\rangle$ be a class-model graph, $G_1^{\mathcal{C}} = \langle G_1, t_1, \mathcal{C}\rangle$, $G_2^{\mathcal{C}} = \langle G_2, t_2, \mathcal{C}\rangle$, $G_3^{\mathcal{C}} = \langle G_3, t_3, \mathcal{C}\rangle$, and $G_4^{\mathcal{C}} = \langle G_4, t_4, \mathcal{C}\rangle$ be $\mathcal{C}$-typed graphs, and $f = \langle f_V, f_E\rangle : G_1^{\mathcal{C}} \to G_2^{\mathcal{C}}$, $g = \langle g_V, g_E\rangle : G_2^{\mathcal{C}} \to G_3^{\mathcal{C}}$, and $h = \langle h_V, h_E\rangle : G_3^{\mathcal{C}} \to G_4^{\mathcal{C}}$ be $\mathcal{C}$-typed graph morphisms. $\mathcal{C}$-typed graph morphisms composition is done componentwise, so $(h \circ g) \circ f = \langle (h_V \circ g_V) \circ f_V, (h_E \circ g_E) \circ f_E\rangle$. Then, for each $v \in V_{G_1}$, $((h_V \circ g_V) \circ f_V)(v) = (h_V \circ g_V)(f_V(v)) = h_V(g_V(f_V(v))) = h_V \circ ((g_V \circ f_V)(v)) = (h_V \circ (g_V \circ f_V))(v)$, by associativity of partial function composition. The same reasoning can be applied to each $e \in E_{G_1}$, $((h_E \circ g_E) \circ f_E)(e) = (h_E \circ g_E)(f_E(e)) = h_E(g_E(f_E(e))) = h_E \circ ((g_E \circ f_E)(e)) = (h_E \circ (g_E \circ f_E))(e)$ proving that composition of $\mathcal{C}$-typed graph morphisms is associative.
□

Lemma 3.3 *The identity morphism is a $\mathcal{C}$-typed graph morphism.*

Proof: The identity morphism for a given $\mathcal{C}$-typed graph $G^{\mathcal{C}} = \langle G, t, \mathcal{C}\rangle$ is the trivial morphism $id_G = \langle id_V, id_E\rangle : G \to G$, where for any vertex $v \in V_G$, $id_V(v) = v$, and for any edge $e \in E_G$, $id_E(e) = e$. Then, for all $e \in dom(id_E) = E_G$, $(id_V^* \circ src_G)(e) = id_V^*(src_G(e)) = src_G(e) = src_G(id_E(e)) = (src_G \circ id_E)(e)$. Also, for all $v \in V_G$, $(t_V \circ id_V)(v) = t_V(id_V(v)) = t_V(v)$, and for all $e \in E_G$, $(t_E \circ id_E)(e) = t_E(id_E(e)) = t_E(e)$. Since both $\sqsubseteq_{V_{\mathcal{C}}}$ and $\sqsubseteq_{E_{\mathcal{C}}}$ are reflexive, $(t_V \circ id_V)(v) = t_V(id_V(v)) = t_V(v) \sqsubseteq_{V_{\mathcal{C}}} t_V(v)$ and $(t_E \circ id_E)(e) = t_E(id_E(e)) = t_E(e) \sqsubseteq_{E_{\mathcal{C}}} t_E(e)$. □

Theorem 3.1 (Category CGraphP($\mathcal{C}$)) *There is a category* **CGraphP**$(\mathcal{C})$ *which has $\mathcal{C}$-typed graphs as objects and $\mathcal{C}$-typed graph morphisms as arrows.*

Proof: Lemma 3.1 proves that the composition of two $\mathcal{C}$-typed graph morphisms is a $\mathcal{C}$-typed graph morphism. Lemma 3.2 states that composition of $\mathcal{C}$-typed graph morphisms is associative. The identity morphism, as described in Lemma 3.3, is a $\mathcal{C}$-typed graph morphism, and for any given $\mathcal{C}$-typed graph $G^{\mathcal{C}} = \langle G, t, \mathcal{C}\rangle$ is the trivial morphism $id_G = \langle id_V, id_E\rangle : G \to G$, where for any vertex $v \in V_G$, $id_{GV}(v) = v$, and for any edge $e \in E_G$, $id_{GE}(e) = e$. So, given any $\mathcal{C}$-typed graph morphism $h = \langle h_V, h_E\rangle : G_1^{\mathcal{C}} \to G_2^{\mathcal{C}}$, for any vertex $v \in V_{G_1}$, $(h_V \circ id_{G_1V})(v) = h_V(id_{G_1V}(v)) = h_V(v) = id_{G_2V}(h_V(v)) = (id_{G_2V} \circ h_V)(v)$. Similarly, for any edge $e \in E_{G_1}$, $(h_E \circ id_{G_1E})(e) = h_E(id_{G_1E}(e)) = h_E(e) = id_{G_2E}(h_E(e)) = (id_{G_2E} \circ h_E)(e)$.

The existence of an identity morphism for each object, binary composition of morphisms, and associativity of composition proves that **CGraphP**$(\mathcal{C})$ is a category. □

Since **CGraphP**$(\mathcal{C})$ is a category, the existence of categorical constructs within it can be investigated. However, the general definition of $\mathcal{C}$-typed graphs must be narrowed to correctly represent object-oriented systems. To achieve this goal, some additional functions and structures will be defined next.

Definition 3.3 (Attribute and message sets) *Let $\langle G, t, \mathcal{C}\rangle$ be a $\mathcal{C}$-typed graph, with $G = \langle V_G, E_G, src_G, tar_G\rangle$ and $\mathcal{C} = \langle V_{\sqsubseteq}, E_{\sqsubseteq}, L, src, tar, lab\rangle$. Then the following functions are defined:*

- *the* attribute set function $attr_G : V_G \to 2^{E_G}$ *return for each vertex $v \in V_G$ the set $\{e \in E_G \mid src_G(e) = v \wedge lab(t_E(e)) = \text{attr}\}$;*
- *the* message set function $msg_G : V_G \to 2^{E_G}$ *returns for each vertex $v \in V_G$ the set $\{e \in E_G \mid tar_G(e) = v \wedge lab(t_E(e)) = \text{msg}\}$;*

- *the* extended attribute set function, $attr^*_{\mathcal{C}} : V \longrightarrow 2^E$, *where* $attr^*_{\mathcal{C}}(v) = \{e \in E \mid lab(e) = \text{attr} \wedge src(e) \in \ \uparrow v\}$*;*
- *the* extended message set function, $msg^*_{\mathcal{C}} : V \longrightarrow 2^E$, *where* $msg^*_{\mathcal{C}}(v) = \{e \in E|_{\text{msg}} \mid tar(e) \in \ \uparrow v \wedge \neg \exists e' \in E|_{\text{msg}} : tar(e') \in \ \uparrow v \wedge e' \sqsubseteq_E e\}$*.*

The *attribute/message set function* and the *extended attribute/extended message set function* will help define some other functions, relations and structures along this text. Basically, for any vertex v of a $\mathcal{C}$-typed graph, the *attribute set function* returns the set of all attribute arcs having v as their source. Similarly, given a class-model graph and a vertex v belonging to it, the *extended attribute set function* returns the set of all attribute arcs whose source is v or any other vertex to which v connected via the inheritance relation $\sqsubseteq^*_V$.

The *message set function* returns all messages an object within a $\mathcal{C}$-typed graph is currently receiving, while the *extended message set function* returns all messages an object of a specific type may receive. Notice that message redefinition within objects, expressed by the overriding relation $\sqsubseteq^*_E$ on the class-model graph, must be taken into account, since the redefinition of a class method implies that only the redefined method can be seen within the scope of a specialized class.

For any $\mathcal{C}$-typed graph $\langle G, t, \mathcal{C}\rangle$ there is a total function $t^*_E : 2^{E_G} \longrightarrow 2^{E_{\mathcal{C}}}$, which can be viewed as the extension of the typing function to edge (or node) sets. The function t^*_E, when applied to a set $E \in 2^{E_G}$, returns the set $\{t_E(e) \in E_{\mathcal{C}} \mid e \in E\} \in 2^{E_{\mathcal{C}}}$. Notation $t^*_E|_{\text{msg}}$ and $t^*_E|_{\text{attr}}$ will be used to denote the application of t^*_E to sets containing exclusively message and attribute (respectively) hyperarcs. Now, given the functions already defined, we can present a definition of the kind of graph which represents an object-oriented system.

Definition 3.4 (Object-oriented graph) *Let $\mathcal{C}$ be a class-model graph. A $\mathcal{C}$-typed graph $\langle G, t, \mathcal{C}\rangle$ is an* object-oriented graph *if and only if all squares in the diagram (in* **Set***)*

$$\begin{tikzcd}
2^{E_G} \arrow[d, maps to, "t^*_E|_{\text{msg}}"'] & V_G \arrow[l, maps to, "msg_G"'] \arrow[r, maps to, "attr_G"] \arrow[d, maps to, "t_V"'] & 2^{E_G} \arrow[d, maps to, "t^*_E|_{\text{attr}}"] \\
2^{E_{\mathcal{C}}} & V_{\mathcal{C}} \arrow[l, maps to, "msg^*_{\mathcal{C}}"'] \arrow[r, maps to, "attr^*_{\mathcal{C}}"] & 2^{E_{\mathcal{C}}}
\end{tikzcd}$$

commute. If, for each $v \in V_G$, the function $t^*_E|_{\text{attr}}(attr_G(v))$ is injective, $G^{\mathcal{C}}$ is said a strict *object-oriented graph. If $t^*_E|_{\text{attr}}(attr_G(v))$ is also surjective, $G^{\mathcal{C}}$ is called a* complete *object-oriented graph.*

It is important to realize what sort of message is allowed to target a vertex in an object-oriented graph. The left square on the diagram depicted in Definition 3.4 ensures that an object can only have a message edge targeting itself if that message is typed over one of those edges returned by the extended message set function. It means that the only messages allowed are the least ones in the redefinition chain to which the typing message belongs. This is compatible with the notion of *dynamic binding*, since the method actually called by any object is determined by the actual object present at a certain computation state.

Object-oriented graphs can also be *strict* or *complete*. Strict object-oriented graphs require that nodes do not possess two attribute arcs typed as the same element in the underlying class-model graph. Injectivity of all $t^*_E|_{\mathrm{attr}}(attr_G(v))$, $v \in V_G$, express that all attribute arcs are typed differently (i.e., an object has no exceeding attribute). For an object-oriented graph to be *complete*, however, it is also necessary that *all* attributes defined on all levels along the class-model graph (via the inheritance relation on nodes) are present. The definition of a complete object-oriented graph is coherent with the notion of inheritance within the object-oriented framework, since an object inherits all attributes, and exactly those, from its primitive classes.

Object-oriented systems are often composed by a large number of objects, which can receive messages from other objects (including themselves) and react to the messages received. Object-oriented graphs may also have as many objects as desired, and even if the number and type of attributes (arcs) in each object (vertex) is limited, the number of vertices in the graph representing the system is not.

Object-oriented graphs are just a special kind of $\mathcal{C}$-typed graphs. Therefore we can define a subcategory of $\mathbf{CGraphP}(\mathcal{C})$, as follows.

Definition 3.5 (Category $\mathbf{OOGraphP}(\mathcal{C})$) $\mathbf{OOGraphP}(\mathcal{C})$ *is the full subcategory of* $\mathbf{CGraphP}(\mathcal{C})$ *having object-oriented graphs as objects.*

3.3 Object-oriented graph grammars

Complete object-oriented graphs (Definition 3.4) can model all elements belonging to object-oriented systems. However, in order to capture the system evolution through time, we need a graph grammar formalism to be introduced.

A *graph production*, or simply a *rule*, specifies how a system configuration may be changed. A rule has a *left-hand side* and a *right-hand side*, which are both strict object-oriented graphs, and a $\mathcal{C}$-typed graph morphism to determine what should be altered. Intuitively, a system configuration change occurs in the following way: all items belonging to the left-hand side must be present at the current state to allow the rule to be applied; all items mapped from the left to the right-hand side (via the graph morphism) will be preserved; all items not mapped by the rule morphism will be deleted from the current state; and all items existent in the right-hand side but not in the left-hand side will be added to the current state to obtain the next one.

Rule restrictions may vary, depending on what is intended for them to represent/implement. Unrestricted rules give rise to very powerful systems in terms of representation capabilities, but they also lead to many undecidable problems. Restrictions are needed not just to make interesting problems decidable (which is important *per se*) but also to reflect existing constraints in the actual systems we are trying to model. All rule restrictions presented in this text are object-oriented programming principles, as described next.

First of all, no object may have its type altered nor can any two different elements be merged during the evolution of a computation. This is accomplished by requiring the rule morphism to be injective on nodes and arcs (meaning that different elements cannot be merged by the rule application), and the mapping on nodes to be invertible. The mapping on nodes must be invertible to assure that object types are not modified during a course of computation steps, as explained in Example 3.3 below.

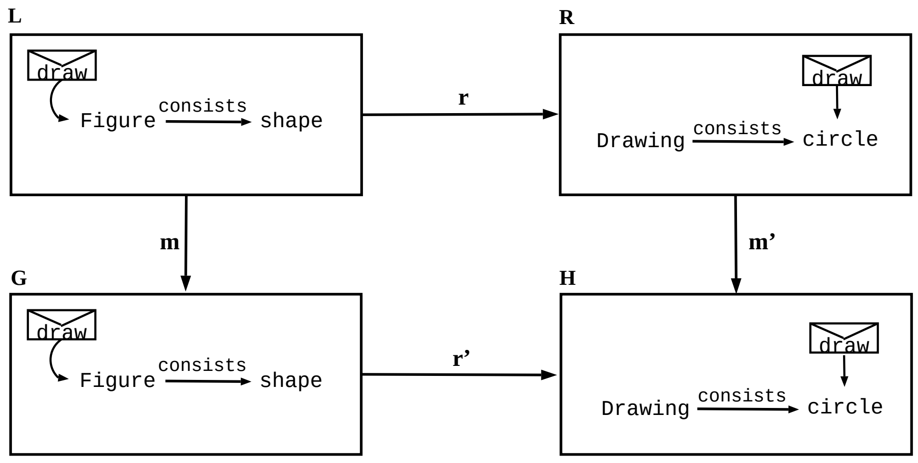

Figure 3.3: Changing object types during a course of computation

Example 3.3 (Object type change) *This example clarifies the meaning of the rule restriction concerning the node mapping being invertible. Consider the object-oriented direct derivation portrayed in Figure 3.3, where elements are named after their types, which are the ones in the class-model graph of Figure 2.3.*

The rule mapping is a perfectly legal $\mathcal{C}$-typed graph morphism, since a Drawing *is a* Figure *and a* circle *is a* shape*. The match morphism is identical on types, and the result of this direct derivations is given by the object-oriented H and $\mathcal{C}$-typed graph morphisms r' and m'.*

G is the graph representing the object-oriented system being transformed, and H is its next step in a computation trace. However, the object typed as Figure *in G is mapped (through the morphism r') to an object typed as* Drawing*; likewise, the object typed as* shape *is mapped to a* circle*. The morphism r' keeps track of the evolution of any object as time passes. It maps the* same *object during any number of computation steps. If the rule morphism is not required to be invertible, this situation can occur, and an object can have its type altered in execution time. This is not the case in any object-oriented programming language, for it will be impossible to determine, in compilation time, if the program is safe or even correct. So, it is not allowed by the formalism either.*

The left-hand side of a rule is required to contain exactly one element of type message, and this particular message must be deleted by the rule application, i.e., each rule represents an object reaction to a message which is consumed in the process. This demand poses no unreasonable restriction, since systems may have many rules specifying reactions to the same type of message (non-determinism) and many rules can be applied in parallel if their triggers are present at an actual state and the referred rules are not in conflict (EHRIG et al., 1996). Systems concurrent capabilities are so expressed by the grammar rules, which can be applied concurrently (accordingly to the graph grammar semantics), so one object can treat any number of messages at the same time.

Additionally, at most one object having attributes will be allowed on the left-hand side of a rule, along with the requirement that this same object must be the target of the above cited message. This restriction implements the principle of *information hiding*, which states that the internal configuration (implementation) of an object can only be visible, and therefore accessed, by itself.

Finally, although message attributes can be deleted (so they can have their value altered, a corresponding attribute (of the same type) must be present in the rule right-hand side, in order to prevent an object from gaining or losing attributes along the computation. Notice that this is a *rule* restriction, for if a vertex is deleted, its incident edges will also be deleted. This situation will be explored next, as different kinds of rules are defined.

Definition 3.6 (Basic object-oriented rule) *A* basic object-oriented rule r *is a $\mathcal{C}$-typed graph morphism $r = \langle r_V, r_E \rangle : L^{\mathcal{C}} \to R^{\mathcal{C}}$, where $L^{\mathcal{C}} = \langle L, t_L, \mathcal{C} \rangle$ and $R^{\mathcal{C}} = \langle R, t_R, \mathcal{C} \rangle$ are strict object-oriented graphs over a class-model graph $\mathcal{C}$, holding the following properties:*

- *r_V is injective and invertible, r_E is injective,*
- *$\{e \in E_L \mid lab_{\mathcal{C}}(t_L(e)) = \text{msg}\}$ is a singleton, whose unique element is called* left-hand side message*, and whose target object is called* attribute vertex*,*
- *if $\{v \in V_L \mid e \in E_L, src_L(e) = v, lab_{\mathcal{C}}(t_L(e)) = \text{attr}\} \neq \emptyset$, then it is a singleton, whose unique element is the attribute vertex,*
- *each rule implements a response to a message call at the level (class) the message is defined, i.e., if l is the left-hand side message, then $(t_V \circ tar_L)(l) = (tar_{\mathcal{C}} \circ t_E)(l)$,*
- *the left-hand side message does not belong to the domain of r, i.e., if l is the left-hand side message, then $l \notin dom(r)$, and*
- *for all $v \in V_L$ there is a bijection $b : \{e \in E_L \mid lab_{\mathcal{C}}(t_L(e)) = \text{attr} \wedge src_L(e) = v\} \leftrightarrow \{e \in E_R \mid lab_{\mathcal{C}}(t_R(e)) = \text{attr} \wedge src_R(e) = r_V(v)\}$, such that $t_R \circ b = t_L$ and $t_L \circ b^{-1} = t_R$.*

Different kinds of rules can be defined based on basic object-oriented rules. We define three of them: strict object-oriented rules (Definition 3.7) do not allow for object creation of deletion; object-oriented rules with creation (Definition 3.8) allow the creation of new objects; and general object-oriented rules (Definition 3.9) permit both creation and deletion operations.

Definition 3.7 (Strict object-oriented rule) *A* strict object-oriented rule r *is a basic object-oriented rule $r = \langle r_V, r_E \rangle : L^{\mathcal{C}} \to R^{\mathcal{C}}$ where r_V is total and surjective.*

A strict object-oriented rule contains the restrictions connected to the object-oriented programming paradigm, already presented, along with restrictions to assure that no object is ever created or deleted along the computation. This goal is achieved by requiring a bijection between the vertex sets from the left and the right-hand side of a rule.

Definition 3.8 (Object-oriented rule with object creation) *An* object-oriented rule with object creation *is a basic object-oriented rule $r = \langle r_V, r_E \rangle : L^{\mathcal{C}} \to R^{\mathcal{C}}$ where r_V is total, and for all $v \in V_R$, if $v \notin im(r_V)$ the diagram*

$$\begin{tikzcd}
2^{E_R} \arrow[d, maps to, "t_E^*|_{\text{msg}}"'] & V_R \arrow[l, maps to, "msg_R"'] \arrow[r, maps to, "attr_R"] \arrow[d, maps to, "t_V"'] & 2^{E_R} \arrow[d, leftrightarrow, "t_E^*|_{\text{attr}}"] \\
2^{E_{\mathcal{C}}} & V_{\mathcal{C}} \arrow[l, maps to, "msg_{\mathcal{C}}^*"'] \arrow[r, maps to, "attr_{\mathcal{C}}^*"] & 2^{E_{\mathcal{C}}}
\end{tikzcd}$$

commutes, and $t^*_E(attr_R(v))$ is a bijection.

Object-oriented rules with object creation differ from strict object-oriented rules in two aspects: r_V is not necessarily surjective, so new vertices can be added by the rule, but all created vertices must have exactly the attributes defined along its class-model graph, to assure that the resulting graph (when the rule is applied) remains complete.

Definition 3.9 (General object-oriented rule) *A general object-oriented rule is a object-oriented rule with object creation $r = \langle r_V, r_E \rangle : L^{\mathcal{C}} \to R^{\mathcal{C}}$ where $dom(r_V) = V_L$ or $dom(r_V) = V_L \backslash \{\text{attribute vertex}\}$.*

General object-oriented rules allow object deletion. Notice, however, that an object can only delete itself, as required by the principles of good object-oriented programming.

Having different types of object-oriented rules give rise to different sorts of computations. However, regardless of the sort of rule chosen, the remaining definitions (namely matches and rule applications) can be stated in an uniform way. Henceforth, the term "object-oriented rule" refers to any of the object-oriented rules types already defined by Definitions 3.7, 3.8, and 3.9.

Definition 3.10 (Object-oriented match) *Given a strict object-oriented graph $G^{\mathcal{C}}$ and an object-oriented rule $r : L^{\mathcal{C}} \to R^{\mathcal{C}}$, an* object-oriented match *between $L^{\mathcal{C}}$ and $G^{\mathcal{C}}$ is a $\mathcal{C}$-typed graph morphism $m = \langle m_V, m_E \rangle : L^{\mathcal{C}} \to G^{\mathcal{C}}$ such that m_V is total, m_E is total and injective, and for any two elements $a, b \in L$, if $m(a) = m(b)$ then either $a, b \in dom(r)$ or $a, b \notin dom(r)$.*

The role of a *match* is to detect a situation when a rule can be applied. It occurs whenever a rule left-hand side is present somewhere within the system graph. Notice that distinct vertices can be identified by the match morphism. This is sensible, because an object can point to itself through one of its attributes, or pass itself as a message parameter to another object. However, it would make no sense to identify different attributes or messages, so the edge component of the match morphism is required to be injective. The match morphism also requires that elements preserved by the rule cannot be identified with elements deleted by it. This restrictions is known in the literature as the *identification condition* and it is there to prevent undesirable collateral effects when a rule is applied (namely, that an element meant to be preserved is actually deleted).

A *direct derivation*, or *derivation step*, represents a discrete system change in time, i.e., a *rule application* over an actual system specified as a graph.

Definition 3.11 (Object-oriented direct derivation) *Let $\mathcal{U}_t$ be the forgetful functor which ignores the typing structure of an object-oriented graph (Definition A.1). Given a complete object-oriented graph $G^{\mathcal{C}} = \langle G, t_G, \mathcal{C} \rangle$, an object-oriented rule $r : L^{\mathcal{C}} \to R^{\mathcal{C}}$, and an object-oriented match $m : L^{\mathcal{C}} \to G^{\mathcal{C}}$, their* object-oriented direct derivation*, or rule* application*, can be computed in two steps:*

1. *Construct the pushout of $\mathcal{U}_t(r) : \mathcal{U}_t(L^{\mathcal{C}}) \to \mathcal{U}_t(R^{\mathcal{C}})$ and $\mathcal{U}_t(m) : \mathcal{U}_t(L^{\mathcal{C}}) \to \mathcal{U}_t(G^{\mathcal{C}})$ in* **HGraphP**, *$\langle H, r' : G \to H, m' : R \to H \rangle$ (EHRIG et al., 1996);*

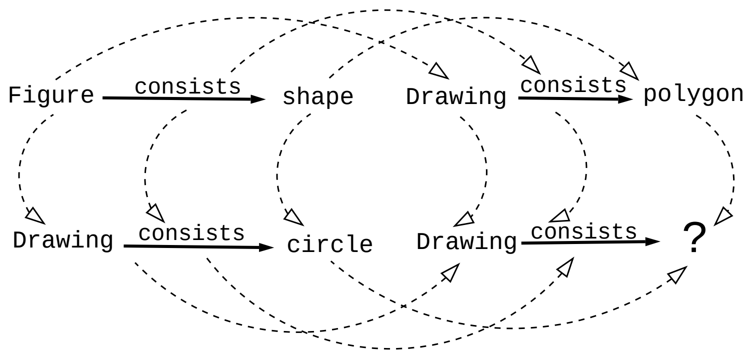

Figure 3.4: Inexistence of correct element typing

2. *equip the result with the following typing structure on nodes and edges, resulting in the graph* $H^{\mathcal{C}} = \langle H, t_H, \mathcal{C}\rangle$ *where,*

   - *for each* $v \in V_H$, $t_H(v) = \sqcap\ \big(t_G(r'^{-1}(v)) \cup t_R(m'^{-1}(v))\big)$,
   - *for each* $e \in E_H|_{\mathrm{attr}}$, $t_H(e) = \sqcap\ \big(t_G(r'^{-1}(e)) \cup t_R(m'^{-1}(e))\big)$,
   - *for each* $e \in E_H|_{\mathrm{msg}}$, $t_H(e) = e'$, *where* $e' \in msg^*_{\mathcal{C}}(t_H(tar_H(e)))$, *and* $e' \sqsubseteq^*_E\ \sqcap\,[t_G(r'^{-1}(e)) \cup t_R(m'^{-1}(e))]$.

The tuple $\langle H^{\mathcal{C}}, r', m'\rangle$ *is then the resulting derivation of rule* r *at match* m*. A direct derivation from graph* $G^{\mathcal{C}}$ *to* $H^{\mathcal{C}}$ *under rule* r *and match* m *is denoted by* $G^{\mathcal{C}} \overset{r,m}{\Rightarrow} H^{\mathcal{C}}$.

An object-oriented derivation collapses the elements identified simultaneously by the rule and by the match, and copies the rest of the context and the added elements, in the very same manner that single-pushout approach does. Actually, the pushout object is built exactly as if the underlying category were **HGraphP**. Element typing is then needed to transform the resulting graph into an object-oriented graph. The typing procedure is performed by getting the greatest lower bound (with respect the partial order relations on nodes and edges) of the elements mapped by morphisms m' and r' to the same element, while the other elements — objects and attributes — have their types merely copied. The object-oriented rule restriction concerning object types — which cannot be altered by the rule — assures that the greatest lower bound of the mapped elements types always exist, as shown in more detail below. Without this restriction, would be impossible to assure the existence of a correct typing structures, as shown in Example 3.4.

Example 3.4 (Inexistence of correct typing) *Consider the four object-oriented graphs in Figure 3.4, typed over the class-model graph in Figure 2.3. Each graph have an equal structure, composed of two nodes and an attribute edge. As usual, elements are named after their types, to simplify the presentation.*

The upper-left graph has a node typed as a Figure, *which* consists *of an element of type* Shape. *The element typed as* Figure *is mapped to elements of type* Drawing *by both morphisms to (respectively) the the upper-right and the lower-left object-oriented graphs. The element of type* Shape, *however, is mapped to an element of type* circle *in the lower-left graph and to an element of type* polygon *in the upper-right graph. This is perfectly legal, because both types circle and polygon are shapes, as indicated by the inheritance relation on nodes. However, there are*

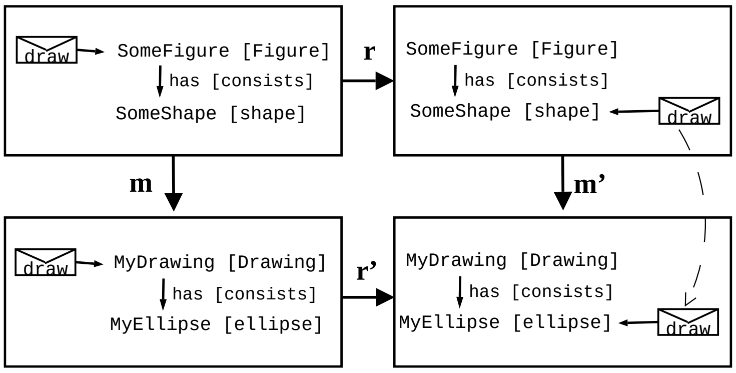

Figure 3.5: Dynamic binding as message retyping

no element which is the greatest lower bound of types circle and polygon in the class-model graph presented in Figure 2.3. Furthermore, by the very definition of a strict relation, the intersection of the down sets of distinct elements which are not related by it is empty.

Messages, however, need some extra care. Since graph $L^{\mathcal{C}}$ contains a single message, which is deleted by the rule application, a message on $H^{\mathcal{C}}$ comes either from $G^{\mathcal{C}}$ or from $R^{\mathcal{C}}$. If it comes from $G^{\mathcal{C}}$, which is an object-oriented graph itself, no retyping is needed. However, if it comes from $R^{\mathcal{C}}$, in order to assure that $H^{\mathcal{C}}$ is also an object-oriented graph, it must be retyped according to the type of the element it is targeting on the graph $H^{\mathcal{C}}$. Notice that this element can have a different type from the one in the rule, since the match can be done to any element belonging to the lower set of the mapped entity. It occurs whenever the object to which the message on the rule is targeting at has a different type to the one it is mapped to by the pushout morphism and, additionally, the message (in the right-hand of the rule) has been redefined on that level.

Figure 3.5 shows a situation where the need for a message retyping is made clear. The typing morphism is shown between brackets, for the sake of readability. Rule r portrays a common situation: the action resulting from a method calling is the calling of another method from one of the attributes belonging to the object. Here, a *Figure* is drawn by making its constituent *shape* to be drawn. However, since the rule left-hand side is matched to a *Drawing* which has an *ellipse* as constituent, and since the method *Draw* is redefined within that level, the resulting message cannot be typed as a *shape Draw*, but as an *ellipse Draw* (indicated by the only explicit arrow from R to H). Notice that m' is still a $\mathcal{C}$-typed graph morphism (although it is not strict anymore). Hence, the result of a direct derivation (when message retyping is performed) is consistent with the occurrence of dynamic binding on the object-oriented computational process.

It is important to realize that an object-oriented direct derivation is well defined, as shown by the proposition below.

Proposition 3.1 *The structure proposed in Definition 3.11 exists.*

Proof: Category **HGraphP** is cocomplete (LÖWE, 1991), so the pushout structure of step 1 in Definition 3.11 exists. It remains to prove that all elements of such structure can be uniquely typed according to the second step of that definition.

The tuple $\langle V_H, r'_V : V_G \to V_H, m'_V : V_R \to V_H \rangle$ is the pushout of arrows $r_V : V_L \to V_R$ and $m_V : V_L \to V_G$ in **SetP**, hence r'_V is injective (because r_V is injective), and r'_V and m'_V are jointly surjective.

For any $h \in V_H$, let $m'^{-1}_V(h) = \{r \in V_R \mid m'_V(r) = h\}$ and $r'^{-1}_V(h) = \{g \in V_G \mid r'_V(g) = h\}$. Now, for any $h \in V_H$, $r'^{-1}_V(h)$ is a singleton (because r'_V is invertible) with unique element g, then, by definition, $\sqcap\, t_G(r'^{-1}_V(h)) = g$; for any $h \in V_H$ such that $m'^{-1}_V(h) \notin im(r_V)$, $m'^{-1}_V(h)$ is also a singleton, and the same reasoning applies; for any $h \in V_H$ such that $m'^{-1}_V(h) \subseteq im(r_V)$, we have that $(t_R \circ m'^{-1}_V)(h) =_{V^*_{\mathcal{C}}} (t_L \circ r^{-1}_V \circ m'^{-1}_V)(h)$; but, for the same element h we have that $(t_G \circ r'^{-1}_V)(h) \sqsubseteq^*_{V^*_{\mathcal{C}}} (t_L \circ m^{-1}_V \circ r'^{-1}_V)(h)$. But, as mentioned before, V_H is a pushout, then $(r^{-1}_V \circ m'^{-1}_V)(h) = (m^{-1}_V \circ r'^{-1}_V)(h)$ and therefore $(t_G \circ r'^{-1}_V)(h) \sqsubseteq^*_{V^*_{\mathcal{C}}} (t_R \circ m'^{-1}_V)(h)$. Since $\sqcap\, t_G(r'^{-1}_V(h))$ is defined, $\sqcap\, (t_G(r'^{-1}_V(h)) \cup t_R(m'^{-1}_V)(h))$ always exists.

For all attributes arcs in E_L, E_R and E_G, the morphism can be reduced to an ordinary partial graph morphism (since they are only connected to themselves via the order relation), and then $\sqcap\, (t_G(r'^{-1}_E(h)) \cup t_R(m'^{-1}_E(h)))$ is always well defined. For message arcs, however, notice that object-oriented rules require that the only message in E_L is deleted by the rule. It means that messages arcs of E_H either come from E_G or from E_R. A message e is typed over the least element from the overriding relation with respect to its actual target vertex's type. This is assured by the fact that the typing edge must belong to the set $msg^*_{\mathcal{C}}(t_H(tar_H(e)))$, which, by definition, assures that there is only one edge of choice (there are no related message arcs in the set $msg^*_{\mathcal{C}}(v)$, for any vertex type v. $\square$

Lemma 3.4 *The object $H^{\mathcal{C}} = \langle H, t_H, \mathcal{C} \rangle$ built according to Definition 3.11 is a complete object-oriented graph.*

Proof: $L^{\mathcal{C}}$, $R^{\mathcal{C}}$ are strict object-oriented graphs, and $G^{\mathcal{C}}$ is a complete one. If $H^{\mathcal{C}}$ is an object-oriented graph, the following diagram can be constructed:

$$\begin{tikzcd}
2^{E_H} \arrow[d, mapsto, "t^*_E|_{\mathrm{msg}}"'] & V_H \arrow[l, mapsto, "msg_H"'] \arrow[r, mapsto, "attr_H"] \arrow[d, mapsto, "t_V"'] & 2^{E_H} \arrow[d, mapsto, "t^*_E|_{\mathrm{attr}}"] \\
2^{E_{\mathcal{C}}} & V_{\mathcal{C}} \arrow[l, mapsto, "msg^*_{\mathcal{C}}"'] \arrow[r, mapsto, "attr^*_{\mathcal{C}}"] & 2^{E_{\mathcal{C}}}
\end{tikzcd}$$

Notice that the set of added vertices $V_R \setminus r_V(V_L)$ can be viewed as a complete object-oriented graph, for it has all necessary attributes. Now, restricting the reasoning to the set of mapped vertices and attributes, one has the following: for each $v \in V_L$, let b_v be the bijection existing between the attribute edges from $A_L \subseteq E_L|_{\mathrm{attr}}$ and $A_R \subseteq E_R|_{\mathrm{attr}}$ defined as the last object-oriented rule restriction in Definition 3.6. Match m between the rule's left-hand side and graph $G^{\mathcal{C}}$ is total on vertices and arcs, and injective on arcs, and by the characteristics of the pushout construction, function m'_E is also total and injective on arcs. Notice that all edges from $G^{\mathcal{C}}$ are either belonging to the image of m_E (the mapped edges) or not (the context edges). Since the context edges are mapped unchanged to the graph $H^{\mathcal{C}}$ (and so there is a natural bijection between them), it must exist a bijection $B : E_G \leftrightarrow E_H$ which implies the existence of the trivial bijection $2^B : 2^{E_G} \to 2^{E_H}$, and since the sets V_G and V_H are isomorphic if we disregard the added vertices (note the existence of an implicit property of an object-oriented rule that prevents it from deleting vertices, since a deletion of a vertex implies the deletion of an edge, which cannot occur, otherwise there would be no bijection b_v), it can be concluded that the right square on the diagram can be constructed. The same reasoning applies to the left square of the diagram, since the rule application assures that messages are typed right. Hence, one can conclude that $H^{\mathcal{C}}$ is a complete object-oriented graph. $\square$

Lemma 3.5 *The morphisms $r' : G^{\mathcal{C}} \to H^{\mathcal{C}}$ and $m' : R^{\mathcal{C}} \to H^{\mathcal{C}}$, built according to Definition 3.11 are $\mathcal{C}$-typed graph morphisms.*

Proof: The morphisms r' and m' preserve the order structure, which is a sufficient condition to assure that they are actually $\mathcal{C}$-typed graph morphisms. $\square$

Next, we show that the an object-oriented derivation is a pushout structure in the category $\mathbf{OOGraphP}(\mathcal{C})$.

Theorem 3.2 (Derivation is a pushout in $\mathbf{OOGraphP}(\mathcal{C})$) *Given an object-oriented graph $G^{\mathcal{C}}$, an object-oriented rule $r : L^{\mathcal{C}} \to R^{\mathcal{C}}$, and an object-oriented match $m : L^{\mathcal{C}} \to G^{\mathcal{C}}$, the resulting derivation of rule r at match m, $\langle H^{\mathcal{C}}, r', m'\rangle$ is a pushout of the arrows r and m in the category $\mathbf{OOGraphP}(\mathcal{C})$.*

Proof: Proposition 3.1 assures that a direct derivation can always be constructed, and Lemmas 3.4 and 3.5 show that the resulting object and morphisms belong to the category $\mathbf{OOGraphP}(\mathcal{C})$. Then, let $\langle H^{\mathcal{C}}, r' : G^{\mathcal{C}} \to H^{\mathcal{C}}, m' : R^{\mathcal{C}} \to H^{\mathcal{C}}\rangle$ be the result obtained by application of rule r under match m. Now, let $H'^{\mathcal{C}}$ be an object-oriented graph and $h_R : R^{\mathcal{C}} \to H'^{\mathcal{C}}$, $h_G : G^{\mathcal{C}} \to H'^{\mathcal{C}}$ be two $\mathcal{C}$-typed graph morphisms such that $h_R \circ r = h_G \circ m$. Then let $h : H^{\mathcal{C}} \to H'^{\mathcal{C}}$ be the $\mathcal{C}$-typed graph morphism built as follows: for all graph element (node or edge) $e \in dom(h_R) \cap dom(m')$, $h(m'(e)) = h_R(e)$; for all $e \in dom(h_G) \cap dom(r')$, $h(r'(e)) = h_G(e)$; it is easy to see that, by construction, $h_R = h \circ m'$ and $h_G = h \circ r'$. Notice that h is a $\mathcal{C}$-typed graph morphism, since all elements $e \in H$ are typed over the greatest lower bound (respecting the concerning order relation) of the elements mapped to them. It means that if exists an element $e' \in H'$ such that there are elements $e_g \in G$ and $e_r \in R$ with $h_G(e_g) = e' = h_R(e_r)$, then (h_G and h_R are $\mathcal{C}$-typed graph morphisms) $t_{H'}(e') \sqsubseteq^*_{V_{\mathcal{C}}} t_G(e_g)$ and $t_{H'}(e') \sqsubseteq^*_{V_{\mathcal{C}}} t_R(e_r)$; since $t_H(e_h) = \sqcap\,(t_G(r'^{-1}(e_h)) \cup t_R(m'^{-1}(e_h)))$ if e_h is a vertex or an attribute, and $t_H(e_h) \in msg^*_{\mathcal{C}}(t_H(tar_H(e_h)))$ if e_h is a message, then $t_{H'}(e') \sqsubseteq^*_{V_{\mathcal{C}}} t_H(e_h)$ for any $e' = h(e_h)$. Hence, h is a $\mathcal{C}$-typed graph morphism.

Suppose there is another $\mathcal{C}$-typed graph morphism $h' : H^{\mathcal{C}} \to H'^{\mathcal{C}}$ such that $h_R = h' \circ m'$ and $h_G = h' \circ r'$ but $h' \neq h$. Then there must be at least one graph element $e \in H$ such that $h(e) \neq h'(e)$. But r' and m' are jointly surjective, so all elements of H belong to the domain of h, so if there is an element $e \in H$ such that $h(e) \neq h'(e)$, the equalities $h_R = h' \circ m'$ and $h_G = h' \circ r'$ would not hold. $\square$

Given the graph structures presented earlier and the rules to be applied to them, some interesting properties can be demonstrated. Closure properties are especially interesting, such as the ones expressed below.

Property 3.1 *The class of complete object-oriented graphs is closed under object-oriented derivations using strict object-oriented rules.*

Proof: Let $G^{\mathcal{C}}$ be a complete object-oriented graph, $\langle L^{\mathcal{C}}, r, R^{\mathcal{C}}\rangle$ be a strict object-oriented rule and $\langle L^{\mathcal{C}}, m, G^{\mathcal{C}}\rangle$ be an object-oriented match, and $\langle L^{\mathcal{C}}, r', m'\rangle$ the resulting derivation of rule r at match m. Being r_V is a total bijection, for any $v \in V_L$ $t_L(v) = t_R(r_V(v))$ holds. Since m is a $\mathcal{C}$-typed graph morphism, for any vertex $v \in dom(r_V) \cap dom(m_V)$, $(t_G \circ m_V(v), t_R \circ r_V(v)) isa^*$, and so $t_H \circ r'_V \circ m_V(v) = t_H \circ m'_V \circ r_V(v) = t_G \circ m_V(v)$. So, V_H is isomorphic to V_G.

Now, let b be the bijection existing between the attribute edges from $A_L \subseteq E_L$ and $A_R \subseteq E_R$ defined as the last basic object-oriented rule restriction in Definition 3.6. Notice b is defined over *all* attribute edges of both graphs L and R. The match m between the rule left-hand side and graph G is total on vertices and arcs, and injective on arcs, and by the characteristics of the pushout construction, function m' is also total and injective on arcs. Notice that all edges from G are either belonging to the image of m_E (the mapped edges) or not (the context edges). Since the context edges are mapped unchanged to the graph H (and so there is a natural bijection between them), it must exist a bijection $B : E_G \leftrightarrow E_H$ which implies the existence of the trivial bijection $2^B : 2^{E_G} \to 2^{E_H}$, and since the sets V_G and V_H are equal (up to isomorphism), it can be concluded that $H^{\mathcal{C}}$ is a complete object-oriented graph. □

Property 3.2 *The class of complete object-oriented graphs is closed under object-oriented derivations using object-oriented rules with object creation.*

Proof: (sketch) The same reasoning applied to the proof of Theorem 3.1 can be used to show that, in this case, V_G is isomorphic to a subset of V_H. The additional vertices of V_H are those created by the rule application (i.e., those isomorphic to the set $\{v \in V_R \mid v \notin im(r_V)\}$). But since all $v \notin im(r_V)$ is required to behave like a complete object-oriented graph when considered alone, so its inclusion on H will assure, along with Theorem 3.1, that it is also a complete object-oriented graph. □

Property 3.3 *The class of complete object-oriented graphs is not closed under object-oriented derivations using general object-oriented rules.*

Proof: Assume that the class of complete object-oriented graphs is closed under object-oriented derivations using general object-oriented rules. Let n be a node of an object-oriented complete graph such that n has an attribute e with target a. Let r be an object-oriented rule which deletes an object of the same type as a. The application of rule r to object a will cause attribute e to become a dangling edge; therefore, it will be deleted during rule application. Node n will loose attribute e, so the whole derived graph will not be a complete object-oriented graph anymore. Hence, the class of complete object-oriented graphs is not closed under object-oriented derivations using general object-oriented rules. □

Property 3.3 describes a situation known as deletion in unknown contexts. This situation is very common in distributed systems, where the deletion of an object causes a number of dangling pointers to occur in the system as a whole. So, rules that allow object deletion can be used to detect this kind of undesirable situations within a specification.

An interesting side effect derived from the use of rules that allow object deletion is that any dangling pointer would cause an edge cease to exist. In this case, any rule which takes that particular edge into consideration can no longer be applied (for no match can be found for that rule). When modeling system execution, this situation leads to the prevention of an execution runtime error, which would occur if an attempt to access an object which is no longer there is made.

Definition 3.12 (Object-oriented graph grammar) *An object-oriented graph grammar is a tuple $\langle I^{\mathcal{C}}, P^{\mathcal{C}}, \mathcal{C}\rangle$ where $I^{\mathcal{C}}$ is a complete object-oriented graph, $P^{\mathcal{C}}$ is a finite set of object-oriented rules, and $\mathcal{C}$ is a class-model graph.*

Graph $I^{\mathcal{C}}$ portrays the initial system configuration. The object-oriented system specified by an object-oriented graph grammar evolves through the repeated applications of its productions, which are also typed over the class-model graph $\mathcal{C}$.

3.4 Observation semantics

Graph grammars are equipped with an usual (sequential) semantics given as the set of all possible derivations beginning in the initial graph. Formally, let $GG = \langle I, P \rangle$ be a graph grammar. A *derivation* of (an algebraic) GG is given by any (finite or infinite) sequence of direct derivations

$$\begin{tikzcd}
 & L_1 \arrow[r, "r_1"] \arrow[dl, "m_1"'] & R_1 \arrow[dr, "m'_1"] & & L_2 \arrow[r, "r_2"] \arrow[dl, "m_2"'] & R_2 \arrow[dr, "m'_2"] & & \cdots \arrow[dl] \\
I \arrow[rrr, "r'_1"] & & & G_1 \arrow[rrr, "r'_2"] & & & G_2 \arrow[r, "r'_4"] & \cdots
\end{tikzcd}$$

where I is the initial graph, $r_1, r_2, \dots$ are all elements of the production set P, and $m_1, m_2, \dots$ are matches (as defined in the grammar) (RIBEIRO, 1996), (HABEL, 1992), (CORRADINI et al., 1994).

The usual semantics is very concrete. A single derivation holds information about all the graphs in it, all rules applied, and all morphisms belonging to each direct derivation (including the counterparts morphisms r'_i and m'_i for rule r_i and match m_i). The set of derivations holds information concerning all derivations.

Object-oriented graph grammars intend to model object-oriented systems. Therefore, the usual semantics would be inadequate in many aspects. The most notable one concerns information provided by matches. A match is a total mapping from the rule left-hand side to a graph representing a system state. Information concerning attributes is stored, while it should be not, because object attributes are, in principle, invisible. Notice that they are relevant to system computations, since their value determine (at least most of the times) how an object responds to a message, and even if a rule can or cannot be applied. However important, they are not visible as entities, they belong to an object internal state which cannot be accessed by other system entities.

The semantics of a single object can be defined separately from the others (in that case, formal definitions of interfaces and context — environment — behaviour are necessary, since an object does not operate alone), and the behaviour of the whole environment (system) can be defined as the composition of the semantics of each part (objects). Using that approach, the use of attributes is necessary, because most properties of object behaviour would be expected to be expressed in terms of attribute values.

An object-oriented system semantics can also be defined in terms of the whole system, and then we can abstract away the elements we are not interested in. We will use that approach, because we are mainly interested in verifying properties about object-oriented systems and their behaviours, not about particularities of program execution or state structure. Additionally, model checkers have their performance degraded, or even made unfeasible, if we use a large amount of information to represent program executions or states. The object-oriented paradigm is about independence of implementation, so how an object is implemented (i.e., what exactly its attributes are) should not matter. We will present a more abstract semantics for object-oriented graph grammars based on *observations.* This semantics holds information about events happening in a system (message exchange among objects), and forgets about system structure. Therefore, although we are not able to express properties based on object states, we are still allowed to investigate properties of objects based on how they respond to the rules applied to them.

The verification method we propose in Chapter 4 is based on events. We consider an event to be a rule application. Events are observable, as well as the elements (identities of objects) they act upon. Object-oriented rules have a very specific format: a rule application corresponds exactly to a method call[2] in usual object-oriented programming. Therefore, we will use this fact to relate rule applications with the objects targeted by the rule match, and we end this section by presenting an observation semantics for object-oriented grammars. The following definitions are needed to construct it.

Given any object-oriented graph grammar, it is possible to present a uniquely defined partition on its set of productions, where each partition set contains the collection of rules that can be applied to a particular type of object.

Definition 3.13 (Partition on productions indexed by vertices) *Let $\mathcal{G} = \langle I^{\mathcal{C}}, P^{\mathcal{C}}, \mathcal{C}\rangle$ be an object-oriented graph grammar. The partition induced by the classes of the class-model graph $\mathcal{C}$ on the production set $P^{\mathcal{C}}$ is defined as:*

$$\Pi^V_{\mathcal{G}} = \{\Pi^V_v \mid v \in V_{\mathcal{C}}\}$$

where a production $p \in P^{\mathcal{C}}$ is in Π^V_v if and only if the attribute vertex (Definition 3.6) of p is a node a and $t(a) = v$.

It is easy to see that $\Pi^V_{\mathcal{G}}$ is indeed a partition: any production p has exactly one attribute vertex, which is mapped via the typing morphism to exactly one vertex of the class-model graph $\mathcal{C}$ (single inheritance). So, the intersection of any two sets $\Pi^V_{v_i}$ and $\Pi^V_{v_j}$ with $v_i \neq v_j$ is empty. By definition, we have that $\bigcup \Pi^V_v = P^{\mathcal{C}}$. The last requirement for $\Pi^V_{\mathcal{G}}$ to be a partition is that each $\Pi^V_v \in \Pi^V_{\mathcal{G}}$, must be nonempty. This cannot be guaranteed in general, but an empty Π_v means that there is an object of type v which cannot handle any message. Since the only way we can gather information or alter the state of an object is by sending messages to it, then any object of type v cannot be accessed by its environment. Furthermore, v would never send any message, since a message sent is always (according to the rule format) a response to a received message. Therefore, such a "silent" object would have no direct influence in the system computations, so it can be dismissed from any analysis. Notice that even if there is an object possessing an attribute of type v, by the principle of information hiding it would be invisible to the rest of the system. Hence, we can safely consider $\Pi^V_{\mathcal{G}}$ a partition.

The same applies to productions with a particular type of message edges:

Definition 3.14 (Partition on productions indexed by messages) *Let $\mathcal{G} = \langle I^{\mathcal{C}}, P^{\mathcal{C}}, \mathcal{C}\rangle$ be an object-oriented graph grammar. The partition induced by messages of the class-model graph $\mathcal{C}$ on the production set $P^{\mathcal{C}}$ is defined as:*

[2]Although methods in object-oriented programming languages which obey a sequential flow of execution usually have to be translated into a number of grammar rules, we have been using the terms "method" and "message" as synonyms, because the declarative programming style used in graph programming makes such distinction blurred.

$$\Pi^E_{\mathcal{G}} = \{\Pi^E_e \mid e \in E_{\mathcal{C}}|_{\mathrm{msg}}\}$$

where a production $p \in P^{\mathcal{C}}$ is in Π^E_e if and only if the left-hand side message (Definition 3.6) of p is an edge a and $t(a) = e$.

Again, it can be shown that $\Pi^E_{\mathcal{G}}$ is a partition. Since there is exactly one message in the left-hand side of any production $p \in P^{\mathcal{C}}$, and any message is uniquely typed over a single element in the class-model graph, we have that $\Pi^E_e \cap \Pi^E_{e'} = \emptyset$ if $e \neq e'$. By the same reasoning, $\bigcup \Pi^E_e = P^{\mathcal{C}}$. However, there might be a message m on the class-model graph for which there is no production. This means that such method is not implemented. Again, if this is the case, it will have no impact on the system behaviour as a whole, because those messages (if they exist in the initial graph or if they are produced by other rules), will never be consumed. This is a very unusual situation, which can easily be transformed into an equivalent one, where those messages, if produced, are consumed without modifying anything else in the system. This can be achieved through the addition of rules which have only the left-hand side message and the attribute vertex on their left-hand side, and the attribute vertex on the right-hand side. This way, the only effect this rule application will have is the message consumption, without transforming anything else. Therefore, we can consider $\Pi^E_{\mathcal{G}}$ a partition.

Notice that partitions $\Pi^V_{\mathcal{G}}$ and $\Pi^E_{\mathcal{G}}$ can be defined in terms of each other, since they are related by their messages and message targets. The point of having both of them defined is a matter of convenience: sometimes we are interested in all productions defined at some class level, while in other times we could be interested in all productions for the same kind of message.

The behaviour of an object depends, however, on the choice of how it makes use of its inherited elements. Traditionally, we have two different approaches: semantic or syntactic inheritance (MONTEIRO; PORTO, 1991). Semantic inheritance means that the behaviour of two objects u and v — with $u \sqsubseteq v$ — must be computed separately and then combined, while in syntactic inheritance the syntax of u and v are first combined, and then the behaviour of u is computed. Syntactic inheritance assures that the inherited methods from v can be used by u to perform its own computations; since the object-oriented graph morphisms make it possible (a class can always use its inherited methods to build its own), we will use this type of inheritance when defining an object behaviour, which is usually the kind of inheritance existing in object-oriented programming languages.

An object is then allowed to make use of its inherited methods to perform its own computations. Therefore, an object behaviour must be defined in terms of all the messages it can respond to. However, redefined methods hide their ancestors, so they must be dismissed from the behaviour definition. A *class set of rules*, as defined below, computes all messages that could be applied to an object belonging to a class. Namely, it selects all inherited messages minus the ones that have been redefined by some other method also in that set.

Definition 3.15 (Class set of rules) *Let $\mathcal{G} = \langle I^{\mathcal{C}}, P^{\mathcal{C}}, \mathcal{C}\rangle$ be an object-oriented graph grammar, $\Pi^V_{\mathcal{G}} = \{\Pi_v \mid v \in V_{\mathcal{C}}\}$ be the partition on the production set $P^{\mathcal{C}}$ induced by vertices (Definition 3.13), and $\Pi^E_{\mathcal{G}} = \{\Pi_m \mid m \in E_{\mathcal{C}}|_{\mathrm{msg}}\}$ be the partition on the production set $P^{\mathcal{C}}$ induced by messages (Definition 3.14).*

The class set of rules*, for each $v \in V_{\mathcal{C}}$, is defined as*

$$\mathcal{R}_v = \bigcup_{v \sqsubseteq^*_{V_{\mathcal{C}}} w} \Pi^V_w \setminus \bigcup_{\substack{m \sqsubseteq^+_{E_{\mathcal{C}}} m' \\ v \sqsubseteq^+_{V_{\mathcal{C}}} tar(m') \\ v \sqsubseteq^*_{V_{\mathcal{C}}} tar(m)}} \Pi^E_{m'}$$

In object-oriented programming the principle known as information hiding makes any object private information unavailable to the whole state of the system it belongs to. Moreover, the behaviour of an object cannot rely on the internal states of other objects in the system. For instance, if one wants to analyze how an object of type *queue* behaves, it should not matter how the object is implemented, it only matters how it responds to the messages addressed to it. Therefore, we can hide the existence of objects belonging to the internal state of another object in order to abstract away concrete implementations. This can be achieved by restricting the visibility of elements within a particular system.

Definition 3.16 (Visible types) *Let $\mathcal{C} = \langle V_{\sqsubseteq}, E_{\sqsubseteq}, L, src, tar, lab \rangle$ be a class-model graph. The* visible types *of $\mathcal{C}$ is any subgraph $\mathcal{C}_O = \langle V_{O\sqsubseteq}, E_{O\sqsubseteq}, L, src_O, tar_O, lab_O \rangle$ such that $V_O \subseteq V$, $E_O \subseteq E|_{\text{msg}}$, $src_O = src|_{E_O}$, and $tar_O = tar|_{E_O}$.*

The visible types of a system are the ones we are interested in analyzing. Since attributes are hidden from everything but the object possessing them, they often do not need to be part of the visible entities of an object-oriented system (although they could, if necessary) so they are not allowed to be visible.

In concurrent object-oriented programming there is no global state containing values of global variables to be used by entities to communicate. Communication is commonly achieved by (synchronous or asynchronous) message passing. The collection of messages to be executed is, in a sense, a substitute for a global state. The notion of observation, given below, defines a system observation as part of a system state.

Definition 3.17 (Observation) *Let $\mathcal{C}$ be a class-model graph, and $\mathcal{C}_O \hookrightarrow \mathcal{C}$ be a subgraph of visible types. Let $G^{\mathcal{C}} = \langle G, t_G, \mathcal{C} \rangle$ be an object-oriented graph. The observation of $G^{\mathcal{C}}$ with respect to $\mathcal{C}_O$ is the object-oriented graph $O^{\mathcal{C}}_G = \langle O_G, t_{O_G}, \mathcal{C} \rangle$ where*

- *$O^{\mathcal{C}}_G$ is the largest subgraph of $G^{\mathcal{C}}$ containing the elements x such that $t_G(x) \in \mathcal{C}_O$;*
- *$t_{O_G} : O^{\mathcal{C}}_G \to \mathcal{C}$ is the restriction of t_G respecting the elements in $O^{\mathcal{C}}_G$.*

An observation is defined as the collection of all visible messages, together with their parameters and targets, belonging to an object-oriented graph. If this graph is a system state, then an observation consists of all visible messages targeting at visible objects, and their respective parameters, waiting to be processed within the object-oriented system.

Message execution for different objects can always be performed concurrently (if the corresponding matches exist), since there are no possible interference between them. This non interference is caused by the object-oriented rule structure, which prevents a message addressed to an object a to have its implementation making use of internal elements (attributes) of another object b (unless indirectly, if object deletion is allowed, because in that case rule application can present the side-effect of erasing attributes connected to the deleted object). Therefore, all computations occurring with distinct objects will always be parallel independent (EHRIG et al., 1996).

In imperative sequential programming, the *state* of a system is usually defined as the value of all its constituents (usually variables) at a given moment in time (within the time boundaries of the computation process). This notion matches the definition of the state of a program executing in a machine, which corresponds to the contents of the memory at a given instant (WATT, 1991). Since any information from an object can only be obtained through its methods, then its behaviour must also be defined in such terms. Internal changes within an object should not be relevant to whole system, since they cannot be directly observed or retrieved (because they are not visible). This notion of "observation" cannot, for that reason, rely on any global memory structure, as in the imperative programming framework.

How a system state changes when an action is performed can be described as a pair of states indicating the previous and the following states concerning the action execution. This pair of states is called a system *transition*, usually labeled by the performed action. A system computation can then be defined as any finite or infinite sequence of states, where each state is obtained from the previous one by some legal system transition.

Definition 3.18 (Observable transition relation $\longrightarrow$) *Let $\mathcal{G} = \langle I^{\mathcal{C}}, P^{\mathcal{C}}, \mathcal{C} \rangle$ be an object-oriented graph grammar, and $\mathcal{C}_O \hookrightarrow \mathcal{C}$ be its subgraph of visible types. Let $G^{\mathcal{C}}$ and $H^{\mathcal{C}}$ be two object-oriented graphs, and $O_G^{\mathcal{C}} \hookrightarrow G^{\mathcal{C}}$ be the observation of $G^{\mathcal{C}}$ with respect to $\mathcal{C}_O$. We say that $G^{\mathcal{C}}$ and $H^{\mathcal{C}}$ are related under $\longrightarrow$, if there is an object-oriented graph production $r : L^{\mathcal{C}} \to R^{\mathcal{C}} \in P^{\mathcal{C}}$ (with left-hand side message e and attribute vertex o), an object-oriented match $m : L^{\mathcal{C}} \to G^{\mathcal{C}}$ such that $G^{\mathcal{C}} \overset{r,m}{\Rightarrow} H^{\mathcal{C}}$. In that case, we say that*

- *$(G^{\mathcal{C}}, \langle r, o \rangle, H^{\mathcal{C}}) \in \longrightarrow$, or $G^{\mathcal{C}} \xrightarrow{\langle r,o \rangle} H^{\mathcal{C}}$, if $m_E(e) \in O_G^{\mathcal{C}}$ and $m_V(o) \in O_G^{\mathcal{C}}$.*
- *$(G^{\mathcal{C}}, \tau, H^{\mathcal{C}}) \in \longrightarrow$, or $G^{\mathcal{C}} \xrightarrow{\tau} H^{\mathcal{C}}$ if either $m_E(e) \notin O_G^{\mathcal{C}}$ or $m_V(o) \notin O_G^{\mathcal{C}}$.*

Definition 3.18 establishes a relation between object-oriented graphs based on the existence of a derivation from the first to the second, under the productions of a given object-oriented graph grammar $\mathcal{G}$. The relation is also labeled by the grammar production r applied, together with the object o to which the rule was applied to (i.e., the image of the production attribute vertex under the match morphism), if both the message m implemented by rule r and the object o are visible entities. If either of them is not visible, then the relation is labeled with symbol τ (which is the symbol used to represent "silent" transitions in many process calculi like CCS (MILNER, 1989a)).

Definition 3.19 (Object-oriented graph grammar transition semantics) *Let $\mathcal{G} = \langle I^{\mathcal{C}}, P^{\mathcal{C}}, \mathcal{C} \rangle$ be an object-oriented graph grammar, and $\mathcal{C}_O \hookrightarrow \mathcal{C}$ be its subgraph of visible types. The transition semantics of $\mathcal{G}$ is given by the labeled transition system $\mathcal{T}^{\mathcal{G}} = \langle S, s_0, L, \to \rangle$, where the set of* *states* $S = \{G^{\mathcal{C}} \mid I^{\mathcal{C}} \Rightarrow^* G^{\mathcal{C}}\}$*, the initial state* $s_0 = I^{\mathcal{C}}$*, the set of labels* $L = \{\tau\} \cup \{\langle p, o\rangle \in P^{\mathcal{C}} \times V_G \mid G \in S^{\mathcal{T}} \wedge t_G(o) \in \mathcal{C}_O \wedge p \in \Pi^E_{t_G(o)}\}$*, and the transition relation* $\rightarrow$ *is the observable transition relation* $\longrightarrow$ *(Definition 3.18).*

3.5 Summary

This chapter presented the concepts to establish the model of computation provided by object-oriented graph grammars. First, $\mathcal{C}$-typed graphs and their morphisms are defined. $\mathcal{C}$-typed graphs are hypergraphs typed over a class-model graph, but the typing morphism is more flexible than the traditional one, in the sense that mapped hyperedges need to preserve *relations* between sources and targets. This feature adequately models inheritance, for any object can make use of inherited attributes or messages.

$\mathcal{C}$-typed graphs and their morphisms form a category, called within this text **CGraphP**($\mathcal{C}$). Next, object-oriented graphs are presented. Object-oriented graphs are restricted $\mathcal{C}$-typed graphs, in the sense that messages addressed to objects must be *correctly typed*, i.e., an object only receives inherited messages which are at the bottom of the redefinition chain it belongs to. This implements the hiding of elements which had been overridden in some derived class. Category **OOGraphP**($\mathcal{C}$) has object-oriented graphs as objects and $\mathcal{C}$-typed graph morphisms as arrows. Most results of the single-pushout approach to graph grammars are given in terms of categorical constructs, hence it is necessary to build suitable categories of (special) graphs and graph morphisms in order to benefit from the theory already developed.

The model of computation provided by object-oriented programs is Turing complete. Nevertheless, there are some inherent restrictions related to the encapsulation of data and functions within a single syntactic structure (i.e., a class). Object-oriented rules respect those restrictions: an object can only access or modify its own attributes, and each method implementation corresponds to a response to a received message. Additionally, following the semantics of rule application, for each attribute deleted another one (of the same type) must be created.

Object-oriented rules can be of four types: *basic*, *strict*, *with object creation* and *general*. Basic object-oriented rules implement the constraints related to the paradigm, and form the basis from each other type of object-oriented rule is built. Strict object-oriented rules do not allow new objects to be created, while object-oriented rules with object creation do, making sure that an object is created with all its attributes (inherited or not). General object-oriented rules, on the other side, let objects to be deleted, although an object can only delete itself, as required by object-oriented principles.

Matches are defined as total $\mathcal{C}$-typed graph morphisms, where preserved and deleted elements cannot be identified by the referred morphism (identification condition). It means that a match does not need to be identical on object types, it is only required that it preserves the underlying inheritance relation. The consequence is that every time a rule is defined for an object of type x, it can be applied in any object of type y, as long as y is a class derived from x. Thus, object-oriented matches implement the concepts of polymorphism and method inheritance among objects.

Direct derivations (or rule application) are then defined. It is shown that a direct derivation is a pushout in the category **OOGraphP**($\mathcal{C}$). This is the most important result of this chapter, in the sense that direct derivations in the single-pushout approach are defined as a pushout of two arrows (namely, a rule and a match) in a suitable category of graphs and graph morphisms. Therefore, all results having to do with derivations being pushouts can be used within this framework.

Closure properties on derivations are then stated and proven: complete object-oriented graphs are closed under direct derivations with strict and object creation rules, but they are not closed under general object-oriented rules. This happens because a deletion of an object causes that all incident edges to that object are deleted as well. It can be useful to model deletion in unknown contexts, which is very common when a link to something becomes useless when the link target is deleted.

Finally, object-oriented graph grammars are presented. Object-oriented graph grammars are ordinary graph grammars where the productions are object-oriented rules, and the initial graph is an object-oriented graph. The semantics of (observable) computations is presented next.

The *visible types* of a system are defined as any subgraph of the underlying class-model graph, restricted to nodes and message edges. Therefore, no attribute can be visible, but entire classes or just some messages can be made invisible. Notice that this visibility does not have anything to do with computations: the productions consuming invisible messages can still be applied, even into invisible objects. The only difference is that they will not belong to the *observation* of a state, which is defined as the largest subgraph of the graph state with respect to its visible types.

Using the notion of observation, a (labeled) transition relation on graphs is defined. This transition relation represents the existence of a derivation between two graphs. It is labeled with a pair of information consisting of the production applied together with the object the production was applied to (because of their structure, object-oriented rules model method invocation of objects in a system), provided that both the object and the message the rule implements are visible entities. If at least one of them is invisible, the transition is labeled with the symbol τ, to indicate that this transition label is itself invisible.

The transition system semantics for an object-oriented graph grammar is then the transition system generated by the transition relation previously described, using the grammar initial graph as the transition system initial state. Transitions are labeled by pairs $\langle p, o \rangle$, with o being a node in the graph $G^{\mathcal{C}}$ which is the source of the transition, and p is a production belonging to the class set of rules of o, such that there is an object-oriented match from the left-hand side of p to $G^{\mathcal{C}}$. A class set of rules has a straightforward meaning, and represent all messages an object can receive, which are all messages applicable to the class which types the object, together with all inherited messages from its primitive classes minus the messages which had been overridden.

4 OBJECT-ORIENTED VERIFICATION

4.1 Introduction

The primary advantage of having a formally specified system is to reason about it using available analysis methods. Concurrent systems validation is regarded as a hard problem, mainly because errors can be caused by the *order* in which processes are executed (there is an exponential number of ways in which program executions may interleave — i.e., it is an NP-complete problem at best, when program size is used as the measure). Therefore, in order to assure that a system presents only desirable behaviours (or that it does not present any undesirable ones), formal verification techniques should be incorporated into the software development process.

Model checking, as explained in Section 1.4, is perhaps the most successful technique to prove the presence (or absence) of properties in systems whose behaviour is described as a finite transition system. This success is due to the fact that it is a fully automatic process which outputs a yes/no response to a query in the form of a property; moreover, if the property is not true in the system, the model checker also produces a counterexample, showing a program execution trace which led to the violation of the specified property. Hence, it can also help in fixing problems in specifications or in actual programs.

There are two different approaches to model check formally described systems: the first one is to build a model checker to generate and verify models written in the specification language, and the other one is to make use of existing model checkers by translating the language used to describe the system to the chosen model checker input language. Of course, the second approach is usually simpler. Naturally, the translation must be correct, in the sense that the behaviour (semantics) of any system must be preserved by the translation, at least concerning the properties being checked. The development of (correct) model checkers is a task which requires a large amount of time and effort, mainly because good model checkers not only perform model generation and property verification: techniques such as abstraction (DWYER et al., 2001), (SCHMIDT, 2002) program slicing (HATCLIFF; DWYER; ZHENG, 2000), (KRINKE, 2003) and partial order reductions (VISSER et al., 2003) are necessary to verify systems with a large state space.

SPIN (Simple Promela INterpreter) (HOLZMANN, 1997) is the model checking tool chosen for analyzing the logical consistency of a concurrent object-oriented program written as an object-oriented graph grammar. The input language of SPIN is Promela (PROtocol/PROcess MEta LAnguage) which is a specification language to model state transition systems. Promela allows for the dynamic creation of concurrent processes, which can communicate via synchronous (i.e. rendezvous), or asynchronous (i.e. buffered) message channels or via global variables. Promela has a (programming language) C-like syntax (KERNINGHAN; RITCHIE, 1986) and constructs for sending/receiving messages are similar to those of CSP (HOARE, 1985). Non determinism is modeled using conditions and repetition structures clearly inspired by Dijkstra's guarded commands (DIJKSTRA, 1975).

This chapter is structured as follows: the formal translation from an object-oriented graph grammar specification to a Promela program is given in Section 4.2. Compatibility between the semantics of an object-oriented graph grammar and its translated Promela program computations is argued in Section 4.3. Section 4.4 shows an example of this method to verify a program written as an object-oriented graph grammar. The example is a classic problem in the theory of concurrency, known as the *Dining Philosophers* problem. We present an attempt to model it, which has the possibility of deadlock, and we present how the translation is done, and how the error is discovered by model checking the resulting program. Final remarks are presented in Section 4.5.

4.2 The translation algorithm

We will use SPIN (Simple Promela INterpreter) (HOLZMANN, 1997) to analyze the logical consistency of object-oriented systems specified as object-oriented graph grammars. Therefore, we must provide a semantically sound translation from object-oriented graph grammar specifications to Promela programs.

The translation algorithm presented here is both a restriction and an extension of the one in (DOTTI et al., 2003) for object-based graph grammars. It is a restriction because we do not deal with (algebraic/numerical) values for attributes. It is an extension, however, since object-oriented features are translated, to assure that the semantics of execution of the generated Promela code is compatible with the computations of the source object-oriented graph grammar.

Objects will be modeled as Promela processes. The semantics of Promela computations follow an interleaving approach, so the object-oriented grammar (sequential) transition semantics defined in Section 3.4 is adequate to be compared to the translated one. Message exchange between objects will be modeled through asynchronous communication channels.

A major contribution of this work is the encoding of inheritance, polymorphism, and dynamic binding in Promela. Promela originally does not have any object-oriented features. Our approach is somewhat different from classic object-oriented programming languages implementations (MACLENNAN, 1999), (WATT, 1990), (GHEZZI; JAZAYERI, 1998), (PRATT; ZELKOWITZ, 1996), where a virtual table determines which method should be called in execution time. Our method dispatch mechanism uses a little computational reflection (SMITH, 1982), in the sense that each object (process) is aware of its own type, and that information is made available to other entities when they have access to the object (as an attribute, or as a message parameter). So an object can decide, in run time, the adequate message to send based on the actual type of the message receiver.

Dynamic binding implementations require that inheritance and overriding relations are explicit within the model. Therefore, they must appear in the translation. Inheritance appears in the form of a global array which can be accessed by any program element. Dynamic binding is implemented by the rule application procedure within each object process definition. Because both inheritance and overriding relations remain static throughout the computation, even large hierarchies should not increase the program state space size, making model checking in SPIN[1] still feasible.

As most model checkers, SPIN performs model-based verification, which means that properties can only be defined over states, and not over transitions. We use a set of global variables to allow verification over events, as well as over states. An object-oriented system described as a complete object-oriented graph may possess multiple instances of the elements belonging to the underlying class-model graph. Frequently, one is interested in knowing if all elements having a particular type behave properly, or if an interaction between two (or more) elements is adequate. Generally, asking questions about system elements requires that all elements are somehow identified. The translated program has one global variable for each class belonging to the class-model graph over which the grammar is typed, to identify the last object of that type that had a production applied to it (provided that both the object type and the message consumed are visible entities). If a message is received by an object (now belonging to the system graph), and consumed by some rule application, then the identity of this object is assigned to the respective variable. Additionally, there is a global variable to identify which *rule* was applied, and it is updated every time such action occurs. Notice that rule application is not equivalent to message consumption. Although each rule application corresponds exactly to a response to a received message, there can be multiple (different) rules implementing actions for the same type of message. This variable is necessary if one is interested in verify (through temporal logics properties) possible orders in which rules can be applied.

Another issue in the translation is to ensure that the semantics of the generated Promela code is compatible with the semantics of object-oriented graph grammars. Buffered channels in Promela, which model the reception of messages in the translation, have a *first come, first served* (FIFO) policy. This means that messages are received in the exact same order they are sent. FIFO semantics for the reception of messages conflicts with the object-oriented grammar semantics presented in Section 3.4. The solution to this problem is reused from (DOTTI et al., 2003): messages, when received, are atomically stored in an array called *object process buffer*. Then, the message is taken from this array and stored in a local object array; the choice made on which local array slot to retrieve a received message is non deterministic. The local array can be seen as a way to shuffle received messages, so they do not necessarily be processed in the exact order of reception.

Let $\mathcal{G} = \langle I^{\mathcal{C}}, P^{\mathcal{C}}, \mathcal{C} \rangle$ be an object-oriented graph grammar, where $I^{\mathcal{C}} = \langle I, t_I, \mathcal{C} \rangle$ is the (complete object-oriented) initial graph, $P^{\mathcal{C}}$ is the set of (object-oriented strict[2]) graph grammar productions (Definition 3.7), $\mathcal{C} = \langle V_{\mathcal{C}_{\sqsubseteq}}, E_{\mathcal{C}_{\sqsubseteq}}, L, src_{\mathcal{C}}, tar_{\mathcal{C}}, lab_{\mathcal{C}} \rangle$ is the underlying class-model graph, $\mathcal{C}_O \hookrightarrow \mathcal{C}$ be the visible types of $\mathcal{C}$, and $\mathcal{R}_v$ is the class v set of rules (Definition 3.15), for each $v \in V_{\mathcal{C}}$. Let us assume that, for any class

[1]SPIN does not generate *all* computation paths, only those actually reachable by the program.

[2]The translation procedure assumes that objects are not being created or deleted along the computation. It is a restriction of the possible actual object-oriented graph grammar productions, but the choice of doing the translation for only object-oriented strict rules is twofold: (i) unbounded creation of objects restricts the type of properties we can verify, since new objects cannot be properly identified before their creation, and temporal logic properties must be defined in terms of object identification, and (ii) the creation and deletion of objects are straightforward to translate. Therefore, this part of the translation is left out for future work.

$v \in V_{\mathcal{C}}$, (arbitrary) total orders exist for the sets generated by the *extended attribute set function* and *extended message set function* (Definition 3.3). These total orders are needed to make the translation to Promela correct, since the names used in the grammar should not have impact on the meaning (semantics) of the system being defined; it is therefore easier to perform the translation over the relative order of all attributes and messages belonging to a class (being them inherited or not). Notice, however, that those orders can be any permutation of the set elements, and therefore they have no special meaning in the scope of the formalism proposed (i.e., there is no loss of generality in having those total orders defined). Let then, in the following translation, $m_k^v \in msg_{\mathcal{C}}^*(v)$ denote the k-th message that could be received by object $v \in V_{\mathcal{C}}$, $k = 1, \ldots, |msg_{\mathcal{C}}^*(v)|$, and $a_k^v \in attr_{\mathcal{C}}^*(v)$ denote the k-th attribute belonging to object v, with $k = 1, \ldots, |attr_{\mathcal{C}}^*(v)|$.

Let Λ_{OOGG} be the class of all object-oriented graph grammars, and $\Lambda_{Promela}$ be the collection of all syntactically correct Promela programs (the syntax description of Promela is given in Section E.1). The translation function $\mathcal{F} : \Lambda_{OOGG} \times \mathbb{N} \to \Lambda_{Promela}$ is defined as:

$$\mathcal{F}(\mathcal{G},n) = [\mathsf{header}] \cdot [\mathsf{procdefs}] \cdot [\mathsf{init}] \tag{4.1}$$

where the transformational grammar rules for the nonterminal symbols [header], [procdefs], and [init] are defined, respectively, in Figures 4.1, 4.2, and 4.4. Symbols [header], [procdefs], and [init] generate, respectively, the declaration of types and auxiliary macros in the Promela program; the definition of (asynchronous) processes whose behaviour match the behaviour of each object-oriented graph grammar object (given by the set of rules defined for each of them); and the initialization of variables and actual processes, representing the grammar initial graph. The positive integer parameter n determines the translated program buffer sizes. Buffers have two functions on the Promela program: the first one is to store the messages received by the objects (translated as processes), and the second one is to mimic the semantics of object-oriented graph grammars rule application, as previously expounded. It is a parameter of the translation process because adequate buffer sizes depend on the characteristics of the application at hand. If the buffers are not large enough, they will be filled up during the verification process, and an error will occur. It is actually an empirical parameter, and if such error is raised during the verification process the translation must be rebuilt with a larger value for n. In the following, the translation will be informally explained, through all nonterminal symbols existent within the transformational grammar.

The nonterminal symbol [header] generates all constants, types, and global variables used along the Promela program. The first constant defined is `BSIZE`, whose value is the buffer size passed as a translation parameter, which determines the size of the buffers used to manipulate messages. The constant `SIZE_INHERITANCE` has a straightforward meaning, as it defines the size of the set containing the transitive closure of the inheritance relation. All pairs of this relation will be stored in an array (of type `extends`, defined later on), to allow polymorphism to be applied along the computation.

The `mtype` keyword defines an enumeration of names that can be used along the program. Multiple `mtype` declarations have the same effect as a single one, having as elements the (set) union of all `mtype` declarations. We use multiple declarations just to make the translation more readable. `mtype`s are used to define names of

```
[header]
  #define BSIZE ·$n$·
  #define SIZE_INHERITANCE ·| $\sqsubseteq^{+}_{V_{\mathcal{C}}}$ |·
   mtype = {·class_·$v_1$·,·class_·$v_2$·,·...·,·class_·$v_{|V_{\mathcal{C}}|}$·};·                        $v_i \in V_{\mathcal{C}}$
   mtype = {·msg_·$v_{11}$·_·$e_1$·,·msg_·$v_{12}$·_·$e_2$·,·...·,·                                  } $e_i \in E_{\mathcal{C}}|_{\text{msg}}$,
   msg_·$v_{21}$·_·$e_2$·,·msg_·$v_{22}$·_·$e_2$·,·...·,·                                             } $\forall v_{ik} \in V_{\mathcal{C}} : e_i \in msg^{*}_{\mathcal{C}}(v_{ik})$
    msg_·$v_{|E_{\mathcal{C}}|_{\text{msg}}|1}$·_·$e_{|E_{\mathcal{C}}|_{\text{msg}}|}$·};· msg_·$v_{|E_{\mathcal{C}}|_{\text{msg}}|2}$·_·$e_{|E_{\mathcal{C}}|_{\text{msg}}|}$·};·  }
   mtype = {·rule_·$v_{11}$·_·$p_1$·,·rule_·$v_{12}$·_·$p_1$·,·...·,·                                } $p_i \in P^{\mathcal{C}}$,
   ·rule_·$v_{21}$·_·$p_2$·,·rule_·$v_{22}$·_·$p_2$·,·...·,·                                          } $\forall v_{ik} \in V_{\mathcal{C}} : p_i \in \mathcal{R}_{v_{ik}}$
    rule_·$v_{|P^{\mathcal{C}}|1}$·_·$p_{|P^{\mathcal{C}}|}$ rule_·$v_{|P^{\mathcal{C}}|2}$·_·$p_{|P^{\mathcal{C}}|}$·};·  }
   mtype = {·$o_1$·,·$o_2$·,·...·,·$o_{|V_I|}$·};·                                                    $o_i \in V_I$
  typedef object{chan channel; mtype type; mtype id;};·
  typedef extends{mtype primitive; mtype derived;};·
  extends inheritance [SIZE_INHERITANCE];·
  mtype event_RuleName;·
  mtype event_·$v$·;·                                                                                   $\forall v \in V_{\mathcal{C}_O}$
  inline match (received, shouldbe, ok, i) {·
  i = 0; ok = false;·
  if ::  (received == shouldbe) -> ok = true;·
  ::  else -> do ::  (i < SIZE_INHERITANCE) ->·
  if ::  (inheritance[i].primitive==shouldbe) && ·
  (inheritance[i].derived==received) -> ok = true; break; fi; i++;·
  ::  else -> break; od; fi; }·
```

Figure 4.1: Promela translated program header definition.

classes, *messages*, *rules*, and *objects* (classes instances) existing within the program defined by the object-oriented graph grammar. Class names are utilized to type the objects in the program, message names to identify messages received, rule names to identify rule application, and object names to verify properties of specific objects within the system. `typedef` is the type constructor in Promela, and through it the structured type `object` is defined to encapsulate all information needed to manipulate an object: its channel (of type `chan`), which is the way processes communicate in Promela; its type (taken from the classes `mtype` declaration); and its identification (taken from the objects `mtype` declaration). Another type is defined in the same manner: `extends` define a pair of `mtype` values, intending to represent pairs belonging to the inheritance relation. An array of those pairs, called `inheritance` is then defined, and it will be initialized in the main Promela process using the transitive closure of the relation on graph nodes.

The verification of properties can be done in two levels: object verification (to verify object states) and rule verification (to verify when a rule has been applied). Model checkers such as SPIN generally do not allow for the verification of events. If we are interested in properties of computations, we must embody events into program states. This is achieved through global variables `event_RuleName`, and `event_`v, for all v in the class-model graph set of visible nodes. The first variable holds the name of the last grammar production applied, while the other variables hold the identification of the object of type v which was the target of the aforementioned production. Notice that there is one of such variable for each class definition.

The inline macro `match` is an auxiliary function to assist in the left-hand side rule matching procedure. It receives as parameters the type of an actual object, and the type this same object should have for a match to exist. The procedure simply checks if both types are related by the (reflexive and transitive closure of the) inheritance relation. This is done by searching the `inheritance` array for a matching pair. If the pair exists (i.e., either the actual and the intended type are the same or they are related by inheritance), the variable ok (passed as parameter) is set to true, otherwise it is set to false.

The nonterminal symbol [procdefs] (Figure 4.2) generates the two declarations that completely describe an object behaviour in Promela. From each vertex v in the class-model graph $\mathcal{C}$ — which types grammar $\mathcal{G}$ — one macro and one process declaration are derived. Macro [procrules](v) (Figure 4.2) defines how rule applications will be performed in Promela. The translation of graph rules applications establish the existence of matches before the rule can be applied, and the choice of message consumption and production application is non deterministic, as required by the defined grammar semantics. Symbol [procdecl](v) generates the Promela process declaration for objects of type v.

Processes declarations are written with symbol [procdecl](v) (Figure 4.2), where the parameter v is a node from the object-oriented graph grammar underlying class-model graph. The header of [procdecl] for class v consists of a Promela reserved word `proctype`, which is the language processes constructor. The name of the process is the same as the class, and its parameters are (i) the actual object from this class (which is taken from the grammar initial object-oriented graph), and (ii) the list of the actual object attributes, following the order established in the beginning of the translation function. Notice that this procedure follows the way actual object-oriented programming languages are implemented: a method call for an object o is an ordinary procedure call with the object o passed as a parameter (SETHI, 1996). The installation of an actual process of type v can be seen as the creation of a variable typed as v, and the `proctype` header can be seen as a implementation of the object constructor.

Local variables for each process are the following: a `mtype` variable called `msg_Received`, to identify which message has been taken out from the buffer; an empty `object` called `nil_object`, used as an empty parameter to the messages sent by a rule application; a sequence of variables called `par_`v`_`m_k^v`_`$src_{\mathcal{C}}^i(m_k^v)$`_`$i$, to hold the values of (all) possible message parameters. Notice that for each one of the k messages object v can receive, there is a parameter (whose type is also indicated) for each one of the i sources the message hyperedge has in the underlying class-model graph; a boolean variable `match_p` for each of the productions that might be applied to an object of type v, together with an integer variable to be used as an `index` in each type testing during a match; a local buffer to hold messages received through the object channel; an array of boolean values to indicate whether a buffer positions is occupied or not; and a local array of boolean variables to indicate if there was an unsuccessful attempt to apply a rule for the message in the local array slot i. An unsuccessful attempt of rule application means that there is no match for that rule. This will remain true until another message is applied, an then those values are set to false again. Local variables do not increase the state space, so there is no problem in defining unused local variables (for instance, in a class that does not make use of `nil_object`, or whose rules do not need to check over attribute types).

The computation of a process is performed through an infinite loop that continuously tests (non deterministically) if either a new message has arrived at the object main channel — in which case the message is retrieved and placed in some empty slot of the local message buffer — or if there is a message in the local buffer waiting to be consumed — in the latter case, the message is atomically retrieved from the buffer and the production it refers to is applied, by calling macro `rules_`v. In case neither the object channel nor the local message buffer contain any messages

```
[procdefs]
    [procrules]$(v)\cdot$
    [procdecl]$(v)$                $\forall v \in V_{\mathcal{C}}$

[procrules]$(v \in V_{\mathcal{C}})$
  inline rules_$\cdot\, v \,\cdot$ () {$\cdot$
  if
    ::  (msg_Received == msg_$\cdot\, v \,\cdot$ _$\cdot\, m \,\cdot$ ) ->                          ⎫
    [testmatch]$(v, m, p)$                                          $p \in \Pi^E_m$    ⎪
    if$\cdot$                                                                          ⎪
      ::  match_$\cdot\, p \,\cdot$ ->                                              ⎫          ⎪
      [modifystate]$(o, m', p)\cdot$           $m' \in p_L|_{msg} \wedge o = tar_{p_L}(m')$  ⎪          ⎪
      d_step {$\cdot$                                                        ⎪          ⎪
       event_$\cdot\, x \,\cdot$  = opc_$\cdot\, v \,\cdot$ .id;        $\forall x\ v \sqsubseteq_{V_{\mathcal{C}}} x$, if $x \in V_{\mathcal{C}_O}$  ⎬ $p \in \Pi^E_m$ ⎬  $\forall m \in \mathcal{R}_v$
       event_RuleName = rule_$\cdot\, v \,\cdot$ _$\cdot\, p \,\cdot$ ;$\cdot$   if $m \in E_{\mathcal{C}_O}$          ⎪          ⎪
      }$\cdot$                                                              ⎪          ⎪
      [sendmsgs]$(m', p)\cdot$                 $m' \in p_L|_{msg}$             ⎪          ⎪
      inspected[i] = false;               $i = 1, \ldots, n$              ⎭          ⎪
    ::  else -> [nomatch]                                                           ⎪
    fi;$\cdot$                                                                          ⎪
  ::else -> [nomatch]$\cdot$                                                        ⎭
  fi;}

[procdecl]$(v)$
  proctype  $\cdot\, v \,\cdot$ (object opc_$\cdot\, v \,\cdot$ [classattrlist]$(v)$ $\cdot$ ) {$\cdot$
  mtype msg_Received;$\cdot$
  object nil_object;$\cdot$
  object match_tmp;$\cdot$
    bool match_$\cdot\, p \,\cdot$ ;$\cdot$    for each $p \in \Pi^V_v$
  int index;$\cdot$
  [classpardecl]$(v)\cdot$
  [classtmpattrdecl]$(v)\cdot$
  bool busy[BSIZE] = false;$\cdot$
  bool inspected[BSIZE] = false;$\cdot$
     chan opcb_$\cdot\, v \,\cdot$ [BSIZE] = $\cdot$
       [1] of {mtype $\cdot$ (,object)$^i$ $\cdot$ };$\cdot$      $i = \sum_{k=1}^{|msg^*_{\mathcal{C}}(v)|} \mid src_{\mathcal{C}}(m^v_k) \mid$
  RCV: opc_$\cdot\, v \,\cdot$ .channel?msg_Received $\cdot$ [classparlist]$(v)$ $\cdot$ ;
  opc_$\cdot\, v \,\cdot$ .channel!msg_Received $\cdot$ [classparlist]$(v)$ $\cdot$ ;
   do$\cdot$
   ::(len(opc_$\cdot\, v \,\cdot$ .channel) > 0) -> $\cdot$
     if$\cdot$                                                          ⎫
     ::  (busy[ $\cdot\, i \,\cdot$ ]==false) ->                                ⎪
     opc_$\cdot\, v \,\cdot$ .channel?msg_Received $\cdot$ [classparlist]$(v)$ $\cdot$ ;   ⎬  $0 \leqslant i < n$
     opcb_$\cdot\, v \,\cdot$ [ $\cdot\, i \,\cdot$ ] !msg_Received $\cdot$ [classparlist]$(v)$ $\cdot$ ;  ⎪
     busy[ $\cdot\, i \,\cdot$ ]=true;$\cdot$                                     ⎪
     ::else ->                                                  ⎭
   assert(false); (fi;)$^n\cdot$
     ::  busy[ $\cdot\, i \,\cdot$ ] == true && inspected[ $\cdot\, i \,\cdot$ ] == false -> atomic {  ⎫
     opcb_$\cdot\, v \,\cdot$ [ $\cdot\, i \,\cdot$ ]?msg_Received $\cdot$ [classparlist]$(v)$;                  ⎬  $0 \leqslant i < n$
     busy [ $\cdot\, i \,\cdot$ ] = false;                                               ⎪
     rules_$\cdot\, v \,\cdot$ (); }                                                     ⎭
   ::else -> goto RCV
   od; }

[nomatch]
    if$\cdot$                                                        ⎫
    ::  (busy[ $\cdot\, i \,\cdot$ ]==false) ->                              ⎪
    opcb_$\cdot\, v \,\cdot$ [ $\cdot\, i \,\cdot$ ] !msg_Received $\cdot$ [classparlist]$(v)$ $\cdot$ ;  ⎬  $0 \leqslant i < n$
    busy[ $\cdot\, i \,\cdot$ ]=true;$\cdot$ inspected[ $\cdot\, i \,\cdot$ ]=true;$\cdot$            ⎪
    ::else ->$\cdot$                                                 ⎭
  assert(false); (fi;)$^n$

[classpardecl]$(v)$

   $\Big($(object par_$\cdot\, v \,\cdot$ _$\cdot\, m^v_k \,\cdot$ _$\cdot\ t(src^i_{\mathcal{C}}(m^v_k))\ \cdot$ _$\cdot\, i \,\cdot$ ;)$_{i=1}^{|src_{\mathcal{C}}(m^v_k)|}\cdot\Big)_{k=1}^{|msg^*_{\mathcal{C}}(v)|}$

[classtmpattrdecl]$(v)$

   $\Big($(object tmp_attr_$\cdot\, v \,\cdot$ _$\cdot\ tar^i_{\mathcal{C}}(a^v_k)\ \cdot$ _$\cdot\, i \,\cdot$ _$\cdot\, k \,\cdot$ ;)$_{i=1}^{|tar_{\mathcal{C}}(a^v_k)|}\cdot\Big)_{k=1}^{|attr^*_{\mathcal{C}}(v)|}$
```

Figure 4.2: Promela translated program processes definitions.

to be consumed, the process will jump to the beginning of the main loop, and stay blocked there until a new message arrives.

The local buffer slot where the received messages are put on, or retrieved from, is chosen non deterministically, according to the semantics of the (guarded) Promela commands `if` and `do`. Therefore, there is no particular order in which the messages are consumed, as required by the object-oriented graph grammar semantics. Notice that, from the time a message is taken out of the local buffer to the time a rule application is completed — by either applying the rule or by putting the message back to the local buffer, if no match exist for that rule — no other process can interleave with that execution, because of the `atomic` keyword. The atomicity of the rule application process is necessary, to mimic the way rules are applied in the graph grammar, where the whole matching and application procedure is performed in a single step. Furthermore, if interleaving was allowed, errors could appear: if a process is stopped between finding a match for a production and the application of that production, meanwhile the state graph could be altered in a way that turns the rule application impossible; therefore a match/application procedure is considered a critical region of any object behaviour.

The declaration of [testmatch](v, m, p) (Figure 4.3) has the purpose of testing if a match for a production p (which consumes message m, which points to an element of type v) exists in the system graph. This matching procedure tests whether all attributes have the correct type, and if all attributes target at the right objects (message parameters, other attributes, or the message target object itself). To perform this action, the matching procedure makes use of a boolean variable `match_`p, defined within all processes that can receive the message consumed by production p. As parameters, [testmatch] gets the class v which can receive message m which is, in its turn, the left-hand side message of rule p. The first action is to call macro `match` (defined in Figure 4.1), to test if the type of all attributes present in the left-hand side of p are correct regarding the type they are supposed to have; if they are the same, or related by inheritance, the boolean variable `match_`p is set to true, otherwise it is set to false. Next, the values of attributes are tested. The term "values" here means the elements to which attribute arcs point to, not actual algebraic values (since we did not define attributes to be elements of some algebraic sort). An attribute appearing on the left-hand side of a rule can only point to another attribute value, to a message parameter, or to the attribute vertex itself, because those are the only vertices allowed to appear in the left-hand side of an object-oriented production (Definition 3.6). Therefore, for each attribute (of order k), if some of their targets are (respectively) the i-th parameter of the left-hand side message, another attribute (of order k') target, or the attribute vertex, the actual values of each of them are tested against the values they are supposed to have. Notice that if an attribute targets at an object which is neither a message parameter nor another attribute target or the main object, only its target *type* needs to be tested, which has already been done before. A conjunction of variable `match_`p with each of these tests leads to its final value: true if and only if a match for production p exists.

The nonterminal symbol [procrules](v) defines, for each class belonging to the class-model graph, the rule application procedure for objects of this class. For each possible message received by an object of type v (i.e., for each $m \in \mathcal{R}_v$, with $\mathcal{R}_v$ being the class set of rules for class v, built as in Definition 3.15), and for each production p in the object-oriented grammar set of productions which implements

$$
\begin{array}{l}
[\mathsf{testmatch}](v \in V_{\mathcal{C}}, m \in \mathcal{R}_v, p \in \Pi^E_m)\\
\quad \texttt{match\_}\cdot p\cdot\texttt{ = true;}\cdot\\
\quad\quad \left.\begin{array}{l} \texttt{match (attr\_}\cdot v\cdot\texttt{\_}\cdot(tar^i_{\mathcal{C}}\circ t_L)(e)\cdot\texttt{\_}\cdot i\cdot\texttt{\_}\cdot k\cdot\texttt{.type,}\\ \texttt{class\_}\cdot(t_L\circ tar^i_L)(e)\texttt{, match\_tmp, index);}\cdot\\ \texttt{match\_}\cdot p\cdot\texttt{ = match\_}\cdot p\cdot\texttt{ \&\& match\_tmp;}\cdot\end{array}\right\} \quad \forall e\in p_L|_{\mathrm{attr}} : t_L(e)=a^v_k, i=1,\dots,|(tar_{\mathcal{C}}\circ t_L)(e)|\\
\quad\quad \left.\begin{array}{l} \texttt{match\_}\cdot p\cdot\texttt{ = match\_}\cdot p\cdot\texttt{ \&\&}\\ \texttt{(attr\_}\cdot v\cdot\texttt{\_}\cdot(tar^i_{\mathcal{C}}\circ t_L)(e)\cdot\texttt{\_}\cdot i\cdot\texttt{\_}\cdot k\cdot\texttt{==}\\ \texttt{par\_}\cdot v\cdot\texttt{\_}\cdot m\cdot\texttt{\_}\cdot src^j_{\mathcal{C}}(m)\cdot\texttt{\_}\cdot j\texttt{);}\cdot\end{array}\right\} \quad \begin{array}{l}\forall e\in p_L|_{\mathrm{attr}}, m'\in p_L|_{\mathrm{msg}} : t_L(e)=a^v_k\wedge\\ t_L(m')=m\wedge tar^i_L(e)=src^j_L(m')\end{array}\\
\quad\quad \left.\begin{array}{l} \texttt{match\_}\cdot p\cdot\texttt{ = match\_}\cdot p\cdot\texttt{ \&\&}\\ \texttt{(attr\_}\cdot v\cdot\texttt{\_}\cdot(tar^i_{\mathcal{C}}\circ t_L)(e)\cdot\texttt{\_}\cdot i\cdot\texttt{\_}\cdot k\cdot\texttt{==}\\ \texttt{attr\_}\cdot v\cdot\texttt{\_}\cdot(tar^j_{\mathcal{C}}\circ t_L)(e')\cdot\texttt{\_}\cdot j\cdot\texttt{\_}\cdot k'\cdot\texttt{);}\cdot\end{array}\right\} \quad \begin{array}{l}\forall e,e'\in p_L|_{\mathrm{attr}} : t_L(e)=a^v_k\wedge t_L(e')=a^v_{k'}\wedge\\ tar^i_L(e)=tar^j_L(e')\end{array}\\
\quad\quad \left.\begin{array}{l} \texttt{match\_}\cdot p\cdot\texttt{ = match\_}\cdot p\cdot\texttt{ \&\&}\\ \texttt{(attr\_}\cdot v\cdot(tar^i_{\mathcal{C}}\circ t_L)(e)\cdot\texttt{\_}\cdot i\cdot\texttt{\_}\cdot k\cdot\texttt{==}\\ \texttt{opc\_}\cdot v\cdot\texttt{;}\cdot\end{array}\right\} \quad \begin{array}{l}\forall e\in p_L|_{\mathrm{attr}}, m'\in p_L|_{\mathrm{msg}} : t_L(e)=a^v_k\wedge\\ t_L(m')=m\wedge tar^i_L(e)=tar_L(m')\end{array}
\end{array}
$$

$$
\begin{array}{l}
[\mathsf{modifystate}](o\in V_L, m\in E_L, p: L\to R)\\
\quad \left.\begin{array}{l}
\texttt{tmp\_attr\_}\cdot t_L(o)\cdot\texttt{\_}\cdot(tar^i_{\mathcal{C}}\circ t_R)(e)\cdot\texttt{\_}\cdot i\cdot\texttt{\_}\cdot k\cdot\texttt{=}\cdot\\
\left\{\begin{array}{ll}
\texttt{attr\_}\cdot t_L(o)\cdot\texttt{\_}\cdot(tar^j_{\mathcal{C}}\circ t_L)(e')\cdot\texttt{\_}\cdot j\cdot\texttt{\_}\cdot k'\texttt{;}\cdot & - \left[\begin{array}{l}\text{if } t_L(e')=a^{t_L(o)}_{k'}\wedge\\ (p_V\circ tar^j_L)(e')=tar^i_R(e)\end{array}\right.\\
\texttt{par\_}\cdot t_L(o)\cdot\texttt{\_}\cdot t_L(m)\cdot\texttt{\_}\cdot(src^j_{\mathcal{C}}\circ t_L)(m)\cdot\texttt{\_}\cdot i\cdot\texttt{;}\cdot & - \left[\begin{array}{l}\text{if } t_L(m)=m^{t(o)}_{k'}\wedge\\ (p_V\circ src^j_L)(m)=tar^i_R(e)\end{array}\right.\\
\texttt{opc\_}\cdot t_L(o)\cdot\texttt{;}\cdot & - \quad \text{if } p_V(o)=tar^i_R(e)
\end{array}\right.
\end{array}\right\} \quad \odot\\
\quad\quad \begin{array}{l}\texttt{attr\_}\cdot t_R(o)\cdot\texttt{\_}\cdot(tar^i_{\mathcal{C}}\circ t_R)(e)\cdot\texttt{\_}\cdot i\cdot\texttt{\_}\cdot k\cdot\texttt{=}\cdot\\ \texttt{tmp\_attr\_}\cdot t_R(o)\cdot\texttt{\_}\cdot(tar^i_{\mathcal{C}}\circ t_R)(e)\cdot\texttt{\_}\cdot i\cdot\texttt{\_}\cdot k\cdot\texttt{;}\cdot\end{array} \quad i=1,\dots,|tar_R(e)| \quad \odot\\
\quad \odot\,\forall e\notin im(p_E) : t_R(e)=a^{t_R(o)}_k, i=1,\dots,|(tar_{\mathcal{C}}\circ t_L)(e)|
\end{array}
$$

$$
\begin{array}{l}
[\mathsf{sendmsgs}](l\in E_L|_{\mathrm{msg}}, p: L\to R), \text{with } o=tar_L(l)\\
\quad \left.\begin{array}{l}
\left\{\begin{array}{ll}
\begin{array}{l}\texttt{assert(nfull(attr\_}\cdot t_L(o)\cdot\texttt{\_}\\ \cdot(tar^j_{\mathcal{C}}\circ t_L)(e)\cdot\texttt{\_}\cdot j\cdot\texttt{\_}\cdot k\cdot\texttt{.channel));}\cdot\end{array} & - \left[\begin{array}{l}\text{if } t_L(e)=a^{t_L(o)}_k\wedge\\ (p_V\circ tar^j_L)(e)=tar_R(m)\end{array}\right.\\
\begin{array}{l}\texttt{assert(nfull(par\_}\cdot t_L(o)\cdot\texttt{\_}\cdot\\ t_L(m)\cdot\texttt{\_}\cdot(src^j_{\mathcal{C}}\circ t_L)(m)\cdot\texttt{\_}\cdot i\cdot\texttt{.channel));}\cdot\end{array} & - \left[\begin{array}{l}\text{if } t_L(l)=m^{t(o)}_k\wedge\\ (p_V\circ src^j_L)(l)=tar_R(m)\end{array}\right.\\
\texttt{assert(nfull(opc\_}\cdot t_L(o)\cdot\texttt{.channel));}\cdot & - \quad \text{if } p_V(o)=tar_R(m)
\end{array}\right.\\
\texttt{if}\cdot\\
\quad \begin{array}{l}\texttt{::}\cdot tar_R(m)\cdot\texttt{.type == class\_}\cdot c\cdot\texttt{->}\cdot\\ tar_R(m)\cdot\texttt{.channel!msg\_}\cdot c\cdot\texttt{\_}\cdot t_R(m)\cdot[\mathsf{msgarglist}](l,m,p)\cdot\texttt{;}\end{array} \quad \forall c\in\downarrow t_R(tar_R(m))\\
\texttt{fi;}\cdot
\end{array}\right\} \quad \forall m\in E_R|_{\mathrm{msg}}
\end{array}
$$

$$
\begin{array}{l}
[\mathsf{msgarglist}](l\in E_L|_{\mathrm{msg}}, m\in E_R|_{\mathrm{msg}}, p: L\to R), \text{ with } t_R(m)=m^{(t_R\circ tar_R)(m)}_k\\
\quad\quad \left(\texttt{,nil\_object}\cdot\right)^{|src_{\mathcal{C}}(t_R(m'))|}_{i=1}\;\Big|^{k-1}_{j=1} \quad m'=m^{(t_R\circ tar_R)(m)}_j\\
\quad \left(\left\{\begin{array}{ll}
\begin{array}{l}\texttt{, attr\_}\cdot(t_L\circ tar_L)(l)\cdot\texttt{\_}\cdot\\ (tar^j_{\mathcal{C}}\circ t_L)(e)\cdot\texttt{\_}\cdot j\cdot\texttt{\_}\cdot k'\cdot\end{array} & - \left[\begin{array}{l}\text{if } \exists e\in E_L|_{\mathrm{attr}} : src^i_R(m)=(p_V\circ tar^j_L)(e)\wedge\\ t_L(e)=a^{(t_L\circ tar_L)(l)}_{k'}\end{array}\right.\\
\begin{array}{l}\texttt{, par\_}\cdot(t_L\circ tar_L)(l)\cdot\texttt{\_}\cdot t_L(m)\cdot\texttt{\_}\cdot\\ (t_L\circ src^j_L)(m)\cdot\texttt{\_}\cdot j\cdot\texttt{\_}\cdot k'\cdot\end{array} & - \left[\begin{array}{l}\text{if } src^i_R(m)=(p_V\circ src^j_L)(l)\wedge\\ l=m^{(t_L\circ tar_L)(l)}_{k'}\end{array}\right.\\
\texttt{, opc\_}\cdot(t_L\circ tar_L)(l)\cdot & - \quad \text{if } src^i_R(m)=p_V(o))
\end{array}\right.\right)^{|src_R(m)|}_{i=1}\\
\quad\quad \left(\texttt{,nil\_object}\cdot\right)^{|src_{\mathcal{C}}(t_R(m'))|}_{i=1}\;\Big|^{|msg^*_{\mathcal{C}}(t_R(o))|}_{j=k+1} \quad m'=m^{(t_R\circ tar_R)(m)}_j
\end{array}
$$

$$
\begin{array}{l}
[\mathsf{classattrlist}](v)\\
\quad \left(\texttt{; object attr\_}\cdot v\cdot\texttt{\_}\cdot tar^i_{\mathcal{C}}(a^v_k)\cdot\texttt{\_}\cdot i\cdot\texttt{\_}\cdot k\right)^{|tar_{\mathcal{C}}(a^v_k)|}_{i=1}\cdot\;\Big|^{|attr^*_{\mathcal{C}}(v)|}_{k=1}
\end{array}
$$

$$
\begin{array}{l}
[\mathsf{classparlist}](v)\\
\quad \left(\texttt{, par\_}\cdot v\cdot\texttt{\_}\cdot m^v_k\cdot\texttt{\_}\cdot t(src_i(m^v_k))\cdot\texttt{\_}\cdot i\right)^{|src_{\mathcal{C}}(m^v_k)|}_{i=1}\cdot\;\Big|^{|msg^*_{\mathcal{C}}(v)|}_{k=1}
\end{array}
$$

Figure 4.3: Promela translated processes rule application definitions.

that particular message (i.e., for each $p \in \Pi_m^E$), a test to find if a match for that production exists is carried out (the matching procedure was described in the previous paragraph). Next, a conditional test for all variables match_p (which contain the information whether a match for production p into object v exists in the system graph) is performed. Since a conditional test in Promela has a non deterministic result if more than one conditional is true, the choice of which production to apply (provided that at least one match does exist) is also non deterministic, as required by the grammar semantics. All conditionals are guarded by the boolean variable match_p, and each of them leads to the following procedure: (i) the process local variables are modified according to the rule morphism, as described by the nonterminal symbol [modifystate]; (ii) variable event_RuleName is set to rule_v_p, to register which production was applied; (iii) variables event_x, for all classes to which v is related by inheritance, are set to opc_v_id;, the identity of the object that received the message. Notice that an event over an object of type v is also an event over an object of type x, where x is any primitive class of v; (iv) finally, all messages that appear in the right-hand side of p are created. This procedure is described by symbol [sendmsgs](v, p), and it is particularly relevant, since it is this procedure which performs dynamic binding. The right message to send is based on the type of the actual object which is receiving the message. Since the lower set of any node (respecting the inheritance hierarchy) is finite and does not chance along the program execution, a conditional structure takes care of this. If no rule is applied (because no match is possible for any production implementing the received message), then the message is put back in the local buffer, and marked as inspected.

The state graph is modified by changing the values of attributes which are changed by the rule. The attributes that have their values changed are those which are created by the rule application (i.e., those that do not belong to the image of the rule morphism). There are only four types of nodes an attribute can point to in the right-hand side of a rule: (i) a node which was an attribute of the attribute vertex; (ii) a node which was a parameter of the left-side message; (iii) a node that was created by the rule application; or (iv) the attribute vertex itself.

Since we are defining a general translation (i.e. it should be applicable to any object-oriented graph grammar to produce a semantic compatible Promela program), some care is needed when changing the values of object attributes. If we just change the values of attributes in any order, we may get undesirable effects. An example occurs when the values of two distinct attributes are exchanged by a production application. It does not matter in which order we make value attributions, the result will always be wrong, because an auxiliary variable is needed to perform the exchange of two values in sequential programming. Although it is possible to check for all possible dependencies among variable attributions, in order to build a more efficient program, this is not our purpose here (and optimization can always be performed over the translated Promela source code). Therefore, to solve these dependencies problems, we chose an approach that mimic parallel variable value attribution: each object attribute a has a counterpart local variable called tmp_a. Value attribution is first done to this temporary variables and then transported to the actual object attributes. This way, all existing dependencies on the order of attributions are taken care of.

Symbol [modifystate](o, m, p) describes how rule application concerning changes in attribute values is performed in Promela when a production p implementing a

```
[init]
  init { atomic { object nil_object;·
    inheritance[i].primitive = class_·$x$·;·                         $\forall (x,y) \in \sqsubseteq_{V_\mathcal{C}}^{+}, i = 0, \ldots, |\sqsubseteq_{V_\mathcal{C}}^{+}| - 1$
    inheritance[i].derived = class_·$y$·;·
    chan channel_·$o$·=·          }
    [BSIZE] of ·                  }                                     $\forall o \in V_I$
    {mtype (, object)$^i$ };·     }                                     $i = \sum_{k=1}^{|msg_\mathcal{C}^*(t_I(o))|} | src_\mathcal{C}(m_k^{t_I(o)}) |$
    object obj_·$o$;·                       }
    obj_·$o$·.channel = channel_·$o$;·      }                           $\forall o \in V_I$
    obj_·$o$·.type = class_·$t_I(o)$·;·     }
    obj_·$o$·.id = ·$o$·;·                  }
    run ·$t_I(o)$·(·obj_·$o$· $\left((, src_I^i(a_k^{t_I(o)}))_{i=1}^{|src_I(a_k^{t_I(o)})|}\right)_{k=1}^{|attr_\mathcal{C}^*(t_I(o))|}$ ·);·   }     $\forall o \in V_I$
    assert(nfull(·obj_·$tar_I(m)$·.channel));·                          $\forall m \in E_I|_{\text{msg}}$
    obj_·$tar_I(m)$·.channel!msg_·$(t_I \circ tar_I)(m)$·_·$t_I(m)$·[msgparlist]$(o,m)$·;
  } }

[msgparlist]$(o,m)$, with $t_I(m) = m_k^{(t_I \circ tar_I)(m)}$
    $\left((,\text{nil\_object}\cdot)_{i=1}^{|src_\mathcal{C}(t(m)_i^{t(o)})|}\right)^{k-1}$
   obj_·$o_1$·,·obj_·$o_2$·,·...·,·obj_·$o_l$·                  $src_I(m) = o_1 \cdot o_2 \cdot \ldots \cdot o_l$
    $\left((,\text{nil\_object}\cdot)_{i=1}^{|src_\mathcal{C}(t(m)_i^{t(o)})|}\right)^{|msg_\mathcal{C}^*(t(o))|-k}$
```

Figure 4.4: Promela translated program initialization definitions.

method m is applied to object o. An attribute has its value changed if it is not preserved by the rule morphism. According to the definition of an object-oriented rule (Definition 3.6), for any attribute that is not preserved by the morphism, another one (with the same type) must be added to the right-hand side of the rule. This is required in order to maintain the right number and the right type of object attributes along computations (i.e., states remain complete object-oriented graphs). Therefore, for each attribute arc which does not belong to the image of the rule morphism (i.e., was created by rule application), its corresponding parameters on the Promela program are updated, according to each of the sources of the new attribute (hyper)edge: if it is a node which was being pointed by an attribute on the rule left-hand side, a parameter of the left-side message, or the object which received the message itself (the attribute vertex). Notice that those are the only possible nodes in the right-hand side of the rule (aside from the nodes that have been created by the rule application, which we are not considering in this translation). The aforementioned "parallel attribution" is performed after all temporary variables have been updated.

Symbol [init] (Figure 4.4) creates a Promela process named `init`, which is the process equivalent to the main program in imperative programming languages. The `init` program consists of an atomically executed block of commands whose role is to launch all processes corresponding to the objects belonging to the object-oriented graph grammar initial graph. First, the inheritance hierarchy is initialized: all pairs belonging to the transitive closure of the inheritance relation form the `inheritance` array (which is itself an array of pairs). Next, a communication channel channel_o with `BSIZE` slots is created for each initial graph vertex o. The size of each channel is determined by all parameters of all messages object o can receive. It is impossible to know, *a priori*, which message will be received, and since message parameters are actual local variables of each process, it is not possible first to receive a message and only then copy the parameter values to their rightful places. Therefore, each message receives all possible parameters of all possible messages. When a message is sent, however, the parameters which do not belong to that particular message are filled up with an empty object (`nil_object`, the variable defined in the beginning of each process). The list of message parameters is built with symbol [msgparlist](o, m), for each message m belonging to the initial graph.

For each object belonging to the initial graph, a variable of type `object` in created in the program (those variables, however, are local to the process `init`). The structure of each variable has a channel (`channel_`o, previously described, a type (the `mtype` class name `class_`$t_I(o)$), and an identification (the object name o itself). Having all objects created, a process for each one of them is launched, through the Promela command `run`. The arguments of `run` are the process name (which is the name of the class object o belongs to, i.e., $t_I(o)$), the object identity o, and all its attributes (given in the previously defined order on attributes). Since I must be (by definition) a complete object-oriented graph, then all necessary attributes are present, and the call is hence valid. Following the creation of processes for the initial graph, the messages addressed initially to the objects are sent. Message dispatching is performed by writing the message name and parameters into an object channel. This is performed by the last commands generated by [init]. The symbols for sending and receiving messages (! and ?, respectively) in Promela are the same as those traditionally used in process calculi (FOKKINK, 2000) such as CCS (MILNER, 1989a) or the π-calculus (MILNER, 1999).

4.3 Semantic compatibility

In order to prove that the translation presented in Section 4.2 is correct, in the sense that system behaviour is preserved, it is necessary to show that for every possible object-oriented graph grammar computation there exists a corresponding Promela program execution which leads to the same (translated) result, and that the reverse is also true. To compare object-oriented graph grammars executions with Promela translated programs, we compare the *paths* existing in the respective labeled transition systems.

Definition 4.1 (Path) *Given a labeled transition system (LTS) $\mathcal{T} = \langle S^{\mathcal{T}}, s_0^{\mathcal{T}}, L^{\mathcal{T}}, \rightarrow^{\mathcal{T}} \rangle$, a* path *in $\mathcal{T}$ is a sequence of states $s_1, s_2, s_3, \ldots$, where $s_i \in S^{\mathcal{T}}$, and $(s_i, s_{i+1}) \in \rightarrow^{\mathcal{T}}$ for all $i \geqslant 1$. The set of all paths in $\mathcal{T}$ is denoted by $Path(\mathcal{T})$ and it is the set of all paths $s_1, s_2, s_3, \ldots$ where $s_1 = s_0^{\mathcal{T}}$.*

The comparison between an object-oriented graph grammar $\mathcal{G}$ and Promela translated program $\mathcal{F}(\mathcal{G}, n)$ (according to equation 4.1) computations will be carried out by the translation of paths belonging to $Path(\mathcal{T}^{\mathcal{G}})$ to the corresponding paths belonging to $Path(\mathcal{T}^{\mathcal{F}(\mathcal{G},n)})$, and vice-versa. Therefore, states and transitions in $\mathcal{T}^{\mathcal{G}}$ must be translated (respectively) into states and transitions in $\mathcal{T}^{\mathcal{F}(\mathcal{G},n)}$, and the same must be done for the states and transitions in $\mathcal{T}^{\mathcal{F}(\mathcal{G},n)}$, which must be translated into states and transitions in $\mathcal{T}^{\mathcal{G}}$ (if we want to show equivalence).

The proofs concerning behaviour preservation must be based on the Promela language semantics. As an input language for a model checker, one naturally expects that Promela should be equipped with a standard formal semantics. This is, however, not the case. Apparently, there are only three references in the literature where a formal semantics for Promela is provided. The first one (NATARAJAN;

HOLZMANN, 1996), published by the SPIN developers, defines an operational semantics for a subset of the language, while claiming that the only formal semantics for the language is the C implementation of SPIN itself. The second (WEISE, 1997) and third (BEVIER, 1997) ones were presented one year later, in the same SPIN workshop series. (BEVIER, 1997) defines an ACL2 (KAUFMANN; MANOLIOS; MOORE, 2000) specification for the semantics of Promela. It is a very interesting approach, since it is possible to check the semantic definition itself; also, it is possible to define the predicate that defines a legal Promela state and to show that it is an invariant of the semantic definition. The ACL2 specification provides a definition for a subset of Promela, and it is described as a work in progress, but it seems that it did not have posterior developments. Finally, (BEVIER, 1997) presents an incremental structural operational semantics for Promela, extending the work done in (NATARAJAN; HOLZMANN, 1996) to encompass most of the language constructions which had not been treated there. This latter semantic definition was used in (DOTTI et al., 2003) to prove the semantic compatibility of their translation, and some inconsistencies were found regarding the constructions for atomicity.

The semantics of Promela will not be formally (re)constructed here. The semantic definitions given in (NATARAJAN; HOLZMANN, 1996), encompass some language constructs not used in the translation, while others are not described, like structured types, multiple parameterized processes and messages with any (finite) number of arguments. Therefore, formal semantic-based proofs are left for future work. However, we will argue that the proposed translation preserves behaviour, based on a less formal approach.

According to the translation presented in Section 4.2, each initial graph node is transformed into a process, having as parameters all the targets of its attributes (ordered according to the arbitrary total order imposed on the object attributes). Each initial message is put into the proper object channel, together with its parameters (the sources of each message arc in the initial graph). Therefore, targets of attribute edges become processes parameters, and message parameters become processes local variables. Object behaviour (i.e., the process executable code) is constructed from the grammar rules, and it can be summarized by the following pseudo-code:

```
pinit: wait message
       while (true) do
          choose
             . case (there is a message in the primary object channel)
                 move message to local object channel
             . case (there is a message in the local object channel)
                 apply a rule for the message received
             . else
                 jump to pinit
          end choose
       end while
```

The rule application procedure is performed atomically (i.e., it cannot be interleaved with other processes executions). Rule application can be summarized in the following pseudo-code (the complete Promela code generated for each object rule application procedure was given in Figure 4.3):

```
for each message the object can receive do
```

Object-oriented graph grammar	**Promela program**
The i-th attribute edge e with source o and targets $a_1 \ldots a_n$	`attr`$\_(src_{\mathcal{C}} \circ t)(e)\_(tar^k_{\mathcal{C}} \circ t)(e)$`_k_i`, with $k = 1, \ldots, n$ (process o parameter)
The i-th message m with target o and sources $a_1 \ldots a_n$	`msg`$\_t(o)\_t(m)$ (`mtype` name)
The i-th parameter of message m addressed to object o	`par`$\_t(o)\_t(m)\_(src^i_{\mathcal{C}} \circ t)(e)\_i$ (process o local variable)
Object o	`opc`$\_t(o)$ (process o parameter)
Object o main channel	`opc`$\_t(o)$`.channel` (process o parameter)
Object o type	`opc`$\_t(o)$`.type` (process o parameter)
Object o id	`opc`$\_t(o)$`.id` (process o parameter)
Rule p, applicable to object o	`rule`$\_t(o)\_p$ (`mtype` name)

Figure 4.5: Correspondence between grammar and program entities

```
    for each production implementing the message do
        test for a match for the production
    if there is a match for some production
        modify the object state according to the chosen production
        send the messages according to the chosen production
        register the event corresponding to rule application
        register the events corresponding to object identification
        mark all messages in the local buffer as not inspected
    else
        put the message back into the local object buffer
        mark message as inspected
end for
```

Given the translation defined by function $\mathcal{F}$, it is always possible to transform a graph belonging to a computation path in an object-oriented graph grammar into a translated Promela program state. The opposite, however, is not true. The transition system of a translated program $\mathcal{F}(\mathcal{G}, n)$ has several states which do not correspond to any states in the transition system of the original grammar $\mathcal{G}$. This is due to the fact that a rule application in the grammar occurs atomically, while in the Promela program the same action requires several steps. Thus, the states in the transition system $\mathcal{T}^{\mathcal{F}(\mathcal{G},n)}$ which represent a partial treatment of a message or the pre-processing of received messages do not correspond to any state in $\mathcal{T}^{\mathcal{G}}$. Besides that, the process of receiving a message from the primary object channel and moving it to the local message buffer does not have any counterpart on the grammar transition system.

Notice that even the initial states are not the same. The initial state of the grammar transition system is the initial graph. The initial state of the Promela program does not have any active processes corresponding to the objects in the initial graph: it has only one active process (the process `init`), which must run to its completion for all elements of the initial graph to be created. Since the whole `init` code is atomically executed, the moment it executes its last action – and thus terminates – we have a state in the Promela program corresponding to the grammar initial graph. The next state in the program that will correspond to a graph state is the one after the completion of a production application. And, even if the rule application procedure *per se* is performed atomically, the actions related to the motion of messages between the primary and the local object channels do not have any counterparts in grammar derivations. Actually, all process computations performing message motion can be regarded as the same graph state, because only local variables which have nothing to do with the graph state are updated: the state of a process is altered, but the same messages continue to appear in the process channels, and object attributes are not modified.

Therefore, there are multiple states in the program transition system which correspond to the same state in the grammar transition system. In order to transform an object-oriented system graph into a Promela state, we have to make a choice on which of those multiple states we are going to map. A reasonable choice would be to put all messages pointing to an object in the system graph into the primary object channel in the program, and make all objects be at the beginning of the infinite loop that receives/executes messages. Figure 4.6 gives sketch of all `proctype` behaviours, and this would mean that all processes should be at point of execution indicated by letter A (according to the legend in that figure, point A indicates the beginning of the aforementioned infinite loop). The reverse, however, is not a total function, because any state between the points of execution indicated by letters B and D could not be mapped to any graph state, because they represent part of a rule application. Any other state, however, do correspond to a system graph: we only have to map the messages in both local and primary channels as message arcs, respecting the value of their parameters, and make the corresponding attribute edges by inspecting which are the values of the object attributes at a given point of a computation. There is a direct correspondence between graph entities and program variables, which is summarized in Figure 4.5. Therefore, it is straightforward to transform graphs in program states and vice-versa.

Notice that messages, when sent to an object, are stored in buffers accessible only by the recipient object. Once read, their parameters (objects themselves) references are stored in local variables. Therefore, the whole process of acquiring and reading a message only affects the object process itself, and not any other system entity. Rule application, however, can cause an object to modify its own state, which would mean to modify the system graph structure. If the system state is modified, matches can cease to exist, so the matching process is performed within (atomically) rule application procedure. Then the rule application is safe, in the sense that it can certainly be performed.

As an example, consider a program whose initial graph has only two nodes, which are transformed in Promela processes P and Q. Figure 4.7 gives the possible interleaving of processes P and Q executions. Processes names are indexed with the labels used in Figure 4.6 to denote processes points of execution (0 to the process initial state, letters A to I for the remaining states). Error states are not shown, for the sake of clarity. The dashed arrows $\dashrightarrow$ correspond to transitions which exist only in the program transition system, while regular arrows $\rightarrow$ represent transitions

$$\begin{tikzcd}
 & & & \boxed{B} \arrow[r] & \cdots \arrow[r, "4"] & \boxed{D} \arrow[lll, bend left] & & \\
\boxed{P_I} \arrow[r] & \cdots \arrow[r, "1"] & \boxed{A} \arrow[ur, "2"] \arrow[dr, "3"] & & & & & \\
 & & & \boxed{C} \arrow[r, "6"] \arrow[dr, "5"] & \boxed{F} \arrow[r, "7"] & \boxed{G} \arrow[r, "8"] & \boxed{H} \arrow[r, "9"'] & \boxed{I} \arrow[ulllll, bend right] \\
 & & & & \boxed{E} & & &
\end{tikzcd}$$

States		**Transitions**	
P_I	initial state of process P	1	local message buffer initialization
A	beginning of infinite `do` loop	2	local channel is not empty
B	beginning of (atomic) rule application procedure	3	primary channel is not empty
C	choice on which local slot to store the message	4	rule application transitions
D	end of rule application	5	no local slot is empty
E	error state	6	empty slot i is chosen
F	beginning of the message moving process	7	receiving the message from the primary channel
G	message taken out the primary channel	8	sending the message to the local channel
H	message put in the local channel	9	setting the local slot i as occupied
I	local slot marked as occupied		

Figure 4.6: Object process transition system

existing also in the grammar transition system, which correspond to rule applications. Notice that a single transition in the grammar is translated into a sequence of transitions in the program. We use a single transition in the figure because whatever the number of states there is between the beginning and the end of program rule application are, the sequence is performed atomically, and no interleaving of processes execution is allowed in that interval of time.

There is a correspondence between the labels on the grammar transition system (Definition 3.19) and the global variables `event_RuleName` and `event_`x, with $x \in V_{\mathcal{C}}$. The former contains the identity of the production applied, while the latter contain the identity of the object targeted by the rule match. Therefore, a transition label $\langle p, o\rangle$ belonging to the grammar transition system can be translated into the pair $\langle$`event_RuleName`, `event_`$t(o)\rangle$. If either one of them is not visible, then the former transition is labeled with symbol τ, meaning that there is no information valid on any of those variables.

Now, let $\mathcal{G} = \langle I^{\mathcal{C}}, P^{\mathcal{C}}, \mathcal{C}\rangle$ be an object-oriented graph grammar, $\mathcal{C}_O \hookrightarrow \mathcal{C}$ be its subgraph of visible types, and $\mathcal{T}^{\mathcal{G}} = \langle S, s_0, L, \rightarrow\rangle$ be its underlying observation transition semantics. Let f denote the function that transforms a grammar state in a Promela one. Let $I^{\mathcal{C}} = s_0 \xrightarrow{p_1,o_1} s_1 \xrightarrow{p_2,o_2} \ldots \xrightarrow{p_n,o_n} s_n = G^{\mathcal{C}}$ be a path of $\mathcal{T}^{\mathcal{G}}$. We will argue that there is a path $f(I^{\mathcal{C}}) = f(s_0) \xrightarrow{f(p_1),f(o_1)} f(s_1) \xrightarrow{f(p_2),f(o_2)} \ldots \xrightarrow{f(p_n),f(o_n)} f(s_n) = f(G^{\mathcal{C}})$ in $\mathcal{T}^{\mathcal{F}(\mathcal{G})}$, which correspond to the translated path of execution.

The initial state of each Promela process o occurs by the end of the `init` process. Therefore, if there is a transition $s_0 \xrightarrow{p_1,o_1} s_1$, it means that the left-hand side message of production p_1 was generated by process `init` (because a message received is always a result of another message consumption, and it is the first rule applied in the initial graph). If production p_1 could be applied over object o_1, it means that a match existed. Therefore, process o_1 can move the left-hand side message m of p to its local channel (Promela transitions $f(s_0) = f(s_A^0) \xrightarrow{\tau} f(s_C^0) \xrightarrow{\tau} f(s_F^0) \xrightarrow{\tau} f(s_G^0) \xrightarrow{\tau} f(s_H^0) \xrightarrow{\tau} f(s_I^0) \xrightarrow{\tau} f(s_A^0)$) and then apply rule p (Promela transitions

$$\begin{tikzcd}
 & & & P_0Q_0 \arrow[dl, dashed, "p_1"'] \arrow[dr, dashed, "q_1"] & & & & \\
 & & P_AQ_0 \arrow[dl, dashed, "p_3"'] \arrow[d, dashed, "p_2"'] \arrow[dr, dashed, "q_1"] & & P_0Q_A \arrow[dl, dashed, "p_1"'] \arrow[d, dashed, "q_2"'] \arrow[dr, dashed, "q_3"] & & & \\
 & P_CQ_0 \arrow[dl, dashed, "p_6"'] \arrow[dr, dashed, "q_1"] & P_BQ_0 \arrow[dl, "p_4"'] \arrow[dr, dashed, "q_1"] & P_AQ_A \arrow[dl, dashed, "p_3"'] \arrow[d, dashed, "p_2"'] \arrow[dr, dashed, "q_1"] \arrow[drr, dashed, "q_3"] & P_0Q_B \arrow[d, dashed, "p_1"'] \arrow[drr, "q_4"] & P_0Q_C \arrow[d, dashed, "p_1"'] \arrow[drr, dashed, "q_6"] & & \\
P_FQ_0 \arrow[d, dashed] & P_DQ_0 \arrow[d, dashed] & P_CQ_A \arrow[d, dashed] & P_BQ_A \arrow[d, dashed] & P_AQ_B \arrow[d, dashed] & P_AQ_C \arrow[d, dashed] & P_0Q_D \arrow[d, dashed] & P_0Q_F \arrow[d, dashed] \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots
\end{tikzcd}$$

Figure 4.7: Interleaving of the Promela program execution

$f(s^0_A) \xrightarrow{\tau} f(s^0_B) \xrightarrow{\tau} \ldots \xrightarrow{\tau} f(s^0_D) \overset{f(p_1),f(o_1)}{\longrightarrow} f(s^0_A) = f(s_1)$.

This computation is always possible, since only another rule applied to the same object could make a match nonexistent. As explained before, rule application on different objects are always sequentially independent, because matches depend solely on the object attributes, which can only be modified by a rule application in the same object. Therefore, even if there is another rule waiting to be processed in the object local buffer, the choice of which message to apply is nondeterministic. Therefore, any message that could be applied in the grammar can be applied in the Promela program. The same reasoning can be applied to the remaining messages waiting to be processed in the system graph.

The reversed process can also be computed. Given any Promela program path which ends in either points A or D, it is possible to translate it back into a grammar transition system path. Each time any process reaches a D state, the global variables which identify the transition label are updated. Therefore, we can take a snapshot of the Promela program state and translate it back to a graph state. This is possible because at this point no other process is in the middle of a rule application, because an atomic block of commands have just been ended.

4.4 The *Dining Philosophers* problem

The Dining Philosophers problem was first presented by Edsger Dijkstra, in 1965, to illustrate an undesirable situation which can occur with concurrent programs.

A number n of philosophers are sitting around a circular table and each of them has a plate of spaghetti in front of him with a fork at either side (i.e. n philosophers, n plates, and n forks). The life of a philosopher consists of alternating periods of thinking and eating. When a philosopher decides to stop thinking, he has to acquire the forks placed at his right side and at his left side, in order to be able to eat the spaghetti. He cannot start to eat before both forks are grabbed. Since all forks are shared between two different philosophers, adjacent philosophers cannot eat at the same time.

The intuitive approach to solve this problem is to have each philosopher first pick up his left fork, and then his right one. However, this algorithm could lead to a deadlock state, since all four of the necessary and sufficient conditions for deadlock come into play: blocking shared resources (forks), no pre-emption (one philosopher cannot ask any of his adjacent colleagues to drop their forks), holding while acquiring (a philosopher holds his left fork before trying to pick up his right fork) and circular waiting (each two philosophers share a fork) (TANENBAUM; WOODHULL, 1997).

The problem consists in developing an algorithm to avoid both *starvation* and *deadlock*. Deadlock occurs if each of the n philosophers has one fork and no one can get a second one. This situation happens, for instance, in the following scenario: all five philosophers are thinking, and they all decide to eat at the same time. Each philosopher then tries to pick up the fork located at his left side. They all succeed, since all forks are on the table. The attempt made by them all to grab the second fork fails, since all forks are, now, hold by someone. So, the philosopher p_1 waits for the fork grabbed by philosopher p_2 who is waiting for the fork of philosopher p_3 and so forth, making a circular chain. If no philosopher gives up eating once he has decided to do so, no philosopher will ever grab the second fork, and a deadlock situation is set. Starvation might also happen independently of deadlock if a philosopher is unable to acquire both forks. If the philosophers decide to give up eating for a while, to wait the unavailable fork to be released, and they all decide to do that at the same time, and then later begin the process again, a starvation situation occurs, if they all wait the same amount of time before restarting the process (i.e., all philosophers can always perform an action – no deadlock occurs —, but the intended final action (eating) never takes place).

The lack of available forks is an analogy to the locking of scarce shared resources in real computer programming. Locking a resource is a common technique to ensure the resource is accessed by only one program or chunk of code at a time. When the resource the program is interested in is already locked by another one, the program waits until it is unlocked. When several resources are involved in locking resources, deadlock or starvation might happen, depending on the circumstances. For example, one program needs two files to process. When two such programs lock one file each, both programs wait for the other one to unlock the other file, which will never happen.

We illustrate our translation technique using a naive (wrong) solution for the problem presented. We will use the translated model to show the original one is indeed wrong, in the sense that it does not prevent a deadlock to occur. The "solution" for the Dining Philosophers problem is modeled by the class-model graph portrayed in Figure 4.8. The graph nodes, representing the type of objects presented in the system are six: *Philosopher*, which is derived into two different types: *Left-HandedPhilosopher* and *Right-HandedPhilosopher*; *Fork*, which represent the shared resources the philosophers are competing upon; *Table*, which intends to model both the place where the philosophers are sat and from where forks can be picked up; and *ForkHolder*, which can be either a *Philosopher* or a *Table*.

Left-handed and right-handed philosophers differ from one another in the way they start the process of eating. Left-handed philosophers always attempt to pick up their left-side forks first. Only when the left-side fork is acquired they begin the process of grabbing the right-side fork. Right-handed philosophers operate symmetrically. The characteristics of the philosophers sat at the table (i.e., whether they are left or right-handed) can be used to check the desirable system properties.

The attributes are the information the elements must possess to compute cor-

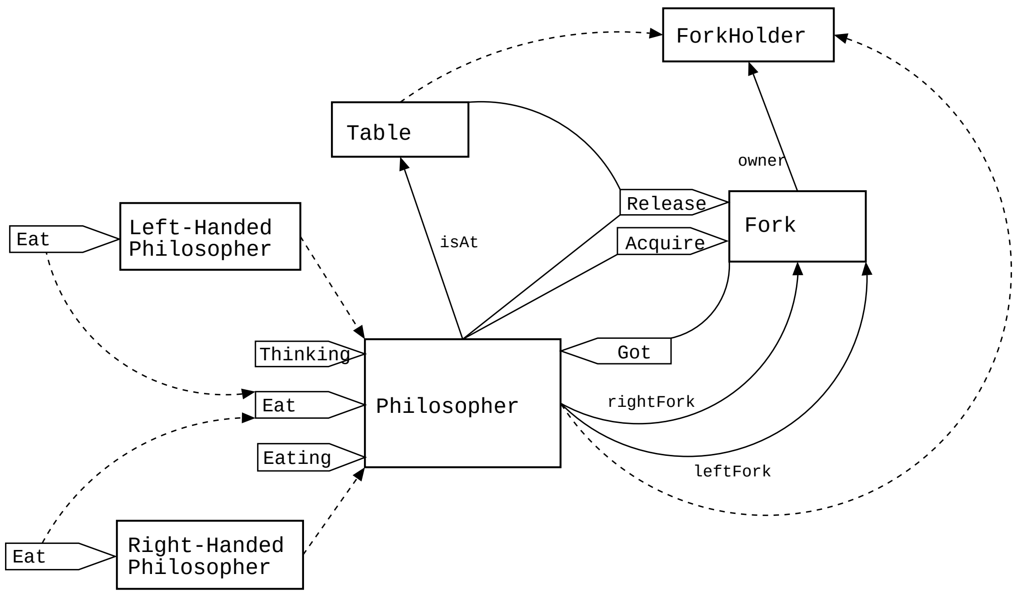

Figure 4.8: Class-model graph for the Dining Philosophers problem.

rectly: a *Philosopher* is at a *Table*, and has a left and a right *Fork* to get in order to eat (notice that a philosopher cannot pick any fork in the table, just the ones closer to him); a *Fork* has an owner, which is a *ForkHolder* (and, by inheritance, a *Table* or a *Philosopher*).

The messages in Figure 4.8 correspond to the actions performed by the actors in the program. A *Fork* can be acquired by a *Philosopher*, and released by a *Philosopher* to a *Table* (messages Acquire and Release, respectively). A *Philosopher* can be Thinking, Eating, or receive a message Eat, which sends him to the process of acquiring his forks, and a message Got, to notify that a *Fork* has been acquired. Left-handed and right-handed philosophers redefine message Eat, as explained in what follows.

Notation: Object-oriented rules left- and right-hand sides are actually object-oriented graphs, i.e., graphs typed over a class-model graph (see Section 3.2). However, in order to make the presentation clearer, all nodes and edges are named after their types, making the typing morphism explicit.

Figure 4.9 presents the rules for class *Fork*. The left-hand side of rule `AcquireFork` has, as expected, a single message Acquire addressed to a node of type *Fork*, having a *Philosopher* as a parameter. Notice that the rule can only be applied if the owner of the *Fork* is of type *Table*: in that case, and only in that case, the owner of the *Fork* is now the *Philosopher*, who is notified that he picked the fork by a message Got (which has the fork grabbed as a parameter). A fork is released (by a message Release) through the grammar rule `ReleaseFork`: an object of type *Fork* receives a message from its current owner (a *Philosopher*), indicating the object of type *Table* to which the fork will now belong, then the *Fork* simply changes the value of its attribute *owner* from the *Philosopher* to the *Table*.

Figure 4.10 presents the rules for class *Philosopher*. Although a *Philosopher* can receive four types of messages, according to the class-model graph for the problem (Figure 4.8), only two of them are implemented by the grammar. The other two

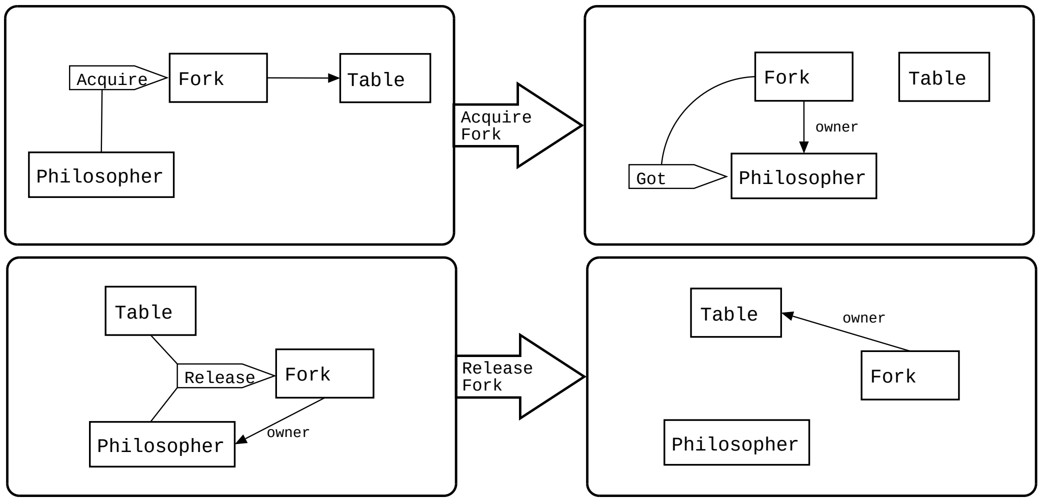

Figure 4.9: *Fork* rules for the Dining Philosophers problem.

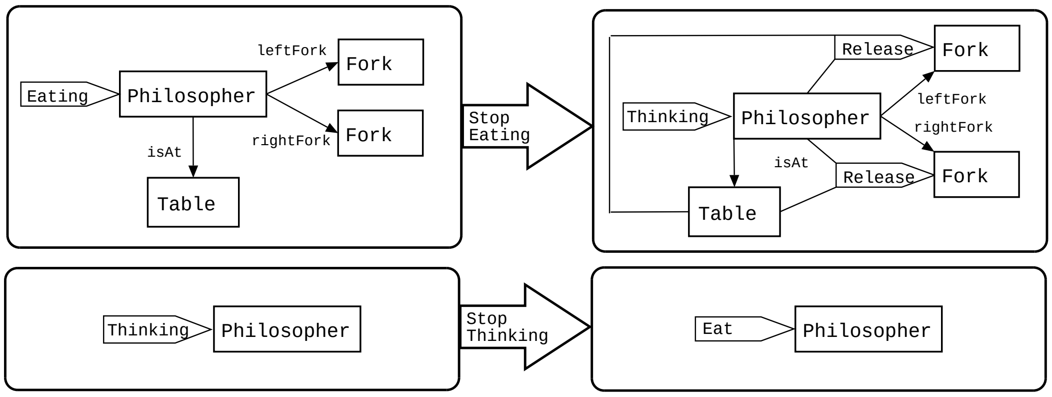

Figure 4.10: *Philosopher* rules for the Dining Philosophers problem.

can be regarded as *abstract* methods, which are methods having the only purpose to be redefined in derived classes, and so assure that dynamic binding is performed in execution time.

The implemented methods for class *Philosopher* are Eating and Thinking. They exist to assure that there could be an undetermined amount of time passed while the philosopher is actually performing those activities. According to the semantics of object-oriented graph grammars (as presented in Section 2.4), there is no particular order or time constraint on rule application. Therefore, a time that a philosopher spends eating or thinking is determined by the time elapsed since the philosopher receives a message (Eating or Thinking, respectively) until it is consumed.

The application of the rule StopEating stops the aforementioned activity of a philosopher: a philosopher which has two forks at a table sends a Release message to his left and to his right forks, using himself and the table he is sat on as parameters. The application of rule StopThinking consumes the existing message Thinking, and stops the thinking activity by sending a message to himself telling it is time to Eat.

A left-handed philosopher is a philosopher who always picks up the left-side fork first. Correspondingly, a right-handed philosophers always begins to eat by picking up his right-side fork. Therefore, rules for *Left-HandedPhilosopher* and *Right-HandedPhilosopher* are symmetric in the sense of the order the forks are taken, and they achieve that by overriding messages Eat and Got from the parent class

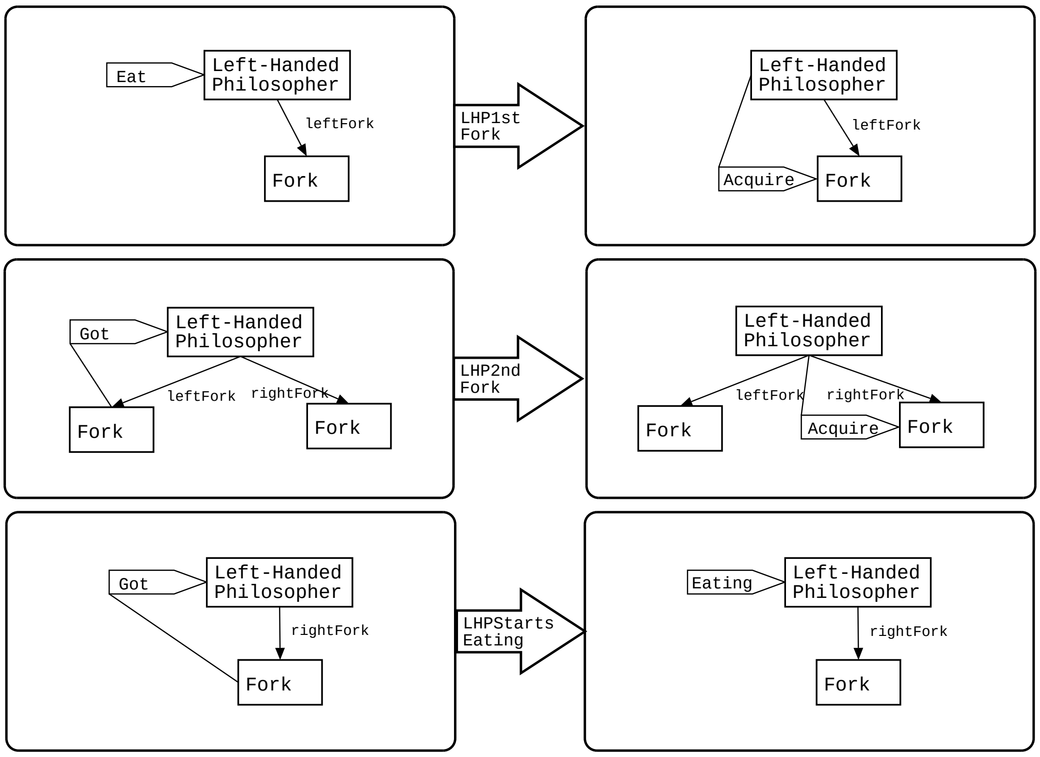

Figure 4.11: *Left-HandedPhilosopher* rules for the Dining Philosophers problem.

Philosopher.

An object of type *Left-HandedPhilosopher* responds to a message Eat by trying to lift its left-side fork. This is modeled by sending a message Acquire to its attribute typed as *leftFork*. Once the left-handed philosopher receives a message Got from the *Fork* he was trying to pick up, he can start pursuing his other fork, which is achieved by sending another Acquire message, now to his *rightFork* attribute. When a message Got is received from this *Fork* then the left-handed philosopher sends himself a message Eating to signal that the period of eating has started. An object of type *Right-HandedPhilosopher* performs the exact same actions, only beginning with its right-side fork. Figures 4.11 and 4.12 present the rules for classes *Left-HandedPhilosopher* and *Right-HandedPhilosopher*, respectively.

Figure 4.13 shows the initial graph scheme for the Dining Philosophers problem. We have five philosophers (whether they are left or right-handed will be considered next) — Socrate, Plato, Nietzche, Hegel and Kant — sat at a table, having a fork between each two of them. Initially, all philosophers are thinking, as indicated by the messages sent to each one of them. The process of thinking can be stopped at any time, through rule `StopThinking` (Figure 4.10), which only requires the existence of a message Thinking to be applied.

The complete translated Promela program for the Dining Philosophers problem is listed in Appendix E, Section E.3.

A number of properties are desirable to be present in a program that implements the Dining Philosophers problem. Particularly for this example, we will check for the following properties:

1. Deadlock absence — a philosopher can always perform an action.

2. Mutual exclusion — any fork is hold by at most one philosopher at a time.

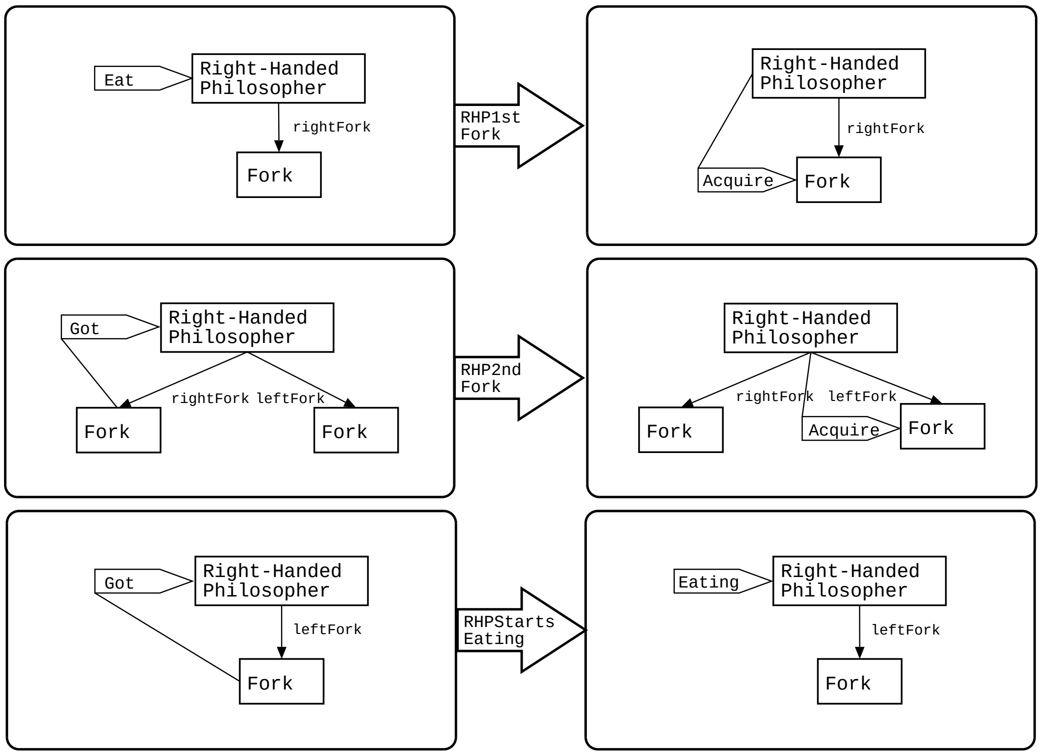

Figure 4.12: *Right-HandedPhilosopher* rules for the Dining Philosophers problem.

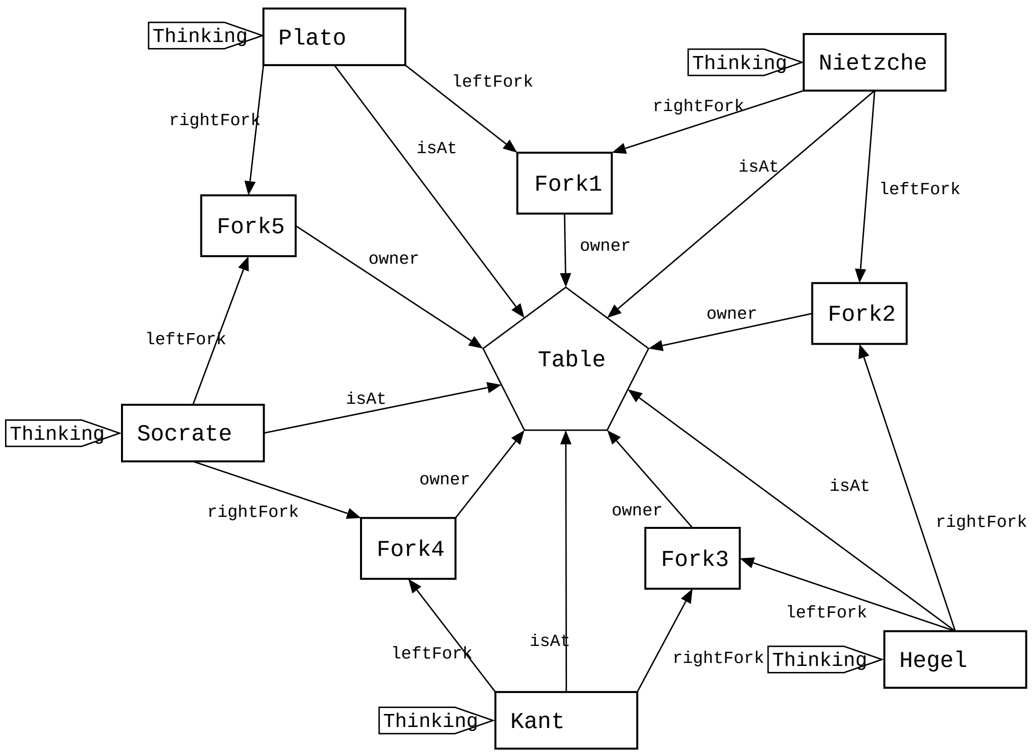

Figure 4.13: The initial graph for the Dining Philosophers problem.

3. Starvation absence — if a philosopher decides to eat, he eventually does so.

4. Cyclic behaviour — periods of thinking and eating succeed each other, in all computations of the system.

All of those four properties can be stated in propositional linear temporal logic. It would be interesting to express them graphically, following the formalism used in specification. This is not done here, as properties must be specified over the translated terms. We will use only events to model those properties, and so we will make use of only the global variables which label the transitions in both underlying transition systems. We believe this is a less obscure way of specifying properties. State verification would still be possible, however state variable names are more complicated. This, however, could be circumvent by defining a simple visual graph language for temporal logic properties (like the one proposed in (KOCH, 1999), for instance).

Transition systems generated by object-oriented graph grammars executions are labeled with production names and the identification of the target object for that production (Definition 3.19). The generated Promela program translates that information into the global variables `event_RuleName` and `event_`x, for all nodes x belonging to the class-model graph $\mathcal{C}$ which types the translated grammar $\mathcal{G}$. The first one contains the identity of the production applied (a `mtype` name `rule_`v`_`p, with $v \in V_{\mathcal{C}}$ and $p \in P^{\mathcal{C}}$, such that a node typed as v is the attribute vertex of production p left-hand side), and the other ones hold the identity of the object to which the mentioned production was applied.

Property verification in SPIN can be done using a multiplicity of methods, among which there is LTL property verification. As described in Section E.2, LTL is a propositional linear time temporal logic, and so properties must be described using only propositions. Since there are only temporal quantifiers, quantification over values cannot be written, so all possible values must be tested explicitly. The XSpin tool allows that propositions can be defined in a C-like way, using the preprocessor macro `#define`. The properties we want to verify will be presented next, and are all event based. Meaningful events to verification, in the case of the Dining Philosophers problem, can be stated, for instance, as "philosopher X starts to eat", or "fork Y is grabbed by a philosopher". Those properties will be defined int terms of the actual objects belonging to the system initial graph (i.e., objects $o \in I^{\mathcal{C}}$). The following macros aim to identify those elements (which were portrayed in Figure 4.13). Since we are only interested in the behaviour of the philosophers (Socrate, Plato, Hegel, Kant, and Nietzche), the following propositions are defined:

```
#define isPlato     (event_Philosopher==Plato)
#define isSocrate   (event_Philosopher==Socrate)
#define isHegel     (event_Philosopher==Hegel)
#define isKant      (event_Philosopher==Kant)
#define isNietzche (event_Philosopher==Nietzche)
```

Notice that there is no need to define propositions for left-handed or right-handed philosophers, since if there is an event for an instance of a subclass, the same object id is set to all superclass variables. Therefore, only the superclass of interest can be used.

The next propositions state events of interest occurring within the system. Namely, they identify when a philosopher wants to eat, begins to eat, and stops to eat.

```
#define aPhilWantsToEat (event_RuleName==rule_Philosopher_StopThinking ||
               event_RuleName==rule_LeftHandedPhilosopher_StopThinking ||
               event_RuleName==rule_RightHandedPhilosopher_StopThinking)

#define aPhilStartsToEat
              (event_RuleName==rule_LeftHandedPhilosopher_StartsEating ||
               event_RuleName==rule_RightHandedPhilosopher_StartsEating)

#define aPhilStartsToThink (event_RuleName==rule_Philosopher_StopEating ||
               event_RuleName==rule_LeftHandedPhilosopher_StopEating ||
               event_RuleName==rule_RightHandedPhilosopher_StopEating)
```

Now, we must define when a known event occurs with a specific object. The propositions below state that:

```
#define philPlatoWantsToEat     (isPlato && aPhilWantsToEat)
#define philSocrateWantsToEat   (isSocrate && aPhilWantsToEat)
#define philHegelWantsToEat     (isHegel && aPhilWantsToEat)
#define philKantWantsToEat      (isKant && aPhilWantsToEat)
#define philNietzcheWantsToEat  (isNietzche && aPhilWantsToEat)

#define philPlatoStartsToEat     (isPlato && aPhilStartsToEat)
#define philSocrateStartsToEat   (isSocrate && aPhilStartsToEat)
#define philHegelStartsToEat     (isHegel && aPhilStartsToEat)
#define philKantStartsToEat      (isKant && aPhilStartsToEat)
#define philNietzcheStartsToEat  (isNietzche && aPhilStartsToEat)

#define philPlatoStartsToThink     (isPlato && aPhilStartsToThink)
#define philSocrateStartsToThink   (isSocrate && aPhilStartsToThink)
#define philHegelStartsToThink     (isHegel && aPhilStartsToThink)
#define philKantStartsToThink      (isKant && aPhilStartsToThink)
#define philNietzcheStartsToThink  (isNietzche && aPhilStartsToThink)
```

Using the propositions defined above, LTL properties about the system behaviour can be written. We are mainly interested in verifying safety and liveness properties. Safety properties are the ones to assure that "nothing bad ever happens", while liveness properties state that "something good eventually happens". Safety properties regarding the Dining Philosophers problem state that adjacent philosophers never eat at the same time. Using LTL syntax, the property above can be stated as

```
 [] ((philPlatoStartsToEat ->
          (!(philSocrateStartsToEat || philNietzcheStartsToEat) U
                  philPlatoStartsToThink))
 && (philSocrateStartsToEat ->
          (!(philPlatoStartsToEat || philKantStartsToEat) U
                  philSocrateStartsToThink))
 && (philKantStartsToEat ->
          (!(philSocrateStartsToEat || philHegelStartsToEat) U
                  philKantStartsToThink))
 && (philHegelStartsToEat ->
          (!(philKantStartsToEat || philNietzcheStartsToEat) U
                  philHegelStartsToThink))
 && (philNietzcheStartsToEat ->
          (!(philHegelStartsToEat || philPlatoStartsToEat) U
                  philNietzcheStartsToThink)))
```

where the symbols `<>`, `[]` and `U` represent the usual linear temporal logic quantifiers $\Diamond$ (eventually), $\Box$ (always), and $\mathcal{U}$ (until) (see Section E.2 for the formal syntax and semantics of LTL).

Liveness properties usually refer to the lack of deadlock and starvation. For the philosophers, it mirrors the idea that when a philosopher decides to eat, it eventually does so. This property is stated below:

```
[] ((philPlatoWantsToEat -> <> philPlatoStartsToEat) &&
    (philSocrateWantsToEat -> <> philSocrateStartsToEat) &&
    (philKantWantsToEat -> <> philKantStartsToEat) &&
    (philHegelWantsToEat -> <> philHegelStartsToEat) &&
    (philNietzcheWantsToEat -> <> philNietzcheStartsToEat))
```

It should be noticed that we are using two different variables in the Promela program to identify when a grammar event occurs. The actions of changing variable values generate (at least) two distinct states in the transition system corresponding to the program behaviour. Therefore, we can end up with a problem regarding the consistency of states when compared with the grammar counterpart. This problem is solved in the translation by enclosing the attributions of object and rule identification between a `d_step` block. A `d_step` sequence is executed as if it were one single indivisible statement. It is comparable to an `atomic` sequence, but it differs in the sense that no system states are saved, restored, or checked during the execution of a `d_step` sequence. Hence, a `d_step` sequence, no matter how long it is, generates a single transition in the underlying automata, thus guaranteeing the consistency of verification.

If all philosophers are right-handed (or left-handed, for that matter), then the liveness property stated before is not true within the model provided. Figure 4.14 shows a graphical counterexample (taken from the model checker output, and generated by the system developed in (SANTOS, 2004)) for them. The counterexample shows three philosophers (Nietzche, Hegel, and Kant) and their respective forks. The processes are indicated by the vertical lines, and the arrows indicate the messages arriving and departing from each process. Notice that a deadlock situation is set: each philosopher was able to grab one fork, and send a message to the other fork to acquire it. However, since each fork now has a philosopher owning it, rule `AcquireFork` (Figure 4.9) cannot ever be applied again, and all philosopher will wait forever. The safety property, however, is true, as should be expected.

A last note on the power of expression of object-oriented graph grammars deserves to be raised. At a first glance, it seems that object-oriented graph grammars, as described in this work are incapable of describing many things due to the lack of attribute values. It is certainly true that we cannot represent, for instance, strings or real numbers using that formalism. Enumerations, however, can be represented,

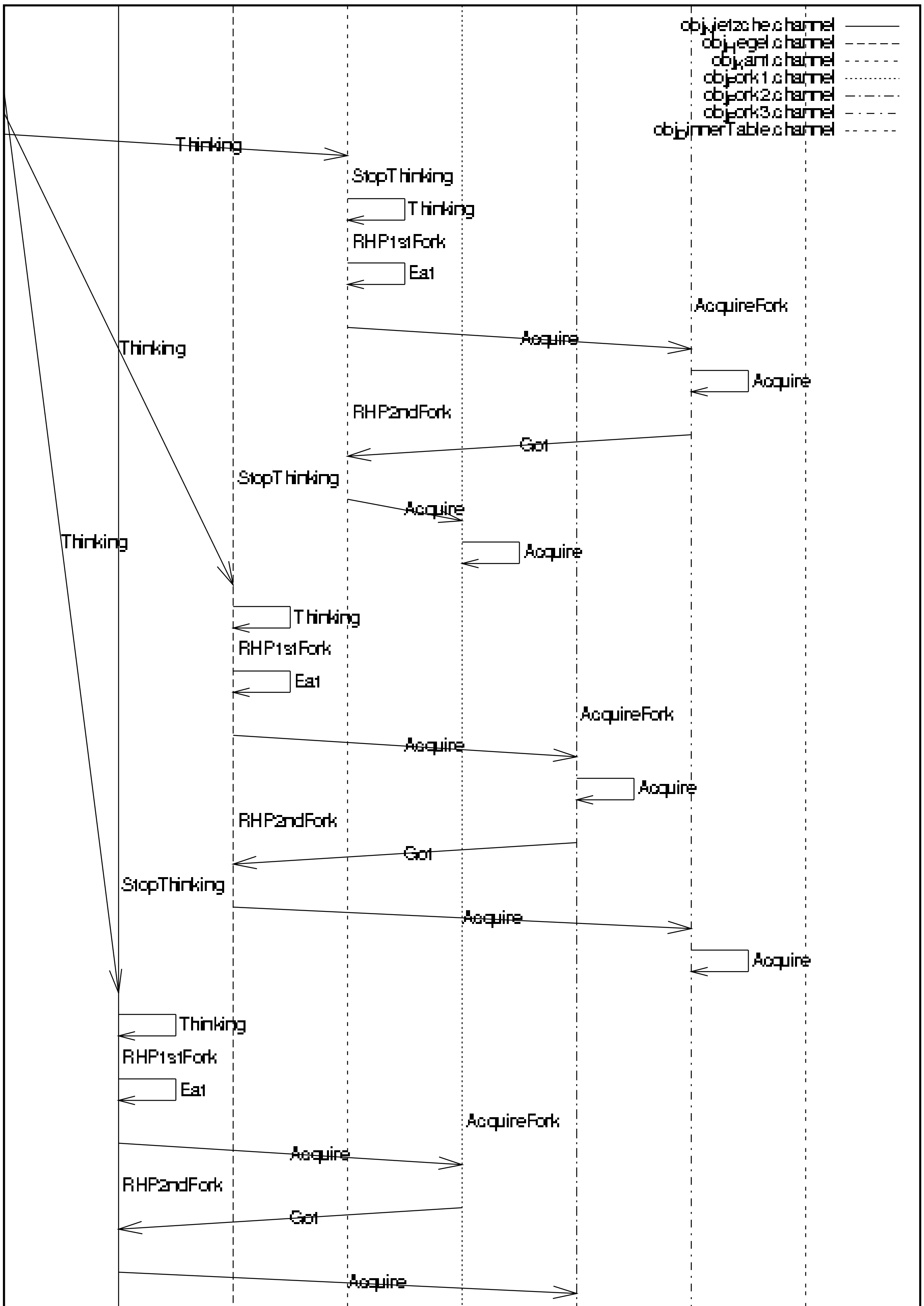

Figure 4.14: Counterexample of the absence of deadlock property

through the mechanism of inheritance. As an example, we have used inheritance to discriminate between a fork being held by a table or by a philosopher. Only in the event a fork is owned by a table, rule `AcquireFork` can be applied (Figure 4.9). It means that enumerations of values can be modeled by subclasses, while the type is their parent class. Although the expressiveness of object-oriented graph grammars is not a subject of investigation here, we believe they have the same power of expression of grammars holding algebraic types with finite sorts. This investigation is left for future work.

4.5 Summary

This chapter presented a set of definitions to allow that specifications written as object-oriented graph grammars (described in Chapter 3) can be formally verified using the model checker SPIN.

A formal translation from object-oriented graph grammar specifications into Promela programs was presented. Objects in the system graph are modeled as Promela processes, having as parameters the object attributes; messages are modeled as buffered communication channels. Buffered channels in Promela have a *first come, first served* (FIFO) policy. This behaviour is changed to become compatible with rule application semantics of graph grammars: messages, when received by a process, are moved and stored in a local array. The choice made on which local array slot to place the received message is deterministic, in order to shrink the translated program state size, but the choice of which slot to take the message from to be consumed is not. Therefore, the local array can be seen as a way to shuffle received messages, so they do not necessarily be processed in the exact order of reception.

Inheritance is encoded in the translated program as a global array variable visible to all processes. Inheritance is used to perform matches, by checking if the match image of the attributes targets (on the rule left-hand side) are correctly typed, assuring that subclass polymorphism is properly applied. Dynamic binding is implemented as a message dispatch mechanism, which checks the actual type of the message to be sent recipient. This way, the right type of message to be sent can be determined in execution time, as required. The novelty of that approach is the implementation of object-oriented features in Promela, which does not have any of those features built in.

The translation presented allows for both state and event verification. SPIN only performs state-based verification, so we use global variables to allow verification over events to be performed. A global variable for each class belonging to the grammar underlying class-model graph exist to identify the last (visible) object of that type that had a (visible) production applied to it. There exists also a global variable to identify which rule was applied, which is used to verify (through temporal logics properties) possible orders in which rules can be applied. Attributions of object and rule identification are enclosed between a `d_step` block, which generates a single transition in the underlying automata, thus guaranteeing the consistency of verification.

Rule application is translated in Promela as a procedure which non deterministically retrieves a message from the local buffer, and checks which matches exist for productions implementing that message. A production is chosen to be applied (also non deterministically) if a match for it exists. Rule application is performed atomically, to mimic the way rules are applied in a graph grammar, where the whole matching and application procedure is performed in a single step.

Semantic compatibility between the behaviour of the original object-oriented graph grammar and the translated Promela program is discussed. To compare object-oriented graph grammars computations with Promela translated programs executions, we compare the *paths* existing in the respective labeled transition systems. The comparison is not formally proven because Promela lacks a formal semantics definition.

Finally, the resulting program can be fed into the model checker SPIN, and properties can be defined over the actual objects of the original grammar or over events such as rule applications over determined objects. Verification is then performed automatically by the model checker. As an example of that procedure we show a translation for the Dining Philosophers problem specified as an object-oriented graph grammar. Safety and liveness properties are specified and verified and the results are shown.

5 RELATED WORK

5.1 Object-oriented models and specification languages

Ever since Simula (NYGAARD; DAHL, 1981) was invented in the 1960's, object-oriented programming has been one of the fastest growing paradigms of computer programming languages. It is true that it took a while (mainly due to hardware constraints) for it to become popular, but in the last twenty years, with the development of commercial programming languages such as C++ (STROUSTUP, 2000), Java (CAMPIONE; WALRATH; HUML, 2000), and recently C# (MICROSOFT CORPORATION, 2005), object-oriented programming has become ubiquitous within the field of software development.

While object-oriented languages had their use spread throughout the world, underlying theories for explaining their meaning and logics lagged far behind. This lack of formal foundations did not occur with other programming paradigms: the semantics of functional programming languages can be understood by looking at the model of computation on which they were based, the λ-calculus; logic programming relies on first-order logic with unification of terms, which were formally defined long before the paradigm to appear; imperative programming languages and their semantics (denotational, axiomatic, and specially operational) have been a matter of study for a long time.

The result of this is that most object-oriented models of computation are based on others, previously existing models (as it is in this work) or actual programming languages. Similarly to the models of computation, most object-oriented formal specification languages were built over previously existing specification languages aiming at the modeling of general software systems. Although the formalism proposed in this work is not an object-oriented specification language *per se*, we believe it is interesting to compare its evolution with respect to graph grammars to some object-oriented formal specification languages built on top of pre-existing general purpose ones. First, we analyze some data-oriented, state-based methods, then we will do the same for process-based ones.

VDM (Vienna Development Method) (BJORNES; JONES, 1978) was initially a language description method inspired by denotational semantics. Compared to the standard VDM-SL language (PLAT; LARSEN, 1992), VDM++ (FITZGERALD et al., 2005), (CSK CORPORATION, 2005) has been extended with classes. With the object oriented facilities offered by VDM++ it is possible to define classes and create objects, define associations and create links between objects, make generalisation and specialisation through inheritance, describe the functional behaviour of the objects using functions and operations, and describe the dynamic behaviour of the system through threads and synchronization constraints. It is interesting to note how some of the object-oriented constructs inserted in VDM++ were actually taken from modern object-oriented programming languages. For instance, access specifiers are called *public*, *protected*, and *private*, the same names used in C++, Java, Delphi, and others.

Perhaps the most well-know set theoretic based method for formal system specification is the Z notation. Z can be roughly described as a syntactic envelope built on top of usual set-theoretic notations (MONIN, 2003) and first-order predicate logic. Sets are used in Z as a universal means of expression: the state space of a system is a set, types are sets, even operations are sets (actually, they are relations, which is just a special case of set). Symbols for various kinds of relations are provided (functions, partial functions, injections, etc.) along with a number of set-theoretic operations allowing for the construction of new relations from previously defined ones. The space state and the operations of a system are declared by means of *schemas*. A schema can contain data, operations and predicates that must hold for the variables and data it contains. Z provides means for schema composition, which gives rise to a *schema calculus* that can be used for consistency and verification purposes. *Object-Z* (SMITH, 1992), (SMITH, 1999) is an extension of Z in which the existing syntax and semantics of Z are retained and new constructs are introduced to facilitate specification in an object-oriented style. Extensions in Object-Z include the class schema which captures the object-oriented notion of a class by encapsulating a single state schema with all the operation schemas which may affect its variables, a parallel operator for composing and synchronizing operations. Object-Z supports both single and multiple inheritance and allows inheritance hierarchies to be used polymorphically. It effects the merging of the type, constant and schema definitions in each of the inherited classes with those declared explicitly in the inheriting class. Therefore, it strongly relies on name spaces, which allows also for operation redefinition.

Algebraic specification (WIRSING, 1998) techniques also have been extend to accommodate object-oriented features. Actually, algebraic specifications have been the first ones to serve as a formal foundation for object orientation (BREU, 1991). OBJ (GOGUEN; MALCOM, 1996) is a family of languages based upon logical systems, in the sense that their programs are sets of sentences in some logical system, and their operational semantics is given by deduction in that logical system. OBJ has three kinds of entitities at its top level: objects, theories, and views. An object encapsulates executable code, while a theory defines properties that may (or may not) be satisfied by another object or theory. Both objects and theories are modules. A view is a binding of the entities declared in some theory to entities in some other module, and also an assertion that the other module satisfies the properties declared in the theory. Modules can import other previously defined modules, and therefore an OBJ program is conceptually a graph of modules. Modules have signatures that introduce new sorts and new operators among both new and old sorts. Variables with declared sorts can also be defined. Terms are built up from variables and operators, respecting their sort declarations. Modules can be parameterized, and parameterized modules use theories to define both the syntax and the semantics of their interfaces. Views indicate how to instantiate a parameterized module with an actual parameter. This kind of module composition is, in practice, more powerful than the purely functional composition of traditional functional programming, because a single module instantiation can compose together many different functions all at once, in complex ways. The order-sorted specification implemented by OBJ provides a notion of subsort that implements multiple inheritance and overloading.

Petri nets (REISIG, 1985) are commonly used for the formal specification of concurrent systems. They possess a graphical representation, and a wide range of automated and semi-automated analysis techniques. It has been shown that graph grammars are a proper generalization of Petri nets (GENRICH et al., 1983), (KORFF; RIBEIRO, 1996), which allow to specify structured states (graphs instead of place vectors) and context dependent transitions, where part of the precondition is read but not consumed (CORRADINI et al., 1993). The theory and languages of Petri nets have also been extended to deal with object-oriented features. Object Petri nets are presented in (LAKOS, 2001) to support an integration of object-oriented concepts into Petri nets, including inheritance polymorphism and dynamic binding. A single unified class hierarchy encompasses both token and subnet types (a token can thus encapsulate a subnet, to model that object attributes can be objects themselves), which means that a net can have multiple levels of activity (corresponding to object executions). Each data element is defined in a Z schema-like incremental style with fresh elements being added to inherited ones. Overriding of functions is allowed, and polymorphism is achieved through the abstraction of places and transitions. Reachability analysis is based on token/place names, thus only a finite number of them are allowed and so new elements cannot be created. The same approach of nesting in used in (VALK, 2001), which investigates high-level modeling capabilities through the study of higher-level net tokens having individual dynamic behavior — tokens are modeled by Petri nets themselves. *Object nets* behave like ordinary tokens, in the sense that they are lying in places and can be moved around by transitions.

The CO-OPN/2 formalism was first presented in (BIBERSTEIN; BUCHS; GUELFI, 1997) and further developed in (BIBERSTEIN; BUCHS; GUELFI, 2001). While the first was aiming at integrating Petri Nets with object-oriented constructs, using a net-like definition of inheritance, polymorphism and dynamic binding, the latter uses an algebraic approach. Classes are templates described as algebraic nets in which places play the role of attributes, and methods are external parameterized transitions. Interactions between objects consist of synchronization expressions, which allow the implementation of different interactions policies. (Inclusion) polymorphism and inheritance derive from the use of order-sorted algebras for the data structures used. The model also makes a clear distinction of inheritance and subtyping, and its semantics is based on a set of structural operational semantics (SOS) rules plus a semantic mechanism to deal with identifiers.

Another work on hierarchical object-oriented nets is (HE; DING, 2001), where object-oriented specifications are built using hierarchical predicate transition nets, using an algebraic specification for data structures, and where classes are modeled as nets and objects are copies of classes with an unique identification. Data and process abstraction give rise, respectively, to the notion of super predicates and super transitions, which allow the definition of object aggregation and inheritance. Inheritance is implemented through nested supernodes denoting containing and contained classes. Derived classes use an explicit identification for inherited elements, which are added to the new subclass, and polymorphism is derived from the underlying algebraic specification.

Most of the Petri nets extensions to object-oriented features are based on algebraic specification for data and nets for data behaviour. It seems that the hierarchical approach, with tokens allowed to be nets themselves is another trendy line of work, mainly to deal with the known scalability problem of Petri nets. Analysis for the latter case, however, is a hard problem.

The idea of formalizing objects in a categorical setting can be dated back to (EHRICH; SERNADAS; SERNADAS, 1987). They propose a cocomplete category to model objects and their interactions, based on the algebraic data type specification theory. Object aggregation, subtyping relations, and general relations such as "part of" or "aspect-of" are shown to be definable as categorical elements and constructions. It is interesting how this work points to the coalgebraic (JACOBS; RUTTEN, 1997) notion of object, defined almost ten years later (JACOBS, 1996).

Coalgebras and object-oriented systems have been related mainly by the work of Bart Jacobs. In his coalgebraic approach to object-orientation, the operations (attributes and methods) in a class are understood jointly as a single coalgebra, acting on some state-space. Objects of this class are elements of this state-space. A coalgebraic class specification describes a class as some uninterpreted coalgebra satisfying certain assertions (JACOBS, 1996), given in equational logic. Two objects are bisimilar if they are observationally indistinguishable, i.e., if their operations are unable to produce any observational difference between them.

CCSL (ROTHE; JACOBS; TEWS, 2001) is a specification language that combines both algebraic and coalgebraic elements. The CCSL compiler translates CCSL specifications into higher-order logic either for PVS or for Isabelle/HOL proof assistants. After translation the theorem prover can be used to examine the specification, build models, or construct refinements. The specification language CCSL contains single sorted, parametric coalgebraic specifications with the following elements: coalgebraic signatures correspond to higher-order polynomial functors, that is, to polynomial functors with arbitrary exponents. The class of models is restricted with axioms in higher-order logic, and the logic is extended with behavioural equality and modal operators. Inheritance of specification allows one to (monotonically) extend specifications. It also contains parametric algebraic abstract data types (as they are found in PVS and Isabelle) as well as signature extensions with type constructors and arbitrary (polymorphic) constants. In (JACOBS, 1998), CCSL is used as input to the *LOOP* tool (HENSEL et al., 1998), which is a front-end tool for a theorem prover, to reason about classes. This reasoning process allows for proving properties about bisimulations and invariants. A number of developments of this work relates coalgebras with modal logics, in the same sense equational logic is related to algebraic specification (JACOBS, 2002), (JACOBS, 2000), (CIRSTEA, 2003), (KURZ, 2001), (PATTINSON, 2001), (RÖSSIGER, 2000). Coalgebraic approaches have been taken recently also in the area of semantic of concurrent programming languages, mainly by categorically relating their denotational and operational semantics (RUTTEN; TURI, 1994), (TURI, 1996).

The spreading in the use of object-orientation as the main current programming paradigm causes two distinct and complementary developments for the area. First, theoretical foundations for object-oriented programming and computations started to be pursuit more consistently. Second, traditional formal methods of system specification began to be extended to accommodate the inherent features of the paradigm. It will discussed in Section 5.2 specifically the attempts to incorporate those features into the different graph transformation approaches, and how this thesis is a contribution in that area.

5.2 Graph grammars and object-oriented systems

Graph grammars have been used to model systems and their computations since they were first introduced (PFALTZ; ROSENFELD, 1969), as a means of solving picture processing problems.

The *algebraic approach to graph grammars*, presented for the first time in (EHRIG; PFENDER; SCHNEIDER, 1973) makes use of categorical constructs to define the relevant aspects of the model of computation provided by graphs grammars. That approach is currently known as *double-pushout* approach, because derivations are based on two pushout constructions in the category of graphs and total graph morphisms. The *single-pushout* approach (LÖWE, 1991), on the other hand, has derivations characterized as a pushout construction in the category of graphs and partial graph morphisms. It is a proper extension of the double-pushout approach (EHRIG et al., 1996) capable of dealing with addition and deletion of items in unknown contexts, which is an important feature for distributed systems. A main advantadge regarding the algebraic approach is that, since it relies on categorical constructs, if those constructs are proven to exist respecting modified objects and arrows within suitable categories, all results can be directly inherited. The single-pushout approach is one of such modifications, where objects are the same as the ones in the double-pushout approach, but arrows are partial graph morphisms, instead of total ones. The algebraic approach, originally defined in terms of labeled graphs and labeled graph morphisms, is suitable for modeling systems and their computations. Concepts of parallel and distributed productions and derivations in the algebraic approach are very useful to model concurrent access, aspects of synchronization, and distributed systems based on local and global graphs (see (EHRIG; ROSEN, 1980), (LÖWE, 1991), (EHRIG; LÖWE, 1993), (KORFF, 1995), (TAENTZER, 1996a), (HECKEL, 1998), and (MONTANARI; PISTORE; ROSSI, 1999)).

Although the literature is filled with developments in (parallel, concurrent, distributed, mobile) system specifications using graph grammars, object-oriented specification is mostly left aside. There are a few papers reporting the use of graph grammars to model object-oriented systems and programs. Most publications deal with object-based systems, which differ from object-oriented systems because encapsulation and data hiding are taken into consideration, but inheritance and polymorphism are not. Object-based graph grammars are used to model mobile systems and their computations (DOTTI; RIBEIRO, 2000). Those grammars only allow rules that do not interfere with attributes of different objects, thus implementing data hiding. However, inheritance is not considered. Distributed state graphs modeling objects in a network are presented in (TAENTZER, 1996b). Again, objects are entities with an internal state, but inheritance relations are left aside. (PAPADOPOULOS, 1996) presents a work where the generalized computational model of *term graph rewriting* is used as the basis for expressing concurrent object-oriented programming. Concurrency is expressed by the very nature of the model it is based on, and so all levels of interaction among entities can be performed in parallel, and it is naturally language independent. No reference to polymorphism is done in that work. (RENSINK, 2004) presents a tool, called GROOVE, to generate the space-state of a graph grammar, in the attempt that the resulting transition system can be model checked. Java programs are translated into graphs and graph productions, but inheritance and polymorphism are not taken into account.

(WAGNER; GOGOLLA, 1996) gives an interpretation of attributed graphs in terms of partial algebras. Using total algebras, attributed graphs can be modeled as a combination of one algebra for the graphical part and another algebra for the data type component. Hence a transformation step consists of the transformation of the graphical part, the transformation of the data part, and a relating step where attributions are computed. This approach switches to partial algebras, and attributed graphs can now be seen as a single partial algebra. Single-pushouts in the category of partial algebras are defined (as well as necessary conditions for pushout existence). Partial algebras, seen as an attribute hypergraph, are then used to give a semantics for the behaviour of object programming language TROLL light (CONRAD; GOGOLLA; HERZIG, 1992) programs. Particularly, valuation and interaction rules in TROLL light are translated to graph grammar rules. This approach, however, also focuses on the data and code encapsulation feature of object-oriented programs, and inheritance and polymorphism do not come into play (although they might have been, because perhaps this framework could be adequate also to order-sorted algebras).

Graphs labeled with alphabets equipped with preorders (i.e., reflexive and transitive binary relations) appear in (PARISI-PRESICCE; EHRIG; MONTANARI, 1986) to deal with variables within graph productions. Unification of terms can be achieved (by the rule morphism) if the terms are related by the order relation, which means that the ordering is actually a sort of variable typing mechanism. The concluding remarks of this work present some ideas on using the framework to describe inheritance, but this direction seems not having been pursuit. Furthermore, preorders on node sets appear in the very first reference to the algebraic approach to graph grammars, but have been dropped in the next publications.

Since so many structures in computer science are usually represented as graphs, and a number of other structures in the same field are adequately represented by order relations, the idea of combining the two formalisms seems to be appealing. However, this combination does not appear often in the literature. In (BRANDENBURG, 1986), for instance, "partially ordered graphs" are defined, which consist of ordinary labeled graphs together with a tree structure on their nodes. Partially ordered graph grammars are also defined, which consist of graph productions and tree productions, which must assure that the rewriting process maintains the tree structure. They are applied on lowering the complexity of the membership problem of context sensitive string grammars.

As far as we know, the first work to implement actual object-oriented features into a graph rewriting model is (FERREIRA; RIBEIRO, 2003), which was the seed to the work presented within this thesis.

5.3 Object-oriented logics and model checking

In the realm of imperative programming, Floyd and Hoare defined two of the first logics of programs ((FLOYD, 1967), and (HOARE, 1969), respectively). Since then, many formalisms, languages and systems were built upon their ideas, and addressed difficult issues such as data abstraction and concurrency. Although there is an extensive formal work on objects, logics for reasoning about object-oriented systems began to be developed quite recently. As expected, some of them are based on a Floyd-Hoare style logics (COUSOT, 1998), that concentrate on verifying pre- and postconditions and/or invariants.

A logic for the specification and verification of object-oriented programs is presented in (ABADI; LEINO, 1997). It aims the logical reasoning about pre- and postconditions of programs written in a basic object-oriented language (a variant of the calculus presented in (ABADI; CARDELLI, 1998)). Like Hoare's logic, it deals with partial correctness (correctness without termination). The language, and corresponding logic, is an object-based one, for it does not allow for the definition of classes or inheritance. It, however, allows aliasing, object self-reference, and subtyping. The logic is proved to be sound, but it is not complete (even for well typed programs).

(PIERIK; BOER, 2003) presents a Hoare logic for a sequential object-oriented language that contains all standard object-oriented features, including inheritance, subtyping and dynamic binding. The logic consists of a weakest precondition calculus for assignment and object creation, as well as Hoare rules for reasoning about method invocation with dynamic binding. The resulting logic is (relatively) complete in the sense that any valid correctness formula can be derived within the logic. This approach is a syntactical one, for it is based on an assertion language (cOOre) of a programming language abstraction level , and because the calculus has only syntactical substitution operations. Decidability issues concerning that logic are still open.

A modal logic for describing properties of terms in the object calculus of (ABADI; CARDELLI, 1998) is presented in (ANDERSEN et al., 1997). The logic is essentially the modal μ-calculus, whose fragment allows the expression of temporal modalities from the temporal logics CTL. The modal logic presented is capable of describing dynamic properties of a calculus term. Although the main focus of this paper is on the definition of a sound and complete translation from the types to logical formulae preserving typability and subtype ordering, it is interesting to note how object-oriented elements such as self-referencing, method activation, and method overriding can be described in this logic. Model checking is still not discussed here.

(DISTEFANO; KATOEN; RENSINK, 2000) presents a temporal logic, called BOTL (Object-based Temporal Logic), which is an object-based extension of CTL. The object part of that logic is inspired by by the Object Constraint Language (OCL) (WARNER; KLEPPE, 1999), an optional part of the Universal Modeling Language (UML) standard which allows expressing static properties over a class diagram in a textual way. The precise relationship between those two logics is defined by means of a mapping of a large fragment of OCL onto BOTL. BOTL is confined within object-based systems, i.e., systems in which inheritance is not considered. The combination of OCL features with temporal features facilitates both the specification of static properties and dynamic properties of object-based systems. There is no mention, however, in how model checking can be performed over BOTL models.

In (WEHRHEIM, 2003) is presented a method for computing the preserved properties of a class from any given subclass. In this work, correctness properties on classes are formalized in the temporal logic CTL (CLARKE; GRUMBERG; PELED, 1999). The investigation tries to establish which modification on the derived classes assure that the properties of the super classes are maintained (in general, since a derived class can redefine all its predecessors methods, it can destroy all their properties). The technique used computes, for every modified or new method in a class C derived from A, the set of variables it influences (directly or indirectly). Atomic propositions depending on variables within this influence set will thus potentially hold at different state in C than in A. Therefore formulas over atomic propositions of the influence set might not be inherited from A to C. All formulas independent of the influence set are preserved. The approach resembles the ones used for program slicing (HATCLIFF; DWYER; ZHENG, 2000).

Model checking was first introduced in (CLARKE; EMERSON; SISTLA, 1986), and it is a fully automatic technique to prove that a model of a concurrent program, specified as a finite transition system containing all possible behaviours of that program model, possess a property, specified in some temporal logic (EMERSON, 1998), (STIRLING, 1992). Within the last two decades, model checking has become one of the leading techniques for proving programs correct. Because of the exhaustive search performed in a (concurrent) program state-space (or in a faithful representation of it), model checkers can validate even programs consisting of a large number of interactive processes, even with very large state-spaces (BURCH et al., 1992).

The translation from formal specification languages to the languages of model-checkers has become a common practice: it is often easier to translate a model than to build a verifier to the chosen specification language.

We have presented, in this section, a number of logics developed to reason about object-oriented programs, and a number of different approaches to model checking program specification. It is not a very extensive survey, mainly because object-oriented logics and model checking techniques are beyond the scope of this work. It merely serves for illustration purposes on how active the research on object-oriented formal reasoning is. The next section presents a (more complete) survey on how logics and graph grammars are being related in research projects.

5.4 Graph grammars and model checking

(QUEMENER; JERON, 1995) describes a model checking algorithm to determine if a finite labeled transition system generated by a simple (deterministic) graph grammar is a model for a given CTL formula. They are mainly interested in infinite graphs composed by a finite number of patterns in a regular way. They express the gluing of patterns with hyperarcs, which led to an hyperedge replacement approach.

(BURKART; QUEMENER, 1996) propose an algorithm to decide whether a state of an infinite graph defined by a graph grammar satisfies a given formula of the alternating-free μ-calculus. A non standard semantics for the μ-calculus, called assertion-based semantics, is proposed, making it possible to reduce the study of the whole infinite graph to parts of it by using the correct assertions. It is an extension of (QUEMENER; JERON, 1995) and (BURKART, 1997) (where is shown that processes modeled by pushdown automata – so called pushdown processes – have a decidable verification procedure) to a decidable fragment of the μ-calculus, and deterministic hyperedge replacement is again used.

Hyperedge replacement generate the so-called context-free graph grammars. The term is a little misleading, because it does not imply that only context-free languages (in the Chomsky hierarchy sense) can be generated. Actually, all context-sensitive languages can be generated by hyperedge replacement grammars. The name is given after the similarities of the theoretical results concerning context-free Chomsky grammars and context-free graph grammars. Graph reachability, for instance, is a decidable property of context-free grammars (HABEL, 1992). The similarity between those theories can be further extended, and graphs can be parsed to see if they were generated by a given grammar.

(KOCH, 1999) presents an approach to the integration of graph transformation system and temporal logics, aiming at the specification and analysis of distributed system. Local aspects of distributed systems are modeled through ordinary single-pushout typed graph productions, while a synchronization relation is globally defined to represent which processes actions must be synchronized. The interest of that approach rely on the definition of *graph-interpreted temporal formulas*, which uses graph morphisms called assignments to keep track of what is changing in the system (and are very similar with most notions of observations found in the literature). Temporal properties can then be expressed as temporal graphs formulae, and a finite *typical model* can be built from possibly infinite models, which assures that a sound (although not complete) model checking can be performed.

(RENSINK, 2003) describes a work-in-progress and sketches a setup in which transition systems are generated from graph grammars and subsequently checked for properties expressed in a temporal logic on graphs. They propose the use of graph grammars to generate transition systems consisting of graphs as states and partial graph morphisms as transitions. An extension of the logic in (DISTEFANO; KATOEN; RENSINK, 2000) is defined, and it includes regular navigation expressions over graphs. The resulting logic is a second-order linear temporal logic, because it is possible to quantify over sets of states. It is claimed that the resulting logic can be model checked on finite graph transition systems, and the extension to unbounded states (which is the main focus of (DISTEFANO; KATOEN; RENSINK, 2000)) is left for future work.

The notion of *observation* plays a central role in most theories of concurrency. Observation can be defined in many different ways. Processes calculi, such as CCS (MILNER, 1989a) and the π-calculus (MILNER, 1989b) define it in terms of visible actions which are performed through communication channels. Tile systems (GADDUCCI; MONTANARI, 2000) also carries a notion of *interface*, which determines a sort of observational elements. (MONTEIRO, 2000) defines observation in terms of observation structures and (complete and separated) observation systems, which turn out to have equivalent categories, showing that every coalgebra gives rise to an observation system, and using this fact to propose an observation semantics for coalgebras.

A coalgebraic “loose” semantics is presented in (HECKEL et al., 2001), as a means to model open systems. Since an open system cannot be properly specified by a graph transformation system — for the environment can change without explicit control — the graph transformation rules are equipped with a “loose” semantics, in the sense that unspecified effects, which are interpreted as activities of the environment are allowed. This is formalized by the notion of double-pullback transitions, which replace the known double-pushout diagrams by allowing for spontaneous (and previously specified) changes in the context of a rule application. The loose semantics of a graph transformation system is defined as a category of coalgebras for a suitable endofunctor (actually, for the endofunctor $F(-) =$) based on the transitions of the system. Each coalgebra of such category represents a transition system G that takes no input and produces as output a transition of G. The complete, unrestricted behaviour of G is given by the final coalgebra (i.e., the full transition system containing all the finite and infinite transition sequences), as it should be expected.

The usual semantics of graph transformation system is not capable of expressing behavioural constraints. (HECKEL et al., 1997) aims to integrate graph grammars with graphical consistency and temporal logic constraints. This integration of graph grammars and behavioural constrains is done both syntactically and semantically. On the syntactical level, a notion of behavioural constraint is assumed together with a satisfaction relation for derivation sequences. On the semantical level, a coalgebraic semantics of graph grammars provides a model of the restricted behaviour of systems. The coalgebraic semantics has been described in the previous paragraph (HECKEL et al., 2001), and it is shown that the semantics can be restricted to a final coalgebra semantics for systems with behavioural constraints. The constraints themselves are represented by a graphical propositional temporal logic which allows to specify, in an axiomatic way, the effect of a production application and the order in which productions are to be applied.

The ideas of object-oriented verification were mostly taken out of (DOTTI et al., 2003). There, they use a translation from object-based graph grammars (DOTTI; RIBEIRO, 2000) to Promela to verify properties of object-based systems. Their translation is somewhat more complicated than the one we have presented here, because the object-based grammars are attributed, and a whole number of types must be taken in consideration (while here we only have to deal with a single type called “object”). Object-oriented features such as inheritance and polymorphism, however, are not treated there, and so this work is, in a sense, both a restriction and an extension of the one presented there.

6 CONCLUSIONS

6.1 Contributions

This thesis presented a graph-based formal framework to model and verify object-oriented specifications. More specifically, an extension of the algebraic single-pushout approach to (typed) graph grammars was developed, where the typing morphisms are compatible with the order relations defined over nodes and edges to represent, respectively, inheritance and overriding of classes and methods. This work is divided in three main lines: static specifications, dynamic behaviour, and formal verification of object-oriented systems.

The object-oriented class hierarchy is modeled by a graph structure called class-model graph. The novelty of that approach is the strict relation structure on nodes and edges to model (respectively) single inheritance and method overriding. The underlying relations of such sets obey additional restrictions, intended to assure that class-model graphs provide an adequate and faithful model of how object-oriented classes are actually organized. Class-model graphs are algebraic structures, and so can be related through morphisms. Composition of (object-oriented) systems is then formally defined as the colimit object of the diagram formed by the systems being composed (with a suitable morphism connecting them). It assures that composition of systems is unique (up to isomorphism) and well defined. The most common way of system extension — inheritance of classes and aggregation — are special cases of class-model graph composition, so an uniform and consistent view of it is given, since the existing ways of augmenting an object-oriented system can be all formalized by the same categorical construction.

Object-oriented graph grammars model the dynamics of object-oriented systems. The starting point are $\mathcal{C}$-typed graphs and their morphisms. $\mathcal{C}$-typed graphs are hypergraphs typed over a class-model graph, but the typing morphism is more flexible than the traditional one, in the sense that mapped hyperedges need to preserve *relations* between sources and targets. This feature adequately models inheritance, for any object can make use of inherited attributes or messages. Object-oriented graphs are restricted $\mathcal{C}$-typed graphs, in the sense that messages addressed to objects must be *correctly typed*, i.e., an object only receives inherited messages which are at the bottom of the redefinition chain it belongs to. This implements the hiding of methods which had been redefined by derived classes. Category $\mathbf{OOGraphP}(\mathcal{C})$ has object-oriented graphs as objects and $\mathcal{C}$-typed graph morphisms as arrows. Mappings between object-oriented graphs assure that subclass polymorphism can be implemented: in any place a superclass object is expected, a subclass object can appear. It is implemented by the characteristics of the morphism: a node x can be mapped to another node y if the type of y is a subclass of the type of x. $\mathcal{C}$-typed graph morphisms, for that reason, make subclass polymorphism a built-in feature of object-oriented graph grammars.

Object-oriented rules respect the principles of encapsulation and information hiding of the object-oriented paradigm. An object which is receiving a message has access only to its own attributes, although it can send messages to any object it has knowledge of (i.e., its own attributes or received message parameters). Additionally, object-oriented rule morphisms must be invertible, which means that an object cannot have its type changed during a grammar computation. Different rule structures give rise to different characteristics of computations and system graph structures. A direct derivation (or rule application) is shown to be a pushout in the category $\mathbf{OOGraphP}(\mathcal{C})$. This is a very significant result because a large body of work from the algebraic theory of graph grammars are based on derivations being (double- or single-) pushouts.

Object-oriented graph grammars, as proposed within this work, are the first development of graph grammars to take all four main features of the object-oriented paradigm — encapsulation, data hiding, inheritance, and polymorphism — into their structure.

An observational semantics for object-oriented graph grammars is also developed. This semantics, based on a labeled transition system, is more abstract than the usual one, because it only keeps information about which rule was applied and the object the rule was applied to. This semantics is based on a notion of visible entities (objects or messages), which are the elements we are interested in for verification purposes.

Finally, a formal translation from object-oriented graph grammar specifications into Promela programs was presented. Objects in the system graph are modeled as Promela processes, having as parameters the object attributes; message exchange is implemented through buffered communication channels. Semantics of grammar rule application is preserved by the nondeterministic choice of which message to consume. Inheritance, polymorphism and dynamic binding are implemented in the Promela program, which has no support for object-oriented features whatsoever. The translation presented assures that both state and event verification can be performed, which is an extension on the way SPIN performs verification.

Semantic compatibility between the behaviour of the original object-oriented graph grammar and the translated Promela program is discussed. We compare object-oriented graph grammars computations with Promela translated programs executions, and argue that for any *path* existing in one labeled transition systems there exists a correspondent path in the other. The resulting translated program can be fed into the model checker SPIN, and properties can be defined over the actual objects of the original grammar or over events such as rule applications over determined objects. Verification is then performed automatically by the model checker.

We believe that all the intended goals for this thesis (initially presented in Section 1.5) were met.

6.2 Future work

Research projects seldom are completely finished. Usually, there is a number of different new results that can be further achieved. The more theoretical the research is, the more this fact is true. There are two distinct possible developments of this thesis. The first kind are extensions closely related to the work presented here. The second kind are more general paths that could be followed by research.

There are a number of issues that have not being developed in this thesis. First, object creation and deletion cannot be translated, since there are no translation rules for that. It is a simple extension of the proposed translation, but we believe that it must be done if the formalism is to be used as a serious form of object-oriented software modeling and development. When it is done, it will be interesting to investigate its impact in the verification process. Unbounded creation of objects make verification a harder issue, since there are a number of things that become undecidable. Additionally, research on using graphs to model dynamic allocated data structures can also be carried out.

Another extension is the automatization of the object-oriented graph grammar $\rightarrow$ Promela program translating process. There is a current ongoing effort in the graph transformation community to establish a formalism for exchanging graph formats (WINTER; KULLBACH; RIEDIGER, 2001). XML-based exchange formats for graphs and graph transformation systems are currently under development, which would make easier the task of constructing tools for model building and model translation. Using a standard XML format for graphs and graph rules would make the translation process algorithmically feasible and straightforward.

A formal semantic-based proof that the provided translation is correct is also necessary. First, a definition of the Promela language semantics is required, so it can be compared against the semantics of object-oriented graph grammars. This would be also an important contribution to the users of SPIN who would have a formal (and complete) semantic definition at hand. If the semantics can be implemented in a theorem prover, the better.

Temporal logics such as CTL or LTL are theoretically elegant, but sometimes developers and even researchers find it difficult to use them to accurately express a complex state or event sequencing properties often needed in software verification. Once written, temporal formulas are frequently hard to reason about, debug, and modify. Therefore, it would be interesting for the user to express graph properties themselves in terms of graphs. A friendly interface for the description of graph properties (including object-oriented properties) and the corresponding translation to temporal logics formulae would complete this work.

Chapter 2 provides a definition for system composition. A possible development for this work is to provide a semantics for object-oriented graph grammars composition, such is done in (RIBEIRO, 1996) for typed graph grammars. In order to do that, suitable morphisms of object-oriented graph grammars must be defined, in such manner that derivations of the composed system are meaningful respecting the composition process. Derivations are essentially the computations of a graph grammar, when regarded as a computational formalism. As pointed out by (CORRADINI et al., 1993), since the pushout object of two arrows is unique up to isomorphism, the application of a production to a graph can produce an unbounded number of different results. This fact is highly counter-intuitive, because in the above situation one would expect a deterministic result, or, at most, a finite set of possible outcomes. Indeed, in the algebraic approach to graph grammars, one usually considers a concrete graph as a specific representation of a "system state", and since any kind of abstract semantics should be representation independent, one handles (more or less explicitly) *abstract graphs*, i.e., isomorphism classes of concrete graphs: with this choice, a direct derivation becomes clearly deterministic. If the semantics we are interested in associates with each grammar all its possible derivations, we must reason also in terms of *abstract derivations*, i.e., equivalence classes of derivations with respect to a suitable equivalence. However, because of inheritance, polymorphism and method redefinition, what constitutes an equivalence class of object-oriented graphs, of an equivalence class of object-oriented systems derivations can differ considerably from other systems. Investigating how those equivalence classes should be constructed, and how those results can be related to the ones already obtained for categories of concrete and abstract derivations is another goal of research.

Other interesting aspects of system comparison are simulations and bisimulations (MILNER, 1989b), (RUTTEN, 2000). Different definitions of bisimulations (such as strong or weak bisimulation) can be given, but generally two systems are (weak) bisimilar if they present the same visible behaviour. The object-oriented programming paradigm is perhaps the one in which this notion has the most meaning: object-oriented programs implement abstract data types, in which we are interested in *what* the system does, and not in *how* it is implemented. So, we are only interested in some object behaviour, which can be defined in terms of applicable rules to it. However, a trace of applicable rules in a system does not provide the behaviour of a single object, because rules applied to object attributes must be meaningless respecting the behaviour definition. Because we define classes as being visible or invisible, and since transitions over invisible messages or objects are labeled as "silent" transitions, we have the right framework to begin investigating bisimulations between objects/systems. A meaningful notion of object bisimulation, and system bisimulation, can be used to investigate properties of programs and types, as in (FIORE, 1993), (SEWELL, 2002), or (SUMII; PIERCE, 2005).

Expressiveness of object-oriented graph grammars are also worth a deep investigation. Inheritance was used in the Dining Philosophers problem example of Chapter 4 to tell elements of different types apart. This is a sign that inheritance might extend the power of expression of graph grammars. Attributed grammars (where nodes or edges can be labeled with elements of some algebra) are more expressive than standard graph grammars. Object-oriented graph grammars, however, appear to be as expressive as attributed grammars with finite algebraic sorts. Further investigation is needed to prove (or disprove) that claim.

Object-oriented graph grammars define a model of computation compatible with the one provided by object-oriented programs. A new computational model, or a new feature in a traditional one, usually is reflected in a new family of programming languages, and new paradigms of software development. The graphs and grammars presented within this thesis can serve as a syntactical and semantical basis for new visual object-oriented declarative languages. Domain specific visual languages, and language generators, also can be developed having object-oriented graph grammars as basis.

REFERENCES

ABADI, M.; CARDELLI, L. **A Theory of Objects**. [S.l.]: Springer, 1998. 396p.

ABADI, M.; LEINO, K. R. M. A Logic of Object-Oriented Programs. In: THEORY AND PRACTICE OF SOFTWARE DEVELOPMENT, TAPSOFT, 1997. **Proceedings...** Berlin: Springer-Verlag, 1997. p.682–696. (Lecture Notes in Computer Science, v.1214).

ABRAMSKY, S.; JUNG, A. Domain Theory. In: ABRAMSKY, S.; GABBAY, D. M.; MAIBAUM, T. S. E. (Ed.). **Handbook of Logic in Computer Science**. Oxford: Oxford Science Publications, 1994. v.3.

ALUR, R.; BRAYTON, R. K.; HENZINGER, T. A.; QADEER, S.; RAJAMANI, S. K. Partial-Order Reduction in Symbolic State-Space Exploration. **Formal Methods in System Design**, [S.l.], v.18, p.97–116, 2001.

ANDERSEN, D. S.; PEDERSEN, L. H.; HÜTTEL, H.; KLEIST, J. Objects, Types and Modal Logics. In: FOUNDATIONS OF OBJECT-ORIENTED LANGUAGES, FOOL, 1997. **Proceedings...** [S.l.: s.n.], 1997.

ANDERSON, R. J.; BEAME, P.; BURNS, S.; CHAN, W.; MODUGNO, F.; NOTKIN, D.; REESE, J. **Model Checking Large Software Specifications**. Seattle: Department of Computer Science and Engineering, University of Washington, 1996. 12p. (Technical Report 96-04-02).

ANDRIES, M.; ENGELS, G.; HABEL, A.; HOFFMANN, B.; KREOWSKI, H.-J.; KUSKE, S.; PLUMP, D.; SCHURR, A.; TAENTZER, G. Graph transformation for specification and programming. **Science of Computer Programming**, [S.l.], v.34, p.1–54, 1999.

ARAÚJO JR., J.; SAWYER, P. Integrating Object-Oriented Analysis and Formal Specification. **Journal of the Brazilian Computer Society**, [S.l.], v.5, n.1, 1998.

ASPERTI, A.; LONGO, G. **Categories, Types and Structures**. Cambridge, MA: MIT Press, 1991.

BARESI, L.; PEZZÈ, M. Can Graph Grammars make Formal Methods more Human? In: WORKSHOP ON GRAPH TRANSFORMATION AND VISUAL MODELING TECHNIQUES, 2000, Geneva (Switzerland). **Proceedings...** [S.l.: s.n.], 2000. p.387–394.

BARNES, J. **Programming in Ada 95**. 2nd.ed. [S.l.]: Addison-Wesley Professional, 1998. 720p.

BAUDERON, M.; JACQUET, H. Node rewriting in graphs and hypergraphs: a categorical framework. **Theoretical Computer Science**, [S.l.], v.266, p.463–487, 2001.

BERGSTRA, J. A.; HEERING, J.; KLINT, P. **Algebraic Specification**. New York: ACM Press, 1989. 397p.

BEVIER, W. R. Toward an Operational Semantics of PROMELA in ACL2. In: SPIN WORKSHOP, 3., 1997, The Nederlands. **Proceedings...** [S.l.: s.n.], 1997. 20p.

BIBERSTEIN, O.; BUCHS, D.; GUELFI, N. CO-OPN/2: a concurrent object-oriented formalism. In: INTERNATIONAL WORKSHOP ON FORMAL METHODS FOR OPEN OBJECT-BASED DISTRIBUTED SYSTEMS, FMOODS, 1997. **Proceedings...** London: Chapman & Hall, 1997. v.2, p.57–72.

BIBERSTEIN, O.; BUCHS, D.; GUELFI, N. Object-Oriented Nets with Algebraic Specifications: the CO-OPN/2 formalism. In: AGHA, G. A.; CINDIO, F. D.; ROZENBERG, G. (Ed.). **Concurrent Object-Oriented Programming and Petri Nets**: advances in Petri nets. Berlin: Springer, 2001. p.73–130. (Lecture Notes in Computer Science, v.2001).

BJORNES, D.; JONES, G. B. **The Vienna Development Method**: the meta-language. Berlin: Springer-Verlag, 1978. (Lecture Notes in Computer Science, v.61).

BLOSTEIN, D.; FAHMY, H.; GRBAVEC, A. **Practical Use of Graph Rewriting**. Kingston, Ontario, Canada: Queen's University, 1995. (95-373).

BORCEUX, F. **Handbook of Categorical Algebra 1 – Basic Category Theory**. Cambridge, UK: Cambridge University Press, 1994. 345p. (Encyclopedia of Mathematics and its Applications, v.50).

BRANDENBURG, F. J. On Partially Ordered Graph Grammars. In: INTERNATIONAL WORKSHOP ON GRAPH GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 3., 1986, Warrenton, Virginia, USA. **Proceedings...** Berlin: Springer-Verlag, 1986. p.99–111. (Lecture Notes in Computer Science, v.291).

BREU, R. **Algebraic Specification Techniques in Object Oriented Programming Environments**. Berlin: Springer-Verlag, 1991. 228p. (Lecture Notes in Computer Science, v.562).

BRUCE, K. B.; FIECH, A.; PETERSEN, L. Subtyping is not a good match for object-oriented languages. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, ECOOP, 1997. **Proceedings...** Berlin: Springer-Verlag, 1997. p.104–127. (Lecture Notes in Computer Science, v.1241).

BURCH, J. R.; CLARKE, E. M.; MCMILLAN, K. L.; DILL, D. L.; HWANG, L. J. Symbolic model checking 10^{20} states and beyond. **Information and Computation**, [S.l.], v.98, n.2, p.142–170, June 1992.

BURKART, O. **Automatic Verification of Sequential Infinite-State Processes**. Berlin: Springer, 1997. 163p. (Lecture Notes in Computer Science, v.1354).

BURKART, O.; QUEMENER, Y.-M. **Model-Checking of Infinite Graphs Defined by Graph Grammars**. Rennes: IRISA — Institut de Recherche en Informatique et Systèmes Aléatoires, 1996. (Technical Report, 995).

BURSTALL, R.; GOGUEN, J. An Informal Introduction to Specifications using Clear. In: BOYER, R.; MOORE, J. (Ed.). **The Correctness Problem in Computer Science**. [S.l.]: Academic Press, 1981. p.185–213.

CAMPIONE, M.; WALRATH, K.; HUML, A. **The Java(TM) Tutorial**. 3rd.ed. Upper Saddle River: Addison-Wesley, 2000. 580p.

CARDELLI, L.; WEGNER, P. On understanding types, data abstraction and polymorphism. **ACM Computing Surveys**, [S.l.], v.17, n.4, p.471–522, 1985.

CIRSTEA, C. On Expressivity and Compositionality in Logics for Coalgebras. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.82, n.1, p.1–18, 2003.

CLARK, K. L.; TAERNLUND, S.-A. **Logic programming**. London: Academic Press, 1982. 366p.

CLARKE, E. M.; EMERSON, E. A. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In: LOGIC OF PROGRAMS WORKSHOP, 1981, Yorktown Heights, NY. **Proceedings...** Berlin: Springer-Verlag, 1981. p.244–263. (Lecture Notes in Computer Science, v.131).

CLARKE, E. M.; EMERSON, E. A.; SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. **ACM Transactions on Programming Languages & Systems**, [S.l.], v.8, n.2, p.244–263, April 1986.

CLARKE, E. M.; GRUMBERG, O.; PELED, D. A. **Model Checking**. Cambridge, MA: MIT Press, 1999. 387p.

CONRAD, S.; GOGOLLA, M.; HERZIG, R. **Troll Light**: a core language for specifying objects. Braunschweig: [s.n.], 1992.

COOK, W. R. Object-oriented programming versus abstract data type. In: BAKKER, J. W. de; ROEVER, W. P. de; ROZEMBERG, G. (Ed.). **Foundations of Object-Oriented Languages**. Berlin: Springer, 1990. p.151–178. (Lecture Notes in Computer Science, v.489).

COOK, W. R.; HILL, W. L.; CANNING, P. S. Inheritance is not subtyping. In: ANNUAL ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, POPL, 17., 1989. **Proceedings...** New York: ACM Press, 1989. p.125–135.

CORRADINI, A.; EHRIG, H.; LÖWE, M.; MONTANARI, U.; PADBERG, J. The Category of Typed Graph Grammars and its Adjunctions with Categories of Derivations. In: INTERNATIONAL WORKSHOP ON GRAPH GRAMMARS

AND THEIR APPLICATION TO COMPUTER SCIENCE, 5., 1994, Williamsburg. **Proceedings...** Berlin: Springer-Verlag, 1994. p.56–74. (Lecture Notes in Computer Science, v.1073).

CORRADINI, A.; EHRIG, H.; LÖWE, M.; MONTANARI, U.; ROSSI, F. Abstract Graph Derivations in the Double Pushout Approach. In: INTERNATIONAL WORKSHOP ON GRAPH TRANSFORMATIONS IN COMPUTER SCIENCE, 5., 1993, Dagstuhl Castle, Germany. **Proceedings...** Berlin: Springer-Verlag, 1993. p.86–103. (Lecture Notes in Computer Science, v.776).

CORRADINI, A.; MONTANARI, U.; ROSSI, F. Graph Processes. **Fundamentae Informatica**, [S.l.], v.26, n.3-4, p.241–265, 1996.

COUSOT, P. Methods and Logics for Proving Programs. In: LEEUWEN, J. van (Ed.). **Handbook of Theoretical Computer Science**. [S.l.]: Elsevier, 1998. v.2.

CSK CORPORATION. **The VDM++ Language**. [S.l.]: CSK Corporation, 2005. 192p. Available at <http://www.vdmbook.com/langmanpp_a4_001.pdf>. Visited in May, 2005.

DAVEY, B. A.; PRIESTLEY, H. A. **Introduction to Lattices and Order**. 2nd.ed. Cambridge: Cambridge University Press, 2002. 298p.

DIJKSTRA, E. W. Guarded commands, nondeterminacy and formal derivation of programs. **Communications of the ACM**, New York, v.18, n.8, p.453–457, August 1975.

DISTEFANO, D.; KATOEN, J.-P.; RENSINK, A. On a Temporal Logic for Object-Based Systems. In: INTERNATIONAL CONFERENCE ON FORMAL METHODS FOR OPEN OBJECT-BASED DISTRIBUTED SYSTEMS, FMOODS, 2000. **Proceedings...** [S.l.]: Kluwer Academic Publishers, 2000. p.305–326.

DOTTI, F. L.; FOSS, L.; RIBEIRO, L.; SANTOS, O. M. Verification of Object-based Distributed Systems. In: INTERNATIONAL CONFERENCE ON FORMAL METHODS FOR OPEN OBJECT-BASED DISTRIBUTED SYSTEMS, FMOODS, 2003, Paris. **Proceedings...** Berlin: Springer-Verlag, 2003. p.261–275. (Lecture Notes in Computer Science, v.2884).

DOTTI, F. L.; RIBEIRO, L. Specification of mobile code using graph grammars. In: INTERNATIONAL CONFERENCE ON FORMAL METHODS FOR OPEN OBJECT-BASED DISTRIBUTED SYSTEMS, FMOODS, 2000. **Proceedings...** [S.l.]: Kluwer Academic Publishers, 2000. p.45–64.

DWYER, M. B.; HATCLIFF, J.; JOEHANES, R.; LAUBACH, S.; PASAREANU, C. S.; ZHENG, H.; VISSER, W. Tool-supported program abstraction for finite-state verification. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 23., 2001, Toronto, Ontario, Canada. **Proceedings...** [S.l.]: IEEE Computer Society, 2001. p.177–187.

EHRICH, H.-D.; SERNADAS, A.; SERNADAS, C. Objects, Object Types, and Object Identification. In: CATEGORICAL METHODS IN COMPUTER SCIENCE, 1987. **Proceedings...** Berlin: Springer-Verlag, 1987. p.142–156. (Lecture Notes in Computer Science, v.393).

EHRIG, H.; ENGELS, G.; KREOWSKI, H.-J.; ROZENBERG, G. **Handbook of Graph Grammars and Computing by Graph Transformation**. Singapore: World Scientific, 1997. v.2. 698p.

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. In: EHRIG, H.; KREOWSKI, H.-J.; MONTANARI, U.; ROZEMBERG, G. (Ed.). **Handbook of Graph Grammars and Computing by Graph Transformation**. Singapore: World Scientific, 1996. v.1, p.247–312.

EHRIG, H.; KREOWSKI, H.-J.; MONTANARI, U.; ROZEMBERG, G. **Handbook of Graph Grammars and Computing by Graph Transformation**. Singapore: World Scientific, 1996. v.1.

EHRIG, H.; KREOWSKI, H.-J.; MONTANARI, U.; ROZEMBERG, G. **Handbook of Graph Grammars and Computing by Graph Transformation**. Singapore: World Scientific, 1999. v.3.

EHRIG, H.; LÖWE, M. Parallel and distributed derivations in the single-pushout approach. **Theoretical Computer Science**, [S.l.], v.109, p.123–143, 1993.

EHRIG, H.; PFENDER, M.; SCHNEIDER, H. J. Graph grammars: an algebraic approach. In: ANNUAL IEEE SYMPOSIUM ON SWITCHING AND AUTOMATA THEORY, 14., 1973. **Proceedings...** [S.l.: s.n.], 1973. p.167–180.

EHRIG, H.; ROSEN, B. Parallelism and concurrency of graph manipulations. **Theoretical Computer Science**, [S.l.], v.11, p.247–275, 1980.

EMERSON, E. A. Temporal and Modal Logic. In: LEEUWEN, J. van (Ed.). **Handbook of Theoretical Computer Science**. Amsterdam: Elsevier, 1998. v.2, p.995–1072.

ENGELFRIET, J.; ROZENBERG, G. Node Replacement Graph Grammars. In: EHRIG, H.; KREOWSKI, H.-J.; MONTANARI, U.; ROZEMBERG, G. (Ed.). **Handbook of Graph Grammars and Computing by Graph Transformation**. Singapore: World Scientific, 1997. v.1.

FERREIRA, A. P. L.; RIBEIRO, L. Towards object-oriented graphs and grammars. In: INTERNATIONAL CONFERENCE ON FORMAL METHODS FOR OPEN OBJECT-BASED DISTRIBUTED SYSTEMS, FMOODS, 2003, Paris. **Proceedings...** Berlin: Springer-Verlag, 2003. p.16–31. (Lecture Notes in Computer Science, v.2884).

FERREIRA, A. P. L.; RIBEIRO, L. Derivations in object-oriented graph grammars. In: INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATIONS, ICGT, 2., 2004, Rome. **Proceedings...** Berlin: Springer-Verlag, 2004. p.416–430. (Lecture Notes in Computer Science, v.3256).

FERREIRA, A. P. L.; RIBEIRO, L. A graph-based semantics for object-oriented programming constructs. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.122, p.89–104, 2005.

FIECH, A. Colimits in the Category DCPO. **Mathematical Structures in Computer Science**, [S.l.], v.6, n.5, p.455–468, 1996.

FIORE, M. A coinduction principle for recursive data types based on bisimulation. In: IEEE SYMPOSIUM ON LOGIC IN COMPUTER SCIENCE, 8., 1993. **Proceedings...** [S.l.: s.n.], 1993.

FITTING, M. **First-order logic and automated theorem proving**. 2nd.ed. New York: Springer, 1996. 326p.

FITZGERALD, J.; LARSEN, P. G.; MUKHERJEE, P.; PLAT, N.; VERHOEF, M. **Validated Designs for Object-oriented Systems**. New York: Springer, 2005. 404p.

FLOYD, R. W. Assigning meanings to programs. In: SYMPOSIUM OF APPLIED MATH, 1967. **Proceedings...** [S.l.]: American Mathematical Society, 1967. v.19, p.19–32.

BRAUER, W.; ROZENBERG, G.; SALOMAA, A. (Ed.). **Introduction to Process Algebra**. Berlin: Springer, 2000. 163p.

GADDUCCI, F. **On the Algebraic Approach to Concurrent Term Rewriting**. 1996. PhD Thesis — Dipartimento di Informatica, Università di Pisa, Pisa, Italy.

GADDUCCI, F.; MONTANARI, U. The Tile Model. In: PLOTKIN, G.; STIRLING, C.; TOFTE, M. (Ed.). **Proof, Language and Interaction**: essays in honour of robin milner. Cambridge, MA: MIT Press, 2000. p.133–166.

GENRICH, H. J.; JANSSENS, D.; ROZENBERG, G.; THIAGARAJAN, P. S. Petri Nets and their relation to Graph Grammars. In: INTERNATIONAL WORKSHOP ON GRAPH GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 2., 1983. **Proceedings...** Berlin: Springer-Verlag, 1983. p.115–129. (Lecture Notes in Computer Science, v.153).

GHEZZI, C.; JAZAYERI, M. **Programming language concepts**. 3rd.ed. New York: John Wiley & Sons, 1998. 427p.

GOGUEN, J. A.; MALCOM, G. **Algebraic semantics of imperative programs**. London: MIT Press, 1996. 228p.

HABEL, A. **Hyperedge Replacement**: grammars and languages. Berlin: Springer-Verlag, 1992. 214p. (Lecture Notes in Computer Science, v.643).

HABEL, A.; KREOWSKI, H.-J. On Context-Free Graph Languages Generated by Edge Replacement. In: INTERNATIONAL WORKSHOP ON GRAPH GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 2., 1982, Haus Ohrbeck, Germany. **Proceedings...** Berlin: Springer-Verlag, 1982. p.143–158. (Lecture Notes in Computer Science, v.153).

HATCLIFF, J.; DWYER, M. B.; ZHENG, H. Slicing Software for Model Construction. **Higher-Order and Symbolic Computation**, [S.l.], v.13, n.4, p.315–353, December 2000.

HE, X.; DING, Y. Object Orientation in Hierarchical Predicate Transition Nets. In: AGHA, G. A.; CINDIO, F. D.; ROZENBERG, G. (Ed.). **Concurrent Object-Oriented Programming and Petri Nets**: advances in Petri nets. Berlin: [S.l.]: Springer, 2001. p.196–215. (Lecture Notes in Computer Science, v.2001).

HECKEL, R. **Open Graph Transformation Systems**: a new approach to the compositional modelling of concurrent and reactive systems. 1998. PhD Thesis — Technische Universität Berlin, Berlin.

HECKEL, R.; EHRIG, H.; WOLTER, U.; CORRADINI, A. Integrating the specification techniques of graph transformation and temporal logic. In: INTERNATIONAL SYMPOSIUM ON MATHEMATICAL FOUNDATIONS OF COMPUTER SCIENCE, 22., 1997. **Proceedings...** Berlin: Springer-Verlag, 1997. p.219–228. (Lecture Notes in Computer Science, v.1295).

HECKEL, R.; EHRIG, H.; WOLTER, U.; CORRADINI, A. Double-pullback transitions and loose semantics of graph transformation systems. **Applied Categorical Structures**, [S.l.], v.9, p.83–110, 2001.

HENSEL, U.; HUISMAN, M.; JACOBS, B.; TEWS, H. Reasoning about classes in object-oriented languages: logical models and tools. In: EUROPEAN SYMPOSIUM ON PROGRAMMING, 1998. **Proceedings...** Berlin: Springer-Verlag, 1998. p.105–121. (Lecture Notes in Computer Science, v.1381).

HOARE, C. A. R. An axiomatic basis for computer programming. **Communications of the ACM**, [S.l.], v.12, p.576–583, 1969.

HOARE, C. A. R. **Communicating Sequential Processes**. USA: Prentice Hall, 1985.

HOLZMANN, G. J. **Design and validation of computer protocols**. [S.l.]: Prentice Hall, 1991. 500p.

HOLZMANN, G. J. The Model Checker SPIN. **IEEE Transactions on Software Engineering**, [S.l.], v.23, n.5, p.1–17, May 1997.

HOPCROFT, J. E. **Formal languages and their relation to automata**. Reading: Addison-Wesley, 1969. 242p.

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. **Introduction to automata theory, languages, and computation**. 2nd.ed. Boston: Addison-Wesley, 2001. 521p.

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

INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATION, 1., 2002, Barcelona, Spain. **Proceedings...** Berlin: Springer-Verlag, 2002. (Lecture Notes in Computer Science, v.2505).

INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATION, 2., 2004, Rome, Italy. **Proceedings...** Berlin: Springer-Verlag, 2004. (Lecture Notes in Computer Science, v.3256).

INTERNATIONAL WORKSHOP ON GRAPH GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 2., 1982, Haus Ohrbeck, Germany. **Proceedings...** Berlin: Springer-Verlag, 1982. (Lecture Notes in Computer Science, v.153).

INTERNATIONAL WORKSHOP ON GRAPH GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 3., 1986, Warrenton, Virginia, USA. **Proceedings...** Berlin: Springer-Verlag, 1986. (Lecture Notes in Computer Science, v.291).

INTERNATIONAL WORKSHOP ON GRAPH GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 4., 1990, Bremen, Germany. **Proceedings...** Berlin: Springer-Verlag, 1990. (Lecture Notes in Computer Science, v.532).

INTERNATIONAL WORKSHOP ON GRAPH GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 5., 1994, Williamsburg. **Proceedings...** Berlin: Springer-Verlag, 1994. (Lecture Notes in Computer Science, v.1073).

INTERNATIONAL WORKSHOP ON GRAPH GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE AND BIOLOGY, 1979, Bad Honnef, Germany. **Proceedings...** Berlin: Springer-Verlag, 1979. (Lecture Notes in Computer Science, v.73).

INTERNATIONAL WORKSHOP ON GRAPH TRANSFORATIONS IN COMPUTER SCIENCE, 1993, Dagstuhl Castle, Germany. **Proceedings...** Berlin: Springer-Verlag, 1993. (Lecture Notes in Computer Science, v.1073).

INTERNATIONAL WORKSHOP ON THEORY AND APPLICATION OF GRAPH TRANSFORMATIONS, 6., 1998, Paderborn, Germany. **Proceedings...** Berlin: Springer-Verlag, 1998. (Lecture Notes in Computer Science, v.1764).

JACOBS, B. Object and Classes, Coalgebraically. In: FREITAG, B.; JONES, C. B.; LENGAUER, C.; SCHEK, H.-J. (Ed.). **Object-Orientation with Parallelism and Persistence**. [S.l.]: Kluwer Academics Publishers, 1996. p.83–103.

JACOBS, B. Coalgebraic Reasoning about Classes in Object-Oriented Languages. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.11, p.1–12, 1998.

JACOBS, B. Towards a Duality Result in the Modal Logic of Coalgebras. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.33, p.1–29, 2000.

JACOBS, B. The Temporal Logics of Coalgebras via Galois Algebras. **Mathematical Structures in Computer Science**, [S.l.], v.12, p.875–903, 2002.

JACOBS, B.; RUTTEN, J. J. M. M. A Tutorial on (Co)Algebras and (Co)Induction. **EATCS Bulletin**, [S.l.], v.62, p.222–259, 1997.

KAUFMANN, M.; MANOLIOS, P.; MOORE, J. S. **Computer-Aided Reasoning – ACL2 Case Studies**. Norwell, MA: Kluwer Academic Publishers, 2000. 337p.

KENNAWAY, R. Infinitary Rewriting and Cyclic Graphs. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.2, 1995. 14p.

KERNINGHAN, B.; RITCHIE, D. **The C Programming Language**. [S.l.]: Addison-Wesley, 1986.

KOCH, M. **Integration of Graph Transformation and Temporal Logic for the Specification of Distributed Systems**. 1999. PhD Thesis — Technische Universität Berlin, Berlin.

KORFF, M. **Generalized Graph Structure Grammars with Applications to Concurrent Object-Oriented Systems**. 1995. PhD Thesis — Technische Universität Berlin, Berlin.

KORFF, M.; RIBEIRO, L. Formal Relationship between Graph Grammars and Petri Nets. In: INTERNATIONAL WORKSHOP ON GRAPH GRAMMARS AND THEIR APPLICATIONS TO COMPUTER SCIENCE, 5., 1996. **Proceedings...** Berlin: Springer-Verlag, 1996. p.288–303. (Lecture Notes in Computer Science, v.1073).

KOSKO, B. **Neural networks and fuzzy systems**: a dynamical systems approach to machine intelligence. Englewood Cliffs: Prentice-Hall, 1992. 449p.

KRINKE, J. Context-sensitive slicing of concurrent programs. In: ACM SIGSOFT SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING/EUROPEAN SOFTWARE ENGINEERING CONFERENCE, 2003, Helsinki, Finland. **Proceedings...** [S.l.: s.n.], 2003. p.178–187.

KURZ, A. Specifying coalgebras with modal logic. **Theoretical Computer Science**, [S.l.], v.260, p.119–138, 2001.

LAKOS, C. Object-oriented modeling with Object Petri Nets. In: AGHA, G. A.; CINDIO, F. D.; ROZENBERG, G. (Ed.). **Concurrent Object-Oriented Programming and Petri Nets**: advances in Petri nets. Berlin: Springer, 2001. p.1–37. (Lecture Notes in Computer Science, v.2001).

LEWIS, H. R.; PAPADIMITRIOU, C. H. **Elements of the Theory of Computation**. 2nd.ed. Upper Saddle River: Prentice Hall, 1998. 361p.

LISCHNER, R. **Delphi in a Nutshell**. [S.l.]: O'Reilly, 2000. 576p.

LORETO, A. B.; RIBEIRO, L.; TOSCANI, L. V. Decodability and tractability of a problem in object-based graph grammars. In: IFIP WORLD COMPUTER CONGRESS - THEORETICAL COMPUTER SCIENCE, 17., 2002, Montreal. **Proceedings...** [S.l.]: Kluwer, 2002.

LÖWE, M. **Extended Algebraic Graph Transformation**. 1991. Tese (Doutorado em Ciência da Computação) — Technischen Universität Berlin, Berlin.

MACLANE, S. **Categories for the Working Mathematician**. 2nd.ed. New York: Springer, 1998. 314p.

MACLENNAN, B. J. **Principles of programming languages** : design, evaluation, and implementation. 3rd.ed. New York: Oxford University, 1999. 509p.

MARTIN, J. C. **Introduction to Languages and the Theory of Computation**. 2nd.ed. [S.l.]: WCB/McGraw-Hill, 1996. 450p.

MEYER, B. **Eiffel** : the language. [S.l.]: Prentice Hall, 1991. 300p.

MICROSOFT CORPORATION. **C# Language Specification**. Seattle, WA: Microsoft Corporation, 2005. Available at <http://msdn.microsoft.com/>. Visited in April, 2003.

MILNER, R. **A Calculus of Communicating Systems**. Berlin: Springer-Verlag, 1989.

MILNER, R. **Communication and Concurrency**. New York: Prentice-Hall, 1989. 260p.

MILNER, R. **Communicating and Mobile Systems** : the π-calculus. Cambridge: Cambridge University Press, 1999. 161p.

MITCHELL, J. C.; APT, K. **Concepts in Programming Languages**. Cambridge: Cambridge University Press, 2001. 450p.

MITCHELL, J.; VISWANATHAN, R. Effective models of polymorphism, subtyping and recursion. In: INTERNATIONAL COLLOQUIUM ON AUTOMATA, LANGUAGES, AND PROGRAMMING, 23., 1996. **Proceedings...** Berlin: Springer-Verlag, 1996.

MONIN, J.-F. **Understanding Formal Methods**. London: Springer, 2003. 275p.

MONTANARI, U.; PISTORE, M.; ROSSI, F. Modeling Concurrent, Mobile and Coordinated Systems via Graph Transformations. In: EHRIG, H.; KREOWSKI, H.-J.; MONTANARI, U.; ROZEMBERG, G. (Ed.). **Handbook of Graph Grammars and Computing by Graph Transformation**. Singapore: World Scientific, 1999. v.3, p.189–268.

MONTEIRO, L. Observation Systems. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.33, p.261–275, 2000.

MONTEIRO, L.; PORTO, A. Syntactic and Semantic Inheritance in Logic Programming. In: SEMINAR AND WORKSHOP ON DECLARATIVE PROGRAMMING, 1991. **Proceedings...** London: Springer-Verlag, 1991. p.163–173.

MOSBAH, M. Probabilistic Hyperedge Replacement Grammars. **Theoretical Computer Science**, [S.l.], v.159, p.81–102, 1996.

MUHAMMAD, H. H.; FERREIRA, A. P. L. Exploração de reflexão computacional através de um modelo de objetos sem classes. **Revista Eletrônica de Iniciação Científica**, [S.l.], v.3, n.3, p.1–15, 2003.

NAGL, M. Set Theoretic Approaches to Graph Grammars. In: INTERNATIONAL WORKSHOP ON GRAPH GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 3., 1986, Warrenton, Virginia, USA. **Proceedings...** Berlin: Springer-Verlag, 1986. p.41–54. (Lecture Notes in Computer Science, v.291).

NATARAJAN, V.; HOLZMANN, G. J. Outline for an Operational Semantics Definition of PROMELA. In: SPIN WORKSHOP, 2., 1996, NJ, USA. **Proceedings...** [S.l.: s.n.], 1996. p.175–192.

NIKOLOPOULOS, C. **Expert Systems**. New York: Marcel Dekker, 1997. 331p.

NYGAARD, K.; DAHL, O.-J. Simula 67. In: WEXELBLAT, R. W. (Ed.). **History of Programming Languages**. [S.l.]: ACM Press, 1981.

PAPADOPOULOS, G. A. Concurrent object-oriented programming using term graph rewriting techniques. **Information and Software Technology**, [S.l.], n.38, p.539–547, 1996.

PARISI-PRESICCE, F.; EHRIG, H.; MONTANARI, U. Graph Rewriting with Unification and Composition. In: INTERNATIONAL WORKSHOP ON GRAPH GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 3., 1986, Warrenton, Virginia, USA. **Proceedings...** Berlin: Springer-Verlag, 1986. p.496–514. (Lecture Notes in Computer Science, v.291).

PATTINSON, D. Semantical principles in the modal logics for coalgebras. In: INTERNATIONAL SYMPOSIUM ON THEORETICAL ASPECTS OF COMPUTER SCIENCE, STACS, 18., 2001. **Proceedings...** Berlin: Springer-Verlag, 2001. (Lecture Notes in Computer Science, v.2010).

PFALTZ, J.; ROSENFELD, A. Web grammars. In: INTERNATIONAL JOINT CONFERENCES ON ARTIFICIAL INTELLIGENCE, 1969. **Proceedings...** [S.l.: s.n.], 1969. p.609–619.

PIERCE, B. C. **Basic category theory for computer scientists**. Cambridge, MA: MIT Press, 1991. 101p.

PIERIK, C.; BOER, F. S. de. A syntax-directed Hoare logic for object-oriented programming constructs. In: INTERNATIONAL CONFERENCE ON FORMAL METHODS FOR OPEN OBJECT-BASED DISTRIBUTED SYSTEMS, FMOODS, 2003, Paris. **Proceedings...** Berlin: Springer-Verlag, 2003. p.64–78. (Lecture Notes in Computer Science, v.2884).

PLAT, N.; LARSEN, P. G. An Overview of the ISO/VDM-SL Standard. **ACM SIGPLAN Notices**, [S.l.], v.27, n.8, p.76–82, 1992.

PNUELI, A. The Temporal Logic of Programs. In: IEEE SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE, 18., 1977. **Proceedings...** [S.l.: s.n.], 1977. p.46–57.

POLL, E. Subtyping and Inheritance for Inductive Types. In: WORKSHOP ON SUBTYPING, INHERITANCE AND MODULAR DEVELOPMENT OF PROOFS, TYPES, 1997, Durham, UK. **Proceedings...** [S.l.: s.n.], 1997.

PRATT, T. W.; ZELKOWITZ, M. V. **Programming languages** : design and implementation. 3rd.ed. Upper Saddle River: Prentice-Hall, 1996. 654p.

QUEMENER, Y.-M.; JERON, T. **Model-Checking of CTL on Infinite Kripke Structures Defined by Simple Graph Grammars**. Rennes, France: Institut National de Recherche en Informatique et en Automatique (INRIA), 1995. 16p. (Research report RR-2563).

QUIELLE, J.-P.; SIFAKIS, J. Specification and Verification of Concurrent Systems in CESAR. In: COLLOQUIUM ON INTERNATIONAL SYMPOSIUM ON PROGRAMMING, 5., 1981. **Proceedings. . .** Berlin: Springer-Verlag, 1981. p.337–351. (Lecture Notes in Computer Science, v.137).

READE, C. **Elements of Functional Programming**. 2nd.ed. Wokingham: Addison-Wesley, 1995. 600p.

REISIG, W. **Petri Nets**: an introduction. Berlin: Springer-Verlag, 1985. 161p.

RENSINK, A. Towards Model Checking Graph Grammars. In: WORKSHOP ON AUTOMATED VERIFICATION OF CRITICAL SYSTEMS (AVOCS), 2003. **Proceedings. . .** Southampton: University of Southampton, 2003. p.150–160.

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

REPS, T. Algebraic Properties of Program Integration. **Science of Computer Programming**, [S.l.], v.17, n.1-3, p.139–215, 1991.

RIBEIRO, L. **Parallel Composition and Unfolding Semantics of Graph Grammars**. 1996. PhD Thesis — Technische Universität Berlin, Berlin. 202p.

RÖSSIGER, M. Coalgebras and Modal Logic. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.33, p.294–315, 2000.

ROTHE, J.; JACOBS, B.; TEWS, H. The Coalgebraic Class Specification Language CCSL. **Journal of Universal Computer Science**, [S.l.], v.7, n.2, p.175–193, 2001.

RUSSELL, S. J.; NORVING, P. **Artificial Intelligence** : a modern approach. Upper Saddle River: Prentice-Hall, 1995. 932p.

RUTTEN, J. J. M. M. Universal coalgebra: a theory of systems. **Theoretical Computer Science**, [S.l.], v.249, n.1, p.3–80, 2000.

RUTTEN, J. J. M. M.; TURI, D. Initial Algebra and Final Coalgebra Semantics for Concurrency. In: REX SCHOOL/SYMPOSIUM “A DECADE OF CONCURRENCY”, 1994. **Proceedings. . .** Berlin: Springer, 1994. p.530–582. (Lecture Notes in Computer Science, v.803).

SANTOS, O. M. dos. **Verificação Formal de Sistemas Distribuídos Modelados na Gramática de Grafos Baseada em Objetos**. 2004. 89p. Masters Thesis — Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre.

SCHMIDT, D. Structure-preserving binary relations for program abstraction. In: MOGENSEN, T.; SCHMIDT, D.; SUDBOROUGH, I. H. (Ed.). **The Essence of Computation**: complexity, analysis, transformation. [S.l.]: Springer, 2002. p.245–265. (Lecture Notes in Computer Science, v.2566).

SCHÜRR, A. PROGRESS: a VHL-language based on graph grammars. In: INTERNATIONAL WORKSHOP ON GRAPH GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 4., 1990, Bremen, Germany. **Proceedings. . .** Berlin: Springer-Verlag, 1990. p.641–659. (Lecture Notes in Computer Science, v.532).

SETHI, R. **Programming languages**: concepts and constructs. 2nd.ed. Reading: Addison-Wesley, 1996. 640p.

SEWELL, P. From rewrite rules to bisimulation congruences. **Theoretical Computer Science**, [S.l.], v.274, p.183–230, 2002.

SMITH, B. C. **Reflection and Semantics in a Procedural Language**. 1982. PhD Thesis — Massachusetts Institute of Technology, Cambridge, MA. (MIT-LCS-TR-272).

SMITH, G. **The Object Z Specification Language**. [S.l.]: Kluwer Academic Publishers, 1999. 146p.

SMITH, G. P. **An Object-Oriented Approach to Formal Specification**. 1992. 199p. PhD Thesis — Department of Computer Science, University of Queensland, Brisbane.

STERLING, L.; SHAPIRO, E. Y. **The Art of Prolog** : advanced programming techniques. 2nd.ed. Cambridge: MIT Press, 1994. 509p.

STIRLING, C. Modal and Temporal Logics. In: ABRAMSKY, S.; GABBAY, D. M.; MAIBAUM, T. S. E. (Ed.). **Handbook of Logic in Computer Science**. Oxford: Oxford Science Publications, 1992. v.2, p.477–563.

STOLTENBERG-HANSEN, V.; LINDSTRÖM, I.; GRIFFOR, E. R. **Mathematical Theory of Domains**. Cambridge: Cambridge University Press, 1994. 349p. (Cambridge Tracts in Theoretical Computer Science, v.22).

STROUSTUP, B. **The C++ Programming Language**. 3rd.ed. Upper Saddle River: Addison-Wesley, 2000. 911p.

SUMII, E.; PIERCE, B. C. A Bisimulation for Type Abstraction and Recursion. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 32., 2005, Long Beach, California. **Proceedings. . .** [S.l.: s.n.], 2005. p.63–74.

TAENTZER, G. **Parallel and Distributed Graph Transformation Formal Description and Application to Communication-Based Systems**. 1996. PhD Thesis — TU Berlin, Berlin.

TAENTZER, G. Modeling dynamic distributed object structures by graph transformation. **Object Currents**, [S.l.], v.1, n.12, December 1996.

TANENBAUM, A. S.; WOODHULL, A. S. **Operating systems**: design and implementation. 2nd.ed. Upper Saddle River: Prentice Hall, 1997. 939p.

THOMAS, P.; WEEDON, R. **Object-oriented programming in Eiffel**. Harlow: Addison-Wesley, 1997. 690p.

TURI, D. **Functorial Operational Semantics**. 1996. 227p. PhD Thesis — Vrije University, Amsterdam.

UNGAR, D.; CHAMBERS, C.; CHANG, B.-W.; HÖLZLE, U. Organizing Programs without Classes. **Lisp and Symbolic Computation**, [S.l.], v.3, n.4, 1991.

VALK, R. Concurrency in Communicating Petri Nets. In: AGHA, G. A.; CINDIO, F. D.; ROZENBERG, G. (Ed.). **Concurrent Object-Oriented Programming and Petri Nets**: advances in Petri nets. Berlin: Springer, 2001. p.164–195. (Lecture Notes in Computer Science, v.2001).

VARDI, M.; WOLPER, P. An automata-theoretic approach to automatic program verification. In: IEEE SYMPOSIUM ON LOGIC IN COMPUTER SCIENCE, 1., 1986. **Proceedings. . .** [S.l.: s.n.], 1986. p.322–331.

VISSER, W.; HAVELUND, K.; BRAT, G.; PARK, S.; LERDA, F. Model Checking Programs. **Automated Software Engineering Journal**, [S.l.], v.10, n.2, p.203–232, April 2003.

WAGNER, A.; GOGOLLA, M. Defining Operational Behavior of Object Specifications by Attributed Graph Transformations. **Fundamenta Informaticae**, [S.l.], v.26, n.3-4, p.407–431, 1996.

WARNER, J.; KLEPPE, A. OCL: the constraint language of UML. **Journal of Object-Oriented Programming**, [S.l.], v.12, n.1, p.10–28, 1999.

WATT, D. A. **Programming language concepts and paradigms**. New York: Prentice-Hall, 1990. 322p.

WATT, D. A. **Programming Language Syntax and Semantics**. Cambridge: Prentice Hall International, 1991. 389p.

WEHRHEIM, H. Inheritance of Temporal Logic Properties. In: IFIP TC6/WG6.1 INTERNATIONAL CONFERENCE ON FORMAL METHODS FOR OPEN OBJECT-BASED DISTRIBUTED SYSTEMS, FMOODS, 6., 2003, Paris. **Proceedings. . .** Berlin: Springer-Verlag, 2003. p.79–93. (Lecture Notes in Computer Science, v.2884).

WEISE, C. An incremental formal semantics for PROMELA. In: SPIN WORKSHOP, 3., 1997, The Nederlands. **Proceedings. . .** [S.l.: s.n.], 1997. 20p.

WINTER, A.; KULLBACH, B.; RIEDIGER, V. An Overview of the GXL Graph Exchange Language. In: INTERNATIONAL SEMINAR ON SOFTWARE VISUALIZATION, 2001, Dagstuhl Castle, Germany. **Proceedings. . .** Berlin: Springer Verlag, 2001. p.324–336. (Lecture Notes in Computer Science, v.2269).

WIRSING, M. Algebraic Specification. In: LEEUWEN, J. van (Ed.). **Handbook of Theoretical Computer Science**. Amsterdam: Elsevier, 1998. v.2, p.675–788.

APPENDIX A LEMMAS AND PROOFS

A.1 Lemmas

Lemma A.1 *Let $R \subseteq A \times A$ be a strict relation, and R^* its reflexive and transitive closure. Then the upper set $\uparrow x$ of any element $x \in A$ is a chain.*

Proof: Let x be an element of A and $\uparrow x$ its upper set (Definition B.11) with respect to R^*. Suppose that $\uparrow x$ is not a chain (Definition B.10). Then there must exist two distinct elements $a, b \in \uparrow x$ such that neither $(a, b) \in R^*$ nor $(b, a) \in R^*$. But, since they both belong to the upper set of x, we have that $(x, a) \in R^*$ and $(x, b) \in R^*$. Then, there must exist elements $x', a', b' \in \uparrow x$, with $a' \neq b'$, such that $(x, x') \in R^*$, $(x', a') \in R$, $(x', b') \in R$, $(a', a) \in R^*$ and $(b', b) \in R^*$. But this is impossible, since R is a function, and so a' must be equal to b'. Therefore, for any distinct elements $a, b \in \uparrow x$ either $(a, b) \in R^*$ or $(b, a) \in R^*$ (i.e., $\uparrow x$ is a chain). $\square$

Lemma A.2 *Let $R \subseteq A \times A$ be a strict relation, and R^* its reflexive and transitive closure. Then the upper set $\uparrow x$ of any element $x \in A$ always has a greatest element, denoted by $\sqcup\uparrow x$.*

Proof: Lemma A.1 assures that, for any element $x \in A$, the upper set $\uparrow x$ is a chain. According to Definition 2.1, for each $x \in A$, all sequences $(x, a_1), (a_1, a_2), \ldots, (a_{n-1}, a_n) \in R$, $n \geqslant 0$, are finite, which means that the upper set $\uparrow x$ is necessarily finite, and a finite chain has always a greatest element. $\square$

Lemma A.3 *Let $P_\sqsubseteq = \langle P, \sqsubseteq_P^* \rangle$ and $Q_\sqsubseteq = \langle Q, \sqsubseteq_Q^* \rangle$ be two strict ordered sets, and let $f : P_\sqsubseteq \to Q_\sqsubseteq$ be a strict ordered function. Then, for any $x \in dom(f)$, the set $f(\uparrow x)$ is a chain.*

Proof: The upper set of an element $x \in P$ is defined as $\uparrow x = \{a \in P \mid x \sqsubseteq_P a\}$. Since it is a chain (by Lemma A.1), then $\uparrow x = \{x = x_1, x_2, \ldots, x_n = \sqcup\uparrow x\}$, with $x = x_1 \sqsubseteq_P x_2 \sqsubseteq_P \ldots \sqsubseteq_P x_n = \sqcup\uparrow x\}$. Then, $f(\uparrow x) = \{f(x) = f(x_1), f(x_2), \ldots, f(x_n) = f(\sqcup\uparrow x)\}$, and since f is monotonic, then $f(x) = f(x_1) \sqsubseteq_Q f(x_2) \sqsubseteq_Q \ldots \sqsubseteq_Q f(x_n) = f(\sqcup\uparrow x)$. If $f(\uparrow x)$ is not a chain, then there must be $f(x_i)$ and $f(x_j)$, for some $1 \leqslant i < j \leqslant n$ such that $f(x_i) \neq f(x_j)$ and $f(x_j) \sqsubseteq_Q f(x_i)$. But this is impossible, since $\sqsubseteq_Q$ is a partial order, and therefore is antisymmetric. Hence, $f(\uparrow x)$ is a chain. $\square$

Lemma A.4 *Let $P_\sqsubseteq = \langle P, \sqsubseteq_P^* \rangle$ and $Q_\sqsubseteq = \langle Q, \sqsubseteq_Q^* \rangle$ be two strict ordered sets, and let $f : P_\sqsubseteq \to Q_\sqsubseteq$ be a strict ordered function. For any $x \in dom(f)$, there is no elements $a, b, y \in Q$ such that $a \sqsubseteq_Q y \sqsubseteq_Q b$, with $a, b \in f(\uparrow x)$ and $y \notin f(\uparrow x)$.*

Proof: It has already been established, by Lemma A.3, that the set $f(\uparrow x)$ is a chain (under $\sqsubseteq_Q$). Therefore, if there are elements $a, b, y \in Q$ such that $a \sqsubseteq_Q y \sqsubseteq_Q b$, we know that the following relations must also be true: $f(x) \sqsubseteq_Q a \sqsubseteq_Q y \sqsubseteq_Q b \sqsubseteq_Q \sqcup \uparrow f(x)$, because (by the conditions of the theorem) $a, b \in f(\uparrow x)$, and $\uparrow f(x)$ is also a chain with a greatest element (by Lemmas A.1 and A.2, respectively).

Since f is a strict ordered function, then (by definition) for any $x \in dom(f)$, we have that $f(\uparrow x) = \uparrow f(x) \cap \downarrow f(\sqcup \uparrow x)$. Since $b \in f(\uparrow x)$, b is also in the set $\downarrow f(\sqcup \uparrow x)$, in order to belong to the intersection. But then, if $y \sqsubseteq_Q b$, y must also belong to $\downarrow f(\sqcup \uparrow x)$. Additionally, $y \in \uparrow f(x)$, as the condition above explicits. Then, $y \in \uparrow f(x) \cap \downarrow f(\sqcup \uparrow x)$ and therefore $y \in f(\uparrow x)$. □

Lemma A.5 *Strict ordered sets are closed under finite or infinite disjoint union.*

Proof: Let I be a set and $X = \{\langle X_i, \sqsubseteq_{X_i} \rangle \mid i \in I\}$ a set of strict ordered sets indexed by I. The disjoint union of the strict ordered sets $\langle X_i, \sqsubseteq_{X_i} \rangle$ in X is defined as

$$\biguplus_I \langle X_i, \sqsubseteq_{X_i} \rangle = \langle \biguplus_I X_i, \biguplus_I \sqsubseteq_{X_i} \rangle$$

with

$$\biguplus_I X_i = \bigcup_I \{(x, i) \mid x \in X_i\}$$

and

$$\biguplus_I \sqsubseteq_{X_i} = \bigcup_I \{((x, i), (x', i)) \mid (x, x') \in \sqsubseteq_{X_i}\}$$

The disjoint union of any set is always a set, so we only have to prove that the disjoint union of the strict relations $\biguplus_I \sqsubseteq_{X_i}$ is a strict relation. Suppose, without loss of generality, that relations $\sqsubseteq_{X_i}$ are strict relations, and not their reflexive and transitive closure.

A reflexive pair $((x, i), (x, i))$ can only appear in the disjoint union $\biguplus_I \sqsubseteq_{X_i}$ if there is a pair $(x, x) \in \sqsubseteq_{X_i}$ for some $i \in I$; but this is not possible, since all $\sqsubseteq_{X_i}$, $i \in I$ are irreflexive. By the same reasoning, a cycle $((a, i), (a_1, i)), ((a_1, i), (a_2, i)), \ldots, ((a_{n-1}, i), (a_n, i)), ((a_n, i), (a', i)) \in \biguplus_I \sqsubseteq_{X_i}$ only exists if a cycle $(a, a_1), (a_1, a_2), \ldots, (a_{n-1}, a_n), (a_n, a')$ exists in some $\sqsubseteq_{X_i}$, $i \in I$, which cannot occur because none of such relations has cycles. Again, $\biguplus_I \sqsubseteq_{X_i}$ will not have a functional nature if there are pairs $((a, i), (a', i)), ((a, i), (a'', i)) \in \biguplus_I \sqsubseteq_{X_i}$ with $(a', i) = (a'', i)$, but this is not possible, since all relations X_i, $i \in I$ are functions. Finally, all sequences $((a_1, i), (a_2, i)), ((a_2, i), (a_3, i)), \ldots, ((a_{n-1}, i), (a_n, i)) \in \biguplus_I \sqsubseteq_{X_i}$ must be finite, since there are no infinte sequences $(a_1, a_2), (a_2, a_3), \ldots, (a_{n-1}, a_n)$ in any relation $\sqsubseteq_{X_i}$, $i \in I$.

Hence, the (finite or infinite) disjoint union of strict ordered sets is a strict ordered set. □

Lemma A.6 *Let I be a set and $X = \{\langle X_i, \sqsubseteq_{X_i} \rangle \mid i \in I\}$ a set of strict ordered sets indexed by I. The disjoint union of the strict ordered sets $\langle X_i, \sqsubseteq_{X_i} \rangle$ in X is defined as*

$$\biguplus_I \langle X_i, \sqsubseteq_{X_i} \rangle = \langle \biguplus_I X_i, \biguplus_I \sqsubseteq_{X_i} \rangle$$

with

$$\biguplus_I X_i = \bigcup_I \{(x,i) \mid x \in X_i\}$$

and

$$\biguplus_I \sqsubseteq_{X_i} = \bigcup_I \{((x,i),(x',i)) \mid (x,x') \in \sqsubseteq_{X_i}\}$$

let $\iota_i = \langle \iota_{X_i}, \iota_{\sqsubseteq_{X_i}} \rangle : \langle X_i, \sqsubseteq_{X_i} \rangle \to \coprod_I \langle X_i, \sqsubseteq_{X_i} \rangle$ be a family of injections defined as follows:

$$\begin{array}{ll} \iota_{X_i}(x) = (x,i) & x \in X_i \\ \iota_{\sqsubseteq_{X_i}}((x,x')) = ((x,i),(x',i)) & (x,x') \in \sqsubseteq_{X_i} \end{array}$$

then all functions ι_i are strict ordered functions.

Proof: Each $\iota_{X_i} : \langle X_i, \sqsubseteq_{X_i} \rangle \to \coprod_I \langle X_i, \sqsubseteq_{X_i} \rangle$ is, by definition, total. So, all $x \in X_i$, with $i \in I$, belong to the image of ι_{X_i}. Additionally, for each ι_{X_i} and $x \in X_i$, one has that $\iota_{X_i}(\uparrow x) = \{(a,i) \mid a \in \uparrow x\}$, $\uparrow \iota_{X_i}(x) = \uparrow(x,i) = \{(a,i) \mid x \sqsubseteq_{X_i} a\}$ and $\downarrow \iota_{X_i}(\sqcup \uparrow x) = \downarrow(\sqcup \uparrow x, i) = \{(a,i) \mid a \sqsubseteq_{X_i} \sqcup \uparrow x\}$. Since $\uparrow \iota_{X_i}(x) \subseteq \downarrow \iota_{X_i}(\sqcup \uparrow x)$, the the intersection of those two sets equals $\uparrow \iota_{X_i}(x)$, which in its turn equals $\iota_{X_i}(\uparrow x)$. Therefore, all ι_{X_i}, with $i \in I$, are strict ordered functions. □

Lemma A.7 *Let A be a set, $\langle P, \sqsubseteq_P \rangle$ be a partially ordered set. Then any total function $f : A \to P$ induces a partial order relation on A.*

Proof: Let $\sqsubseteq_f$ be the induced relation, defined as $\{(a,a') \mid f(a) \sqsubseteq_P f(a')\}$. Since $\sqsubseteq_P$ is reflexive, then for any $a \in A$ we have that $f(a) \sqsubseteq_P f(a)$, and so $\sqsubseteq_f$ is also reflexive; since $\sqsubseteq_P$ is transitive, then for any $a, a', a'' \in A$ we have that if $f(a) \sqsubseteq_P f(a')$ and $f(a') \sqsubseteq_P f(a'')$ then $f(a) \sqsubseteq_P f(a'')$, therefore if $a \sqsubseteq_f a'$ and $a' \sqsubseteq_f a''$ then $a \sqsubseteq_f a''$, so $\sqsubseteq_f$ is also transitive; finally, since $\sqsubseteq_P$ is antisymmetric, then if $f(a) \sqsubseteq_P f(a')$ and $f(a') \sqsubseteq_P f(a)$ then $a = a'$, therefore if $a \sqsubseteq_f a'$ and $a' \sqsubseteq_f a$ then $a = a'$, and so $\sqsubseteq_f$ is also antisymmetric. Those three properties together prove that$\sqsubseteq_f$ is a partial order relation. □

A.2 Proofs

Proof of Lemma 2.1 (PAG. 26):

Let R be a strict relation and R^* its reflexive and transitive closure. By definition, R^* is both reflexive and transitive. Suppose that R^* is not antisymmetric (there are two pairs $(a,b),(b,a) \in R^*$ such that $a \neq b$). Then there must exist a sequence of elements $c_1, \ldots, c_n$ with $n > 1$ such that $a = c_1 = c_n$ and $b = c_j$ for some $1 < j < n$, such that all pairs $(c_i, c_{i+1}) \in R$, for all $i = 1, \ldots, n-1$. But, since $c_1 = c_n$, then there must exist a cycle in R, which is not allowed by the definition of a strict relation. Therefore, if R is a strict relation then R^* is a partial order relation.

Proof of Theorem 2.2 (PAG. 28):

The initial object (Definition C.3) of **SOSet** is the empty strict ordered set $\langle \emptyset, \emptyset \rangle$. For any other strict ordered set $P_\sqsubseteq = \langle P, \sqsubseteq_P \rangle$, there is a unique (empty) strict ordered function $\emptyset : \langle \emptyset, \emptyset \rangle \to P_\sqsubseteq$. Notice that the empty function is trivially a strict ordered function.

Proof of Theorem 2.3 (PAG. 28):

Let I be a set and $X = \{\langle X_i, \sqsubseteq_{X_i}\rangle \mid i \in I\}$ a set of strict ordered sets indexed by I. The coproduct (Definition C.4) of the objects in X is the I-fold coproduct diagram, consisting of an object $\coprod_I \langle X_i, \sqsubseteq_{X_i}\rangle$, which is the disjoint union of the strict ordered sets $\langle X_i, \sqsubseteq_{X_i}\rangle$, defined as

$$\coprod_I \langle X_i, \sqsubseteq_{X_i}\rangle = \langle \biguplus_I X_i, \biguplus_I \sqsubseteq_{X_i}\rangle$$

with

$$\biguplus_I X_i = \bigcup_I \{(x,i) \mid x \in X_i\}$$

and

$$\biguplus_I \sqsubseteq_{X_i} = \bigcup_I \{((x,i),(x',i)) \mid (x,x') \in \sqsubseteq_{X_i}\}$$

and arrows (coproduct injections) $\iota_i = \langle \iota_{X_i}, \iota_{\sqsubseteq_{X_i}}\rangle : \langle X_i, \sqsubseteq_{X_i}\rangle \to \coprod_I \langle X_i, \sqsubseteq_{X_i}\rangle$ defined as follows:

$$\begin{array}{ll} \iota_{X_i}(x) = (x,i) & x \in X_i \\ \iota_{\sqsubseteq_{X_i}}((x,x')) = ((x,i),(x',i)) & (x,x') \in \sqsubseteq_{X_i} \end{array}$$

the object $\coprod_I \langle X_i, \sqsubseteq_{X_i}\rangle$ is indeed a strict ordered set, since the disjoint union of strict relations is a strict relation (Lemma A.5), an so are the injection functions strict ordered functions (Lemma A.6).

Now, for any I-indexed set of arrows $f_i : \langle X_i, \sqsubseteq_{X_i}\rangle \to \langle C, \sqsubseteq_C\rangle$ let $h : \coprod_I \langle X_i, \sqsubseteq_{X_i}\rangle \to \langle C, \sqsubseteq_C\rangle$ be the strict ordered function defined as

$$h((x,i)) = \begin{cases} f_i(x) & x \in \mathit{dom}(f_i) \\ \mathit{undef} & x \notin \mathit{dom}(f_i) \end{cases}$$

$f_i = h \circ \iota_i$ by construction: for any $x \in X_i$, we have that $(h \circ \iota_i)(x) = h(\iota_i(x)) = h((x,i)) = f_i(x)$, if $x \in \mathit{dom}(f_i)$ and $(h \circ \iota_i)(x) = h(\iota_i(x)) = h((x,i)) = \mathit{undef} = f_i(x)$ otherwise. Moreover, for any other strict ordered functions $h' : \coprod_I \langle X_i, \sqsubseteq_{X_i}\rangle \to \langle C, \sqsubseteq_C\rangle$ with $f_i = h' \circ \iota_i$, and for any $x \in X_i$, $(h' \circ \iota_i)(x) = h'(\iota_i(x)) = h'((x,i)) = f_i(x)$, if $x \in \mathit{dom}(f_i)$ and $(h' \circ \iota_i)(x) = h'(\iota_i(x)) = h'((x,i)) = \mathit{undef} = f_i(x)$ must be true, and so $h = h'$, which is the unique strict ordered function which makes the coproduct diagram to comute.

Proof of Theorem 2.6 (PAG. 34):

The initial object in **CGraph** is the (empty) class-model graph $\mathcal{C}_\emptyset = \langle V_{\emptyset\sqsubseteq}, E_{\emptyset\sqsubseteq}, L, \mathit{src}_\emptyset, \mathit{tar}_\emptyset, \mathit{lab}_\emptyset\rangle$, where $V_{\emptyset\sqsubseteq} = \langle V_\emptyset, \sqsubseteq_{V_\emptyset}\rangle$, $E_{\emptyset\sqsubseteq} = \langle E_\emptyset, \sqsubseteq_{E_\emptyset}\rangle$, where $V_\emptyset = E_\emptyset = \sqsubseteq_{V_\emptyset} = \sqsubseteq_{E_\emptyset} = \emptyset$, and $\mathit{src}_\emptyset, \mathit{tar}_\emptyset, \mathit{lab}_\emptyset : \emptyset \to \emptyset$ are empty functions. Given any class-model graph $\mathcal{C} = \langle V_\sqsubseteq, E_\sqsubseteq, L, \mathit{src}, \mathit{tar}, \mathit{lab}\rangle$, there is an unique morphism $!_\emptyset = \langle !_{\emptyset V} : \emptyset \to \emptyset, !_{\emptyset E} : \emptyset \to \emptyset, \mathit{id}_L : L \to L\rangle : \mathcal{C}_\emptyset \to \mathcal{C}$. Notice that $!_\emptyset$ is a class-model graph morphism, since the empty functions $!_{\emptyset V}$ and $!_{\emptyset E}$ are (trivially) strict ordered functions, and the diagram

$$\begin{tikzcd}
L \arrow[dd, |->, "id_L"'] & E_\emptyset \arrow[l, |->, "lab_\emptyset"'] \arrow[r, |->, "src_\emptyset{,}tar_\emptyset"] & V_\emptyset^* \arrow[dd, "!_{\emptyset V}^*"] \\
 & dom(!_{\emptyset E}) \arrow[u, hook, "!_{\emptyset E}?"'] \arrow[d, |->, "!_{\emptyset E}!"] & \\
L & E \arrow[l, |->, "lab"'] \arrow[r, |->, "src{,}tar"] & V^*
\end{tikzcd}$$

commutes trivially, since there is no element $e \in dom(!_{\emptyset E})$.

Proof of Theorem 2.7 (PAG. 34):

Let I be a set and $\mathfrak{C} = \{\mathcal{C}_i = \langle V_{i\sqsubseteq}, E_{i\sqsubseteq}, L, src_i, tar_i, lab_i \rangle \mid i \in I\}$ a set of class-model graphs indexed by I. The coproduct of the objects in $\mathfrak{C}$ in category **CGraph** is the I-fold coproduct diagram, consisting of an object $\coprod \mathfrak{C}$, and arrows (the coproduct injections) $\iota_i = \langle \iota_{V_i}, \iota_{E_i}, id_L \rangle : \mathcal{C}_i \to \coprod \mathfrak{C}$ defined as follows:

- $V_{\coprod \mathfrak{C}} = \bigcup_I \{\langle v, i \rangle \mid v \in V_i\}$;
- $\sqsubseteq_{V_{\coprod \mathfrak{C}}} = \bigcup_I \{(\langle v, i \rangle, \langle v', i \rangle) \mid (v, v') \in \sqsubseteq_{V_i}\}$;
- $E_{\coprod \mathfrak{C}} = \bigcup_I \{\langle e, i \rangle \mid e \in E_i\}$;
- $\sqsubseteq_{E_{\coprod \mathfrak{C}}} = \bigcup_I \{(\langle e, i \rangle, \langle e', i \rangle) \mid (e, e') \in \sqsubseteq_{E_i}\}$;
- $src_{\coprod \mathfrak{C}}(\langle e, i \rangle) = \langle v_1, i \rangle \langle v_2, i \rangle \ldots \langle v_n, i \rangle$, if and only if $src_i(e) = v_1 \ldots v_n$, $i \in I$;
- $tar_{\coprod \mathfrak{C}}(\langle e, i \rangle) = \langle v_1, i \rangle \langle v_2, i \rangle \ldots \langle v_k, i \rangle$, if and only if $tar_i(e) = v_1 \ldots v_k$, $i \in I$;
- $lab_{\coprod \mathfrak{C}}(\langle e, i \rangle) = lab_i(e)$, $i \in I$;
- $\iota_{V_i}(v) = (v, i)$ for all $v \in V_i$, $i \in I$;
- $\iota_{E_i}(e) = (e, i)$ for all $e \in E_i$, $i \in I$.

the object $\coprod \mathfrak{C}$ is indeed a class-model graph: the disjoint union of strict ordered sets is a strict ordered set (Lemma A.5), and the injection morphisms are class-model graph morphisms, since the component vertex and edge functions are strict ordered functions (Lemma A.6).

Now, let $\mathcal{C} = \langle V_\sqsubseteq, E_\sqsubseteq, L, src, tar, lab \rangle$ be a class-model graph, and let $f_i = \langle f_{iV}, f_{iE}, id_L \rangle : \mathcal{C}_i \to \mathcal{C}$ be a collection of class-model graph morphisms from each class-model graph $\mathcal{C}_i$ to $\mathcal{C}$. Let $h = \langle h_V, h_E, id_L \rangle : \coprod \mathfrak{C} \to \mathcal{C}$ be the class-model graph morphism defined as

$$h_V(\langle v, i \rangle) = \begin{cases} f_{iV}(v) & v \in dom(f_{iV}) \\ undef & v \notin dom(f_{iV}) \end{cases}$$

$$h_E(\langle e, i \rangle) = \begin{cases} f_{iE}(e) & e \in dom(f_{iE}) \\ undef & e \notin dom(f_{iE}) \end{cases}$$

it is easy to show that this is a class-model graph morphism, since it simply reflects the image of the morphisms f_i through the injections ι_i.

For each element $x \in \mathcal{C}_i$ (being x a vertex or an edge), $f_i = h \circ \iota_i$ by construction: $(h \circ \iota_i)(x) = h(\iota_i(x)) = h(\langle x, i \rangle) = f_i(x)$, if $x \in dom(f_i)$, and $(h \circ \iota_i)(x) = h(\iota_i(x)) = h(\langle x, i \rangle) = undef$ otherwise. Moreover, for any other class-model graph morphism $h' : \coprod \mathfrak{C} \to \mathcal{C}$ with $f_i = h' \circ \iota_i$, and for any $x \in \mathcal{C}_i$, $(h' \circ \iota_i)(x) = h'(\iota_i(x)) = h'(\langle x, i \rangle) = f_i(x)$, if $x \in dom(f_i)$ and $(h' \circ \iota_i)(x) = h'(\iota_i(x)) = h'(\langle x, i \rangle) = undef$ must be true, and so $h = h'$, which is the unique class-model graph morphism which makes the coproduct diagram to commute.

A.3 Definitions

Definition A.1 (Forgetful functors) *The forgetful functors $U_{\sqsubseteq}$, and U_G are defined as follows:*

- *let* $U_{\sqsubseteq}$: **CGraph** $\rightarrow$ **LabHGraphP** *be the functor which sends each object* $\langle V_{\sqsubseteq}, E_{\sqsubseteq}, L, src, tar, lab \rangle$*, with* $V_{\sqsubseteq} = \langle V, \sqsubseteq_V \rangle$ *and* $E_{\sqsubseteq} = \langle E, \sqsubseteq_E \rangle$*, in* **CGraph** *to the object* $\langle V, E, L, src, tar, lab \rangle$ *in* **LabHGraphP***, and each class-model graph morphism* $f : \mathcal{C}_1 \rightarrow \mathcal{C}_2$ *to itself;*

- *let* U_G : **CGraph** $\rightarrow$ **SOSet** $\times$ **SOSet** *be the functor which sends each object* $\langle V_{\sqsubseteq}, E_{\sqsubseteq}, L, src, tar, lab \rangle$*, with* $V_{\sqsubseteq} = \langle V, \sqsubseteq_V \rangle$ *and* $E_{\sqsubseteq} = \langle E, \sqsubseteq_E \rangle$*, in* **CGraph** *to the object* $(V_{\sqsubseteq}, E_{\sqsubseteq})$ *in* **SOSet** $\times$ **SOSet***, and each class-model graph morphism* $f = \langle f_V, f_E, id_L \rangle : \mathcal{C}_1 \rightarrow \mathcal{C}_2$ *to the pair of strict ordered functions* (f_V, f_E)*;*

- *let* $\mathcal{U}_t$: **OOGraphP**$(\mathcal{C})$ $\rightarrow$ **HGraphP** *be the functor which sends each object* $\langle G, t_G, \mathcal{C} \rangle$ *in* **OOGraphP**$(\mathcal{C})$ *to the object* G *in* **HGraphP***, and each arrow in* **OOGraphP**$(\mathcal{C})$ *to itself.*

APPENDIX B ORDER RELATIONS

B.1 Definitions

Definition B.1 (Relation) *Let $A_1, A_2, \ldots, A_n$ be sets. A n-ary relation R over sets $A_1, A_2, \ldots, A_n$ is any subset of the cartesian product $A_1 \times A_2 \times \ldots \times A_n$.*

Definition B.2 (Binary relation) *A binary relation R is a 2-ary relation (Definition B.1).*

Definition B.3 (Reflexive relation) *A binary relation $R \subseteq A \times A$ is* reflexive *if and only if $a \in A$ implies $(a, a) \in R$.*

Definition B.4 (Symmetric relation) *A binary relation $R \subseteq A \times A$ is* symmetric *if and only if $(a, b) \in R$ implies $(b, a) \in R$.*

Definition B.5 (Antisymmetric relation) *A binary relation $R \subseteq A \times A$ is* antisymmetric *if and only if $(a, b) \in R$ and $(b, a) \in R$ implies $a = b$.*

Definition B.6 (Transitive relation) *A binary relation $R \subseteq A \times A$ is* transitive *if and only if $(a, b) \in R$ and $(b, c) \in R$ implies $(a, c) \in R$.*

Definition B.7 (Preorder) *A binary relation $R \subseteq A \times A$ is a* quasi-order *or a* preorder *if and only if it is reflexive and transitive.*

Definition B.8 (Partial order relation) *A binary relation $R \subseteq A \times A$ is a* partial order relation *if and only if it is reflexive, antisymmetric and transitive.*

Definition B.9 (Partially ordered set) *A* partially ordered set *(or simply* poset*) $P_{\sqsubseteq} = \langle P, \sqsubseteq_P \rangle$ is a set P together with a partial order relation $\sqsubseteq_P$ on P.*

Definition B.10 (Chain) *A partially ordered set $P_{\sqsubseteq}$ is a* chain *if and only if for all $x, y \in P$ either $x \sqsubseteq_P y$ or $y \sqsubseteq_P x$. A chain is also known as a* totally ordered set *or* linearly ordered set*.*

Definition B.11 (Upper set, lower set) *Let $\langle P, \sqsubseteq_P \rangle$ be a partially ordered set. A subset $A \subseteq P$ is an* upper set *if whenever $x \in A$, $y \in P$ and $x \sqsubseteq_P y$ we have $y \in A$. The upper set of $\{x\}$, with $x \in P$ (also called the set of all elements* above *x) is denoted by $\uparrow x$.*

Dually, a subset $A \subseteq P$ is an lower set *if $x \in A$ implies $y \in A$ for all $y \sqsubseteq x$. The set of all elements below some element $x \in A$ is denoted by $\downarrow x$.*

Definition B.12 (Upper bound, lower bound) *Let $\langle P, \sqsubseteq_P \rangle$ be a partially ordered set. An element $x \in P$ is called an* upper bound *for a subset $A \subseteq P$, written $A \sqsubseteq x$, if and only if $a \sqsubseteq x$ for all $a \in A$. The set of all upper bounds of A is denoted by* $\mathrm{ub}(A)$

Dually, an element $x \in P$ is called a lower bound *for a subset $A \subseteq P$, written $x \sqsubseteq A$, if and only if $x \sqsubseteq a$ for all $a \in A$. The set of all lower bounds of A is denoted by* $\mathrm{lb}(A)$.

Definition B.13 (Largest element, least element) *Let $\langle P, \sqsubseteq_P \rangle$ be a partially ordered set. An element $x \in P$ is the* largest element *of P if it is above all the other elements of P. The largest element of a poset is often called the* top *element, and it is denoted by $\top$.*

Dually, an element $x \in P$ is the least element *of P if it is below all the other elements of P. The least element of a poset is often called the* bottom *element, and it is denoted by $\perp$.*

Definition B.14 (Supremum (lub, join), infimum (glb, meet)) *Let $\langle P, \sqsubseteq_P \rangle$ be a partially ordered set. If for a subset $A \subseteq P$ the set of upper bounds has a least element x, then x is called the* least upper bound, *or* supremum, *or* join *of A, written $x = lub(A)$ or $x = \sqcup A$.*

Dually, if for a subset $A \subseteq P$ the set of lower bounds has a largest element x, then x is called the greatest lower bound, *or*infimum, *or* meet *of A, written $x = glb(A)$ or $x = \sqcap A$.*

Definition B.15 (Monotonic function) *Let $\langle P, \sqsubseteq_P \rangle$ and $\langle Q, \sqsubseteq_Q \rangle$ be two partially ordered sets. A function $f : P \to Q$ is called* monotonic order-preserving, *or simply* monotonic, *if and only if for all $p, p' \in P$, if $p \sqsubseteq_P p'$ then $f(p) \sqsubseteq_Q f(p')$.*

Definition B.16 (Partial monotonic function) *Let $\langle P, \sqsubseteq_P \rangle$ and $\langle Q, \sqsubseteq_Q \rangle$ be two partially ordered sets, and let $S \subseteq P$ be any subset of P. A partial monotonic function between P and Q is any monotonic function $f : S \to Q$.*

B.2 Operations

Definition B.17 (Disjoint union) *Let $\langle P, \sqsubseteq_P \rangle$ and $\langle Q, \sqsubseteq_Q \rangle$ be two partially ordered sets. The* disjoint union, *or* sum, *of $P_\sqsubseteq$ and $Q_\sqsubseteq$, denoted by $P_\sqsubseteq \uplus Q_\sqsubseteq$, is the partially ordered set $\langle (P \uplus Q), \sqsubseteq_{(P \uplus Q)} \rangle$, where $P \uplus Q$ is the usual disjoint union of sets, and $\sqsubseteq_{P_\sqsubseteq \uplus Q_\sqsubseteq}$ is defined as follows: $x \sqsubseteq_{P_\sqsubseteq \uplus Q_\sqsubseteq} y$ if and only if $x, y \in P$ and $x \sqsubseteq_P y$ or $x, y \in Q$ and $x \sqsubseteq_Q y$.*

Definition B.18 (Product) *Let $P_1, P_2, \ldots, P_n$ be partially ordered sets. The cartesian product $P_1 \times P_2 \times \ldots \times P_n$ can be made into a partially ordered set by imposing the coordinate-wise order defined as follows: $(x_1, \ldots, x_n) \sqsubseteq_{(P_1 \times \ldots \times P_n)} (y_1, \ldots, y_n)$ if and only if $x_i \sqsubseteq_{P_i} y_i$ for all $i = 1, \ldots, n$.*

B.3 Theorems

Lemma B.1 *(From (DAVEY; PRIESTLEY, 2002)) The disjoint union operation is associative (up to isomorphism).*

Lemma B.2 *(From (DAVEY; PRIESTLEY, 2002)) Any subset of a totally ordered set (Definition B.10) is a totally ordered set.*

Lemma B.3 *Monotonic functions are closed under composition.*

Proof: Let $f : \langle P, \sqsubseteq_P \rangle \to \langle Q, \sqsubseteq_Q \rangle$ and $g : \langle Q, \sqsubseteq_Q \rangle \to \langle R, \sqsubseteq_R \rangle$ be two monotonic functions. Then, for any $x, y \in P$, if $x \sqsubseteq_P y$ then (by definition) $f(x) \sqsubseteq_Q f(y)$, and (also by definition) $g(f(x)) \sqsubseteq_R g(f(y))$. So, for any $x, y \in P$, such that $x \sqsubseteq_P y$ we have that $(g \circ f)(x) \sqsubseteq_R (g \circ f)(y)$. $\square$

Lemma B.4 *Partial monotonic functions are closed under composition.*

Proof: Let $f : \langle P, \sqsubseteq_P \rangle \to \langle Q, \sqsubseteq_Q \rangle$ and $g : \langle Q, \sqsubseteq_Q \rangle \to \langle R, \sqsubseteq_R \rangle$ be two partial monotonic functions. Let $dom(f) \in P$ and $dom(g) \in Q$ denote the subsets of P and Q to which, respectively, f and g are defined. If $f(x) \in dom(g)$ for all $x \in dom(f)$, then the proof reduces to the one in Lemma B.3. If not, then the elements which are outside the domain of g do not play any role in the determination wether the composition $g \circ f$ is monotonic. $\square$

Lemma B.5 *Composition of monotonic functions and partial monotonic functions is associative.*

Proof: Direct from the fact that composition of total functions and partial functions is associative. $\square$

APPENDIX C CATEGORY THEORY

C.1 Definitions

Definition C.1 (Category) *A category* $\mathbf{C}$ *is a tuple* $\langle Obj_C, Mor_C, dom, cod, \circ, id\rangle$ *where* $Obj_{\mathbf{C}}$ *is a collection of* objects*,* $Mor_{\mathbf{C}}$ *is a collection of* morphisms *(or* arrows*),* $dom, cod : Mor_{\mathbf{C}} \rightarrow Obj_{\mathbf{C}}$ *are two operations, assigning to each arrow* f *two objects, respectively called* domain *(*source*) and* codomain *(*target*) of* f*,* $\circ : Mor_{\mathbf{C}} \times Mor_{\mathbf{C}} \rightarrow Mor_{\mathbf{C}}$ *is the morphism composition operation, and* $id : Obj_{\mathbf{C}} \rightarrow Mor_{\mathbf{C}}$ *is an operation which assigns to each object* b *a morphism* id_b*, called the* identity of b*, such that* $dom(id_b) = cod(id_b) = b$*.*
Identity and composition must satisfy the following conditions:

- *identity law: for any arrows* f*,*g *such that* $cod(f) = dom(g) = b$*,* $id_b \circ f = f$ *and* $g \circ id_b = b$*.*

- *associativity law: for any arrows* f*,* g*,* h *such that* $dom(f) = cod(g)$ *and* $dom(g) = cod(h)$*,* $(f \circ g) \circ h = f \circ (g \circ h)$*.*

Elements of $Obj_{\mathbf{C}}$ *and* $Mor_{\mathbf{C}}$ *are called, respectively,* $\mathbf{C}$*-objects and* $\mathbf{C}$*-arrows.*

Definition C.2 (Functor) *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be two categories. A* functor T *between categories* $\mathbf{A}$ *and* $\mathbf{B}$ *(i.e., with domain* $\mathbf{A}$ *and codomain* $\mathbf{B}$*) is a pair of related functions* $\langle T_O : Obj_{\mathbf{A}} \rightarrow Obj_{\mathbf{B}}, T_f : Mor_{\mathbf{A}} \rightarrow Mor_{\mathbf{B}}\rangle$*. The first one assigns for each object* $a \in Obj_{\mathbf{A}}$ *an object* $T_O(a) \in Obj_{\mathbf{B}}$*, and the second one assigns for each arrow* $f : a \rightarrow a'$ *in* $Mor_{\mathbf{A}}$ *an arrow* $T_f(f) : T_O(a) \rightarrow T_O(a)$ *in* $Mor_{\mathbf{B}}$*. The assignment is such that for all objects* $a \in Obj_{\mathbf{A}}$ *we have that* $T_f(id_a) = id_{T_O(a)}$ *and for all arrows* $f, g \in Mor_{\mathbf{A}}$ *with* $dom(g) = cod(f)$ *we have that* $T_f(g \circ f) = T_f(g) \circ T_f(f)$*.*

Definition C.3 (Initial object) *Let* $\mathbf{C}$ *be a category and* I *an object of* $\mathbf{C}$*.* I *is said an* initial object *in* $\mathbf{C}$ *if and only if for any* $\mathbf{C}$*-object* O *there exists an unique* $\mathbf{C}$*-morphism from* I *to* O*, denoted by* $!_I : I \rightarrow O$*.*

Definition C.4 (Coproduct) *Let* $\mathbf{C}$ *be a category. A* coproduct *of two* $\mathbf{C}$*-objects* A *and* B *is an* $\mathbf{C}$*-object* $A + B$*, together with two* $\mathbf{C}$*-arrows* $\iota_A : A \rightarrow A + B$ *and* $\iota_B : A \rightarrow A + B$*, such that for any* $\mathbf{C}$*-object* C *and pair of* $\mathbf{C}$*-arrows* $f : A \rightarrow C$ *and* $g : B \rightarrow C$ *there is exactly one* $\mathbf{C}$*-arrow* $h : A + B \rightarrow C$ *making the following diagram commute:*

$$\begin{tikzcd}
A \arrow[r, "\iota_A"] \arrow[dr, "f"'] & A + B \arrow[d, dashed, "h"] & B \arrow[l, "\iota_B"'] \arrow[dl, "g"] \\
 & C &
\end{tikzcd}$$

Definition C.5 (Cocone) *Let* $\mathbf{C}$ *be a category and* D *a diagram in* $\mathbf{C}$*. A* cocone *for* D *is a* $\mathbf{C}$*-object* X *and arrows* $f_i : D_i \to X$ *(one for each object* D_i *in* D*), such that, for each arrow* $g : D_i \to D_j$ *in* D*,* $f_i = f_j \circ g$*. The cocone of a diagram* D *is denoted by* $\{f_i : D_i \to X\}$*.*

Definition C.6 (Colimit) *A colimit for a diagram* D *is a cocone* $\{f_i : D_i \to X\}$ *with the property that if* $\{{f'}_i : D_i \to X'\}$ *is another cocone for* D *then there is a unique arrow* $k : X' \to X$ *such that* $f_i = k \circ k'_i$*.*

Definition C.7 (Coequalizer) *Given a category* $\mathcal{C}$ *and two arrows* $a : A \to B$ *and* $b : A \to B$ *of* $\mathcal{C}$*, a tuple* $\langle C, c : B \to C\rangle$ *is called a* co-equalizer *of* $\langle a, b\rangle$ *if* $c \circ a = c \circ b$ *and for all objects* D *and arrows* $d : B \to D$*, with* $d \circ a = d \circ b$*, there is a unique arrow* $u : C \to D$ *such that* $u \circ c = d$*.*

Definition C.8 (Pushout) *Given two morphisms* $f : A \to B$ *and* $g : A \to C$ *in some category* $\mathcal{C}$*, a pushout of* f *and* g *in* $\mathcal{C}$ *is an object* D *with two morphisms* $f_g : C \to D$ *and* $g_f : B \to D$ *in* $\mathcal{C}$ *such that (1)* $f_g \circ g = g_f \circ f$*, and (2) for each pair of* $\mathcal{C}$*-morphisms* $f' : C \to E$ *and* $g' : B \to E$ *with* $f' \circ g = g' \circ f$*, there is a* unique *morphism* $u : D \to E$*, called* the universal morphism *of the pushout, such that* $u \circ f_g = f'$ *and* $u \circ g_f = g'$*.*

C.2 Theorems

Lemma C.1 *Let* $\mathbf{S}$ *be a subcategory of* $\mathbf{C}$*. Then there exists a faithful functor* $\mathcal{I} : \mathbf{S} \to \mathbf{C}$ *which sends each object and arrow in* $\mathbf{S}$ *to itself in* $\mathbf{C}$*.*

Proof: See (MACLANE, 1998), page 15. □

C.3 Constructions

Definition C.9 (Category Set) **Set** *is the category with sets as objects and total functions as arrows.*

Definition C.10 (Coproduct in Set) *Let* X *and* Y *be sets. The coproduct* $\langle X \uplus Y, i_X, i_Y\rangle$ *of* X *and* Y *in* **Set** *is the disjoint union of them, defined as* $X \uplus Y = (X \times \{\cdot\}) \cup (\{\cdot\} \times Y)$ *together with the injections* $i_X : X \to X \uplus Y$ *and* $i_Y : Y \to X \uplus Y$ *where* $i_X(x) = (x, \cdot)$ *and* $i_Y(y) = (\cdot, y)$ *for all* $x \in X$ *and* $y \in Y$*.*

Definition C.11 (Generalized coproduct in Set) *Let* I *be a set, and* X *a family of sets indexed by* I*. The coproduct (Definition C.4) of the objects in* X *is the* I*-fold coproduct diagram, consisting of an object* $\coprod_I X_i$*, which is the disjoint union of the sets* X_i*, defined as*

$$\coprod_I X_i = \bigcup_I \{(x, i) \mid x \in X_i\}$$

and arrows (coproduct injections) $\iota_{X_i} : X_i \to \coprod_I X_i$ *defined as* $\iota_{X_i}(x) = (x, i)$ *for all* $x \in X_i$*.*

Definition C.12 (Coequalizer in Set (MACLANE, 1998)) *Let X and Y be sets and $f, g : X \to Y$ be functions. The coequalizer of arrows f and g in* **Set** *is the pair $\langle C, c : Y \to C\rangle$ where C is the quotient set of Y by least equivalence relation contained in $Y \times Y$ which contains all pairs $(f(x), g(x))$ for all $x \in X$, and c is the projection of Y onto C.*

Definition C.13 (Category SetP) **SetP** *is the category with sets as objects and partial functions as arrows.*

Definition C.14 (Coproduct in SetP) *Coproducts in* **SetP** *are identical to the ones in* **Set**.

Definition C.15 (Coequalizer in SetP) *Let $f, g : A \to B$ be two partial functions. Let $B' \subseteq B$ be the maximal subset of B defined as follows: $e \in B' \Leftrightarrow (e \notin (im(f) \cup im(g)) \vee$ $(e = f(x) \to (x \in dom(g) \wedge g(x) \in B') \wedge$ $e = g(x) \to (x \in dom(f) \wedge f(x) \in B')))$, and let $C \subseteq \mathscr{P}(B')$ be the quotient set of B' by the least equivalence class which contains all pairs $(f(x), g(x)) \in B' \times B'$; and let $c : B \to C\rangle$ be the projection of B' onto C (c is undefined for all elements in $B \setminus B'$). Then the pair $\langle C, c\rangle$ is the coequalizer of arrows f and g in* **SetP**.

Definition C.16 (Pushout in SetP) *Let $f : X \to Y$ and $g : X \to Y'$ be two partial functions, and let $\langle Y + Y', \iota, \iota'\rangle$ be the coproduct of sets Y and Y'. Let $\langle C, c : Y + Y' \to C\rangle$ be the coequalizer of arrows $\iota \circ f : X \to Y + Y'$ and $\iota' \circ g : X \to Y + Y'$. The pushout of arrows f and g in* **SetP** *is the tuple $\langle C, \iota_c : Y \to C, {\iota'}_c : Y' \to C\rangle$ such that the diagram*

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "g"] & Y \arrow[d, hook, "\iota"'] \arrow[ddr, bend left, "c\circ\iota"] & \\
Y' \arrow[r, hook, "\iota'"] \arrow[drr, bend right, "c\circ\iota'"] & Y + Y' \arrow[dr, "c"] & \\
& & C
\end{tikzcd}$$

commutes.

APPENDIX D GRAPHS

D.1 Definitions

Definition D.1 (Graph) *A graph is a tuple $\langle V, E\rangle$ where V is a set of vertices and $E \subseteq V \times V$ is a binary relation over V, called the set of edges.*

Definition D.2 (Graph) *A graph G is a tuple $\langle V_G,\ E_G,\ src_G,\ tar_G\rangle$ where V_G is a set of vertices, E_G is a set of edges, and $src_G, tar_G : E_G \rightarrow V_G$ return for each edge its, respectively, source and target nodes.*

Definition D.3 (Total graph morphism) *Let $G_1 = \langle V_{G_1},\ E_{G_1},\ src_{G_1},\ tar_{G_1}\rangle$ and $G_2 = \langle V_{G_2},\ E_{G_2},\ src_{G_2},\ tar_{G_2}\rangle$ be two graphs. Let $h_V : V_{G_1} \rightarrow V_{G_2}$ and $h_E : E_{G_1} \rightarrow E_{G_2}$ be two total functions. The pair $h = \langle h_V, h_E\rangle : G_1 \rightarrow G_2$ is a total graph morphism between G_1 and G_2 if and only if $h_V \circ src_{G_1} = src_{G_2} \circ h_E$ and $h_V \circ tar_{G_1} = tar_{G_2} \circ h_E$. A total graph morphism is said to be injective (surjective, bijective) if its component functions are injective (surjective, bijective).*

Definition D.4 (Partial graph morphism) *Let $G = \langle V_G,\ E_G,\ src_G,\ tar_G\rangle$ be a graph. A graph $S = \langle V_S,\ E_S,\ src_S,\ tar_S\rangle$ is said to be a* subgraph *of G, written $S \subseteq G$ or $S \hookrightarrow G$, if and only if $V_S \subseteq V_G$, $E_S \subseteq E_G$, $src_S = src_G|_{E_S}$, and $src_S = tar_G|_{E_S}$. A* partial graph morphism *h between two hypergraphs G_1 and G_2 is a total hypergraph morphism from some subgraph $dom(h) \hookrightarrow G_1$ to G_2. $dom(h)$ is called the* domain *of h.*

Definition D.5 (Alphabet, string) *An* alphabet *Σ is any finite set of symbols. A* string *w over an alphabet Σ is a sequence $w_1 \ldots w_n$, $n \geqslant 0$, where each $w_i \in \Sigma$, $i = 0, \ldots, n$. A sequence of zero elements over an alphabet Σ is called the* empty string*, and it is denoted by ε. The set of all finite strings over Σ is denoted by Σ^*.*

Definition D.6 (Hypergraph) *A hypergraph H is a tuple $\langle V_H,\ E_H,\ src_H,\ tar_H\rangle$ where V_H is a set of vertices, E_H is a set of hyperedges, and $src_H, tar_H : E_H \rightarrow V_H^*$ are, respectively, the hyperarcs source and target functions. The elements of V_H^* are strings over V_H (Definition D.5).*

Definition D.7 (Total hypergraph morphism) *Let $H_1 = \langle V_{H_1},\ E_{H_1},\ src_{H_1},\ tar_{H_1}\rangle$ and $H_2 = \langle V_{H_2},\ E_{H_2},\ src_{H_2},\ tar_{H_2}\rangle$ be two hypergraphs. Let $h_V : V_{H_1} \rightarrow V_{H_2}$ and $h_E : E_{H_1} \rightarrow E_{H_2}$ be two total functions, where $h_V^* : V_{H_1}^* \rightarrow V_{H_2}^*$ is the extension of function h_V to strings, such that, for all strings $w = w_1 \ldots w_n \in V_{H_1}^*$, $h_V^*(w) = h_V(w_1)h_V(w_2)\ldots h_V(w_n)$, $n \geqslant 0$. The pair $h = \langle h_V, h_E\rangle : H_1 \rightarrow H_2$ is a* *total hypergraph morphism between H_1 and H_2 if and only if $h^*_V \circ src_{H_1} = src_{H_2} \circ h_E$ and $h^*_V \circ tar_{H_1} = tar_{H_2} \circ h_E$. A total hypergraph morphism is said to be injective (surjective, bijective) if its component functions are injective (surjective, bijective).*

Definition D.8 (Partial hypergraph morphism) *Let $H = \langle V_H, E_H, src_H, tar_H\rangle$ be a hypergraph. A hypergraph $S = \langle V_S, E_S, src_S, tar_S\rangle$ is said to be a* subgraph *of H, written $S \subseteq H$ or $S \hookrightarrow H$, if and only if $V_S \subseteq V_H$, $E_S \subseteq E_H$, $src_S = src_H|_{E_S}$, and $src_S = tar_H|_{E_S}$. A* partial hypergraph morphism *h between two hypergraphs H_1 and H_2 is a total hypergraph morphism from some subgraph $dom(h) \hookrightarrow H_1$ to H_2. $dom(h)$ is called the* domain *of h.*

Definition D.9 (Labeled hypergraph) *A labeled hypergraph H is a tuple $\langle V_H, E_H, L_H, src_H, tar_H, lab_H\rangle$ where $\langle V_H, E_H, src_H, tar_H\rangle$ is a hypergraph, L_H is a finite set of hyperedge labels, and $lab : E_H \rightarrow L_H$ is the hyperedges labeling function.*

Definition D.10 (Total labeled hypergraph morphism) *Let $H_1 = \langle V_{H_1}, E_{H_1}, L_{H_1}, src_{H_1}, tar_{H_1}, lab_{H_1}\rangle$ and $H_2 = \langle V_{H_1}, E_{H_1}, L_{H_1}, src_{H_1}, tar_{H_1}, lab_{H_1}\rangle$ be two labeled hypergraphs. A total labeled hypergraph morphism $h : H_1 \rightarrow H_2$ is a tuple $h = \langle h_V, h_E, h_L\rangle$ where $\langle h_V, h_E\rangle$ is a total hypergraph morphism, h_L is a total function, and $h_L \circ lab_1 = lab_2 \circ h_E$.*

Definition D.11 (Partial labeled hypergraph morphism) *Let $H = \langle V_H, E_H, L_H, src_H, tar_H, lab_H\rangle$ be a labeled hypergraph. A labeled hypergraph $S = \langle V_S, E_S, L_S, src_S, tar_S, lab_S\rangle$ is a* subgraph *of H, written $S \subseteq H$ or $S \hookrightarrow H$, if and only if $\langle V_S, E_S, L_S, src_S, tar_S, lab_S\rangle \hookrightarrow \langle V_H, E_H, L_H, src_H, tar_H, lab_H\rangle$, $L_S \subseteq L_H$, and $lab_S = lab_H|_{L_S}$. A* partial labeled hypergraph morphism *h between two labeled hypergraphs H_1 and H_2 is a total labeled hypergraph morphism from some subgraph $dom(h) \hookrightarrow H_1$ to H_2. $dom(h)$ is called the* domain *of h.*

Definition D.12 (Typed hypergraph) *A typed hypergraph H_T is a tuple $\langle H, t, T\rangle$ where H and T typed (labeled) hypergraphs, and $t : H \rightarrow T$ is a total (labeled) hypergraph morphism.*

Definition D.13 (Typed hypergraph morphism) *Let $H_1^{T_1} = \langle H_1, t_1, T_1\rangle$ and $H_2^{T_2} = \langle H_2, t_2, T_2\rangle$ be two typed hypergraphs. A typed hypergraph morphism $h : H_1 \rightarrow H_2$ is a tuple $h = \langle h_H, h_T\rangle$, where $h_H : H_1 \rightarrow H_2$ and $h_T : T_1 \rightarrow T_2$ are (possibly partial) hypergraph morphisms such that $h_T \circ t_1 = t_2 \circ h_H$.*

D.2 Categories

Definition D.14 (Category Graph) *The category* **Graph** *has graphs as objects and total graph morphisms as arrows.*

Definition D.15 (Category GraphP) *The category* **GraphP** *has graphs as objects and partial graph morphisms as arrows.*

Definition D.16 (Category HGraph) *The category* **HGraph** *has hypergraphs as objects and total hypergraph morphisms as arrows.*

Definition D.17 (Category HGraphP) *The category* **HGraphP** *has hypergraphs as objects and partial hypergraph morphisms as arrows.*

Definition D.18 (Category LabHGraph) *The category* **LabHGraph** *has labeled hypergraphs as objects and total labeled hypergraph morphisms as arrows.*

Definition D.19 (Category LabHGraphP) *The category* **LabHGraphP** *has labeled hypergraphs as objects and partial labeled hypergraph morphisms as arrows.*

Definition D.20 (Category HGraphP(T)) *Given a (labeled) hypergraph T, the category* **HGraphP(T)** *has (labeled) hypergraphs typed over T as objects and typed hypergraph morphisms as arrows.*

APPENDIX E SPIN

E.1 Promela

The following list defines the grammar of the input language for the SPIN model checker version 3.0. The notational conventions are as follows:

- Choices are separated by vertical bars: |.
- Optional parts are included in square brackets: [...].
- A *Kleene star* * (LEWIS; PAPADIMITRIOU, 1998), (HOPCROFT; MOTWANI; ULLMAN, 2001) indicates zero or more repetitions of the immediately preceding fragment.
- Literals are enclosed in single quotes: `'` ... `'`.
- Uppercase names refer to tokens (i.e., terminals) representing keywords. In Promela models, the keywords are spelled like these token names, but in lowercase instead of uppercase.
- Lowercase names refer to grammar rules from this list.

The name `any_ascii_char` refers to any printable ASCII character except the double quote character: ". The statement separator used in this list is the semi-colon `';'`. within conditional choices the semi-colon can be replaced with a two-character arrow symbol: `'->'`, without change of meaning.

The complete Promela context-free grammar is given below:

```
spec    : module [ module ] *

module  : proctype    /* proctype declaration */
         | init       /* init process         */
         | never      /* never claim          */
         | trace      /* event trace, 3.0 only*/
         | utype      /* user defined types   */
         | mtype      /* mtype declaration    */
         | decl_lst   /* global vars, chans   */

proctype : [ active ] PROCTYPE name '(' [ decl_lst ]')'
           [ priority ] [ enabler ] '{' sequence '}'

init     : INIT [ priority ] '{' sequence '}'
```

```
never    : NEVER '{' sequence '}'

trace    : TRACE '{' sequence '}'

utype    : TYPEDEF name '{' decl_lst '}'

mtype    : MTYPE [ '=' ] '{' name [ ',' name ] * '}'

decl_lst : one_decl [ ';' one_decl ] *

one_decl : [ visible ] typename  ivar [',' ivar ] *

typename : BIT | BOOL | BYTE | SHORT | INT | MTYPE | CHAN
           | uname /* user defined type names (see utype) */

active   : ACTIVE [ '[' const ']' ]  /* instantiation */

priority : PRIORITY const    /* simulation priority */

enabler  : PROVIDED '(' expr ')' /* execution constraint */

visible  : HIDDEN | SHOW

sequence : step [ ';' step ] *

step     : stmnt [ UNLESS stmnt ]
           | decl_lst
           | XR varref [',' varref ] *
           | XS varref [',' varref ] *

ivar     : name [ '[' const ']' ] [ '=' any_expr | '=' ch_init ]

ch_init  : '[' const ']' OF '{' typename [ ',' typename ] * '}'

varref   : name [ '[' any_expr ']' ] [ '.' varref ]

send     : varref '!' send_args        /* normal fifo send */
           | varref '!' '!' send_args  /* sorted send */

receive  : varref '?' recv_args                /* normal receive */
           | varref '?' '?' recv_args          /* random receive */
           | varref '?' '<' recv_args '>'      /* poll with side-effect */
           | varref '?' '?' '<' recv_args '>'  /* ditto */

poll     : varref '?' '[' recv_args ']' /* poll without side-effect */
           | varref '?' '?' '[' recv_args ']'  /* ditto */

send_args: arg_lst | any_expr '(' arg_lst ')'

arg_lst  : any_expr [ ',' any_expr ] *

recv_args: recv_arg [ ',' recv_arg ] *  |  recv_arg '(' recv_args
')'

recv_arg : varref | EVAL '(' varref ')' | [ '-' ] const

assign   : varref '=' any_expr   /* standard assignment */
           | varref '+' '+'      /* increment */
           | varref '-' '-'      /* decrement */

stmnt    : IF options FI               /* selection */
           | DO options OD             /* iteration */
           | ATOMIC '{' sequence '}'   /* atomic sequence */
           | D_STEP '{' sequence '}'   /* deterministic atomic */
           | '{' sequence '}'          /* normal sequence */
           | send
           | receive
           | assign
           | ELSE                      /* used inside options */
           | BREAK                     /* used inside iterations */
           | GOTO name
           | name ':' stmnt            /* labeled statement */
           | PRINT '(' string [ , arg_lst ] ')'
           | ASSERT expr
           | expr                      /* condition */

options  : ':' ':' sequence [ ':' ':' sequence ] *

andor    : '&' '&' | '|' '|'

binarop  : '+' | '-' | '*' | '/' | '%' | '&' | '^' | '|'
           | '>' | '<' | '>' '=' | '<' '=' | '=' '=' | '!' '='
           | '<' '<' | '>' '>' | andor

unarop   : '~' | '-' | '!'

any_expr : '(' any_expr ')'
           | any_expr binarop any_expr
           | unarop any_expr
           | '(' any_expr '-' '>' any_expr ':' any_expr ')'
           | LEN '(' varref ')'                /* nr of messages in chan */
           | poll
           | varref
           | const
           | TIMEOUT
           | NP_                               /* non-progress system state */
           | ENABLED '(' any_expr ')'          /* refers to a pid */
           | PC_VAL '(' any_expr ')'           /* refers to a pid */
           | name '[' any_expr ']' '@' name    /* refers to a pid */
           | RUN name '(' [ arg_lst ] ')' [ priority ]

expr     : any_expr
           | '(' expr ')'
           | expr andor expr
           | chanpoll '(' varref ')'   /* may not be negated */

chanpoll : FULL | EMPTY | NFULL | NEMPTY

string   : '"' [ any_ascii_char ] * '"'

uname    : name

name     : alpha [ alpha | number ] *

const    : TRUE | FALSE | SKIP | number [ number ] *
```

```
alpha   : 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j'
          | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't'
          | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'
          | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J'
          | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T'
          | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'
          | '_'

number  : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

E.2 LTL

The syntax of the temporal logics LTL is given by the following context-free grammar:

$$\begin{array}{lcl}
\phi & ::= & \bot \\
\phi & ::= & \top \\
\phi & ::= & p \\
\phi & ::= & \neg\phi \\
\phi & ::= & \phi \wedge \phi \\
\phi & ::= & \phi \vee \phi \\
\phi & ::= & \phi \rightarrow \phi \\
\phi & ::= & \phi \leftrightarrow \phi \\
\phi & ::= & (\bigcirc\phi) \\
\phi & ::= & (\Diamond\ \phi) \\
\phi & ::= & (\Box\ \phi) \\
\phi & ::= & (\phi\ \mathcal{U}\ \phi)
\end{array}$$

A LTL formula ϕ is evaluated over a *path*, or over a *set of paths*. A set of paths S satisfies ϕ if and only if all paths $\pi \in S$ satisfy ϕ. The satisfaction relation $\models$ is defined inductively on the structure of the formula. For the LTL semantics defined below, consider a path $\pi = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$ where π^i is the suffix of π which begins at state s_i (i.e., $\pi^i = s_i \rightarrow s_{i+1} \rightarrow s_{i+2} \rightarrow \ldots$.

π	$\models$	$\top$	
π	$\models$	p	iff $p \in L(s_1)$.
π	$\models$	$\neg\phi$	iff $\pi \models \phi$ is false.
π	$\models$	$\phi \wedge \psi$	iff $\pi \models \phi$ and $\pi \models \psi$.
π	$\models$	$\phi \vee \psi$	iff $\pi \models \phi$ or $\pi \models \psi$.
π	$\models$	$\phi \rightarrow \psi$	iff $\pi \models \neg\phi$ or $\pi \models \psi$.
π	$\models$	$\phi \leftrightarrow \psi$	iff $\pi \models \phi \rightarrow \psi$ or $\pi \models \psi \rightarrow \phi$.
π	$\models$	$(\bigcirc\ \phi)$	iff $\pi^2 \models \phi$.
π	$\models$	$(\Diamond\ \phi)$	iff $\pi^i \models \phi$ for some $i \geqslant 1$.
π	$\models$	$(\Box\phi)$	iff $\pi^i \models \phi$ for all $i \geqslant 1$.
π	$\models$	$(\phi\ \mathcal{U}\ \psi)$	iff $\pi^i \models \psi$ for some $i \geqslant 1$, and for all $j = 1, \ldots, i-1$ we have that $\pi^j \models \phi$.

E.3 Translated Dining Philosophers problem program

The first step to translated the object-oriented specification given in Figures 4.9, 4.11, 4.10, 4.13, 4.8, 4.12 is to establish a total order into the messages and attributes of all classes belonging to the program class-model graph. The table below presents those orders:

Class	**Message**	**Parameter**	**Attribute**
Philosopher	(1) Thinking	–	(1) isAt
	(2) Eat	–	(2) rightFork
	(3) Eating	–	(3) leftFork
	(4) Got	(1) Fork	
LeftHandedPhilosopher	(1) Thinking	–	(1) isAt
	(2) Eat	–	(2) rightFork
	(3) Eating	–	(3) leftFork
	(4) Got	(1) Fork	
RightHandedPhilosopher	(1) Thinking	–	(1) isAt
	(2) Eat	–	(2) rightFork
	(3) Eating	–	(3) leftFork
	(4) Got	(1) Fork	
Fork	(1) Release	(1) Table	(1) owner
		(2) Philosopher	
	(2) Acquire	(1) Philosopher	
Table	–	–	–
ForkHolder	–	–	–

Next, we present (part of) the translated Promela program, generated from Section 4.4. We only show the code concerning the objects actually used in the initial graph, i.e., `Fork` and `RightHandedPhilosopher`. The other ones are similar or do not perform any computations, so we left them out.

```
#define BSIZE 4
#define SIZE_INHERITANCE 6

 mtype = {class_ForkHolder, class_Philosopher, class_LeftHandedPhilosopher,
          class_RightHandedPhilosopher, class_Table, class_Fork};

 mtype = {msg_Fork_Release, msg_Fork_Acquire, msg_Philosopher_Thinking, msg_Philosopher_Eating,
          msg_Philosopher_Eat, msg_Philosopher_Got, msg_LeftHandedPhilosopher_Thinking,
          msg_LeftHandedPhilosopher_Eating, msg_LeftHandedPhilosopher_Eat, msg_LeftHandedPhilosopher_Got,
          msg_RightHandedPhilosopher_Thinking, msg_RightHandedPhilosopher_Eating,
          msg_RightHandedPhilosopher_Eat, msg_RightHandedPhilosopher_Got};

 mtype = {rule_Fork_AcquireFork, rule_Fork_ReleaseFork, rule_Philosopher_StopEating,
          rule_Philosopher_StopThinking, rule_LeftHandedPhilosopher_StopEating,
          rule_LeftHandedPhilosopher_StopThinking, rule_LeftHandedPhilosopher_LHP1stFork,
          rule_LeftHandedPhilosopher_LHP2ndFork, rule_LeftHandedPhilosopher_LHPStartsEating,
          rule_RightHandedPhilosopher_StopEating, rule_RightHandedPhilosopher_StopThinking,
          rule_RightHandedPhilosopher_RHP1stFork, rule_RightHandedPhilosopher_RHP2ndFork,
          rule_RightHandedPhilosopher_RHPStartsEating};

 mtype = {Plato, Socrate, Kant, Hegel, Nietzche, DinnerTable,
          Fork1, Fork2, Fork3, Fork4, Fork5};

typedef object { chan channel; mtype type; mtype id; };
typedef extends { mtype primitive; mtype derived; };

extends inheritance [SIZE_INHERITANCE];
```

```
mtype event_RuleName;
mtype event_ForkHolder;
mtype event_Philosopher;
mtype event_LeftHandedPhilosopher;
mtype event_RightHandedPhilosopher;
mtype event_Table;
mtype event_Fork;

inline match (received, shouldbe, ok, i) {
  i = 0; ok = false;
  if
  :: (received == shouldbe) -> ok = true;
  ::else ->
     do
     :: (i < SIZE_INHERITANCE) ->
        if
          :: (inheritance[i].primitive == shouldbe) && (inheritance[i].derived == received) ->
              ok = true; break;
        fi;
        i++;
     :: else -> break;
     od;
  fi;
}

inline rules_RightHandedPhilosopher () {
 if
 :: (msg_Received == msg_RightHandedPhilosopher_Thinking) ->
     match_StopThinking = true;

     if
     :: match_StopThinking ->
          d_step {
             event_RightHandedPhilosopher = opc_RightHandedPhilosopher.id;
             event_Philosopher            = opc_RightHandedPhilosopher.id;
             event_ForkHolder             = opc_RightHandedPhilosopher.id;
             event_RuleName = rule_RightHandedPhilosopher_StopThinking;
         }
         assert(nfull(opc_RightHandedPhilosopher.channel));
         if
         :: opc_RightHandedPhilosopher.type == class_RightHandedPhilosopher ->
             opc_RightHandedPhilosopher.channel!msg_RightHandedPhilosopher_Eat,
                nil_object;
         fi;
      :: else ->
            if
            :: busy[0] == false ->
               opcb_RightHandedPhilosopher[0]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
               busy[0] = true; inspected[0]=true;
            :: else ->
               if
               :: busy[1] == false ->
                  opcb_RightHandedPhilosopher[1]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
                  busy[1] = true; inspected[1]=true;
               :: else ->
                  if
                  :: busy[2] == false ->
                     opcb_RightHandedPhilosopher[2]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
                     busy[2] = true; inspected[2]=true;
                  :: else ->
                     if
                     :: busy[3] == false ->
                        opcb_RightHandedPhilosopher[3]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
                        busy[3] = true; inspected[3]=true;
                     :: else -> assert(false);
      fi; fi; fi; fi; fi;

  :: (msg_Received == msg_RightHandedPhilosopher_Eat) ->
      match_RHP1stFork = true;
      match (attr_RightHandedPhilosopher_Fork_1_2.type, class_Fork, match_tmp, index);
      match_RHP1stFork = match_RHP1stFork && match_tmp;
      if
```

```
    :: match_RHP1stFork ->
        d_step {
           event_RightHandedPhilosopher = opc_RightHandedPhilosopher.id;
           event_Philosopher            = opc_RightHandedPhilosopher.id;
           event_ForkHolder             = opc_RightHandedPhilosopher.id;
           event_RuleName               = rule_RightHandedPhilosopher_RHP1stFork;
       }
       assert(nfull(attr_RightHandedPhilosopher_Fork_1_2.channel));
       if
       :: attr_RightHandedPhilosopher_Fork_1_2.type == class_Fork ->
          attr_RightHandedPhilosopher_Fork_1_2.channel!msg_Fork_Acquire,
             nil_object, nil_object, opc_RightHandedPhilosopher;
       fi;
    :: else ->
          if
          :: busy[0] == false ->
             opcb_RightHandedPhilosopher[0]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
             busy[0] = true; inspected[0]=true;
          :: else ->
             if
             :: busy[1] == false ->
                opcb_RightHandedPhilosopher[1]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
                busy[1] = true; inspected[1]=true;
             :: else ->
                if
                :: busy[2] == false ->
                   opcb_RightHandedPhilosopher[2]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
                   busy[2] = true; inspected[2]=true;
                :: else ->
                   if
                   :: busy[3] == false ->
                      opcb_RightHandedPhilosopher[3]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
                      busy[3] = true; inspected[3]=true;
                   :: else -> assert(false);
    fi; fi; fi; fi; fi;

:: (msg_Received == msg_RightHandedPhilosopher_Eating) ->
    match_StopEating = true;
    match (attr_RightHandedPhilosopher_Table_1_1.type, class_Table, match_tmp, index);
    match_StopEating = match_StopEating && match_tmp;
    match (attr_RightHandedPhilosopher_Fork_1_2.type, class_Fork, match_tmp, index);
    match_StopEating = match_StopEating && match_tmp;
    match (attr_RightHandedPhilosopher_Fork_1_3.type, class_Fork, match_tmp, index);
    match_StopEating = match_StopEating && match_tmp;
    if
    :: match_StopEating ->
       d_step {
           event_RightHandedPhilosopher = opc_RightHandedPhilosopher.id;
           event_Philosopher            = opc_RightHandedPhilosopher.id;
           event_ForkHolder             = opc_RightHandedPhilosopher.id;
           event_RuleName               = rule_RightHandedPhilosopher_StopEating;
       }
       assert(nfull(attr_RightHandedPhilosopher_Fork_1_2.channel));
       if
       :: attr_RightHandedPhilosopher_Fork_1_2.type == class_Fork ->
          attr_RightHandedPhilosopher_Fork_1_2.channel!msg_Fork_Release,
             attr_RightHandedPhilosopher_Table_1_1, opc_RightHandedPhilosopher, nil_object;
       fi;
       assert(nfull(attr_RightHandedPhilosopher_Fork_1_3.channel));
       if
       :: attr_RightHandedPhilosopher_Fork_1_3.type == class_Fork ->
          attr_RightHandedPhilosopher_Fork_1_3.channel!msg_Fork_Release,
             attr_RightHandedPhilosopher_Table_1_1, opc_RightHandedPhilosopher, nil_object;
       fi;
    :: else ->
       if
       :: busy[0] == false ->
          opcb_RightHandedPhilosopher[0]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
          busy[0] = true; inspected[0]=true;
       :: else ->
          if
          :: busy[1] == false ->
             opcb_RightHandedPhilosopher[1]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
             busy[1] = true; inspected[1]=true;
          :: else ->
             if
             :: busy[2] == false ->
                opcb_RightHandedPhilosopher[2]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
                busy[2] = true; inspected[2]=true;
             :: else ->
                if
                :: busy[3] == false ->
                   opcb_RightHandedPhilosopher[3]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
                   busy[3] = true; inspected[3]=true;
                :: else -> assert(false);
       fi; fi; fi; fi; fi;

:: (msg_Received == msg_RightHandedPhilosopher_Got) ->
    match_RHP2ndFork = true;
    match (attr_RightHandedPhilosopher_Fork_1_2.type, class_Fork, match_tmp, index);
    match_RHP2ndFork = match_RHP2ndFork && match_tmp;
    match (attr_RightHandedPhilosopher_Fork_1_3.type, class_Fork, match_tmp, index);
    match_RHP2ndFork = match_RHP2ndFork && match_tmp;
    match_RHP2ndFork = match_RHP2ndFork &&
     (attr_RightHandedPhilosopher_Fork_1_2.channel == par_RightHandedPhilosopher_Got_Fork_1.channel);

    match_RHPStartsEating = true;
    match (attr_RightHandedPhilosopher_Fork_1_3.type, class_Fork, match_tmp, index);
    match_RHPStartsEating = match_RHPStartsEating && match_tmp;
    match_RHPStartsEating = match_RHPStartsEating &&
     (attr_RightHandedPhilosopher_Fork_1_3.channel == par_RightHandedPhilosopher_Got_Fork_1.channel);

    if
    :: match_RHP2ndFork ->
       d_step {
           event_RightHandedPhilosopher = opc_RightHandedPhilosopher.id;
           event_Philosopher            = opc_RightHandedPhilosopher.id;
           event_ForkHolder             = opc_RightHandedPhilosopher.id;
           event_RuleName               = rule_RightHandedPhilosopher_RHP2ndFork;
       }
       assert(nfull(attr_RightHandedPhilosopher_Fork_1_3.channel));
       if
       :: attr_RightHandedPhilosopher_Fork_1_3.type == class_Fork ->
            attr_RightHandedPhilosopher_Fork_1_3.channel!msg_Fork_Acquire,
                  nil_object, nil_object, opc_RightHandedPhilosopher;
       fi;

    :: match_RHPStartsEating ->
       d_step {
           event_RightHandedPhilosopher = opc_RightHandedPhilosopher.id;
           event_Philosopher            = opc_RightHandedPhilosopher.id;
           event_ForkHolder             = opc_RightHandedPhilosopher.id;
           event_RuleName               = rule_RightHandedPhilosopher_RHPStartsEating;
       }
       assert(nfull(opc_RightHandedPhilosopher.channel));
       opc_RightHandedPhilosopher.channel!msg_RightHandedPhilosopher_Eating, nil_object;
    :: else ->
       if
       :: busy[0] == false ->
          opcb_RightHandedPhilosopher[0]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
          busy[0] = true; inspected[0]=true;
       :: else ->
          if
          :: busy[1] == false ->
             opcb_RightHandedPhilosopher[1]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
             busy[1] = true; inspected[1]=true;
          :: else ->
             if
             :: busy[2] == false ->
                   opcb_RightHandedPhilosopher[2]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
                   busy[2] = true; inspected[2]=true;
             :: else ->
                   if
                   :: busy[3] == false ->
                   opcb_RightHandedPhilosopher[3]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
                   busy[3] = true; inspected[3]=true;
```

```
                 :: else -> assert(false);
    fi; fi; fi; fi; fi;
 :: else ->
    if
    :: busy[0] == false ->
       opcb_RightHandedPhilosopher[0]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
       busy[0] = true; inspected[0]=true;
    :: else ->
       if
       :: busy[1] == false ->
          opcb_RightHandedPhilosopher[1]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
          busy[1] = true; inspected[1]=true;
       :: else ->
          if
          :: busy[2] == false ->
             opcb_RightHandedPhilosopher[2]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
             busy[2] = true; inspected[2]=true;
          :: else ->
             if
             :: busy[3] == false ->
                opcb_RightHandedPhilosopher[3]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
                busy[3] = true; inspected[3]=true;
             :: else -> assert(false);
 fi; fi; fi; fi; fi;
}


proctype RightHandedPhilosopher (object opc_RightHandedPhilosopher;
                                 object attr_RightHandedPhilosopher_Table_1_1;
                                 object attr_RightHandedPhilosopher_Fork_1_2;
                                 object attr_RightHandedPhilosopher_Fork_1_3) {
 mtype  msg_Received;
 object nil_object;
 bool   match_tmp;

 bool   match_StopEating;
 bool   match_StopThinking;
 bool   match_RHP1stFork;
 bool   match_RHP2ndFork;
 bool   match_RHPStartsEating;

 int    index;

 object par_RightHandedPhilosopher_Got_Fork_1;

 object tmp_attr_RighttHandedPhilosopher_ForkHolder_1_1;
 object tmp_attr_RightHandedPhilosopher_Fork_1_2;
 object tmp_attr_RightHandedPhilosopher_Fork_1_3;
 bool   busy [BSIZE] = false;
 bool   inspected[BSIZE] = false;
 chan   opcb_RightHandedPhilosopher [BSIZE] = [1] of {mtype, object};

 RCV: opc_RightHandedPhilosopher.channel?msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
      opc_RightHandedPhilosopher.channel!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;

 do
 :: (len(opc_RightHandedPhilosopher.channel) > 0) ->
     inspected [0] = false;
     inspected [1] = false;
     inspected [2] = false;
     inspected [3] = false;
     if
     :: busy[0] == false ->
         opc_RightHandedPhilosopher.channel?msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
         opcb_RightHandedPhilosopher[0]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
         busy[0] = true;
     :: else ->
        if
        :: busy[1] == false ->
            opc_RightHandedPhilosopher.channel?msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
            opcb_RightHandedPhilosopher[1]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
            busy[1] = true;
        :: else ->
```

```
          if
          :: busy[2] == false ->
              opc_RightHandedPhilosopher.channel?msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
              opcb_RightHandedPhilosopher[2]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
              busy[2] = true;
          :: else ->
             if
             :: busy[3] == false ->
                opc_RightHandedPhilosopher.channel?msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
                opcb_RightHandedPhilosopher[3]!msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
                busy[3] = true;
             :: else -> assert(false);
          fi; fi; fi; fi;
 :: (busy[0] == true && inspected[0]==false) -> atomic {
      opcb_RightHandedPhilosopher[0]?msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
      busy[0] = false;
      rules_RightHandedPhilosopher(); }
 :: (busy[1] == true && inspected[1]==false) -> atomic {
      opcb_RightHandedPhilosopher[1]?msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
      busy[1] = false;
      rules_RightHandedPhilosopher(); }
 :: (busy[2] == true && inspected[2]==false) -> atomic {
      opcb_RightHandedPhilosopher[2]?msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
      busy[2] = false;
      rules_RightHandedPhilosopher(); }
 :: (busy[3] == true && inspected[3]==false) -> atomic {
      opcb_RightHandedPhilosopher[3]?msg_Received, par_RightHandedPhilosopher_Got_Fork_1;
      busy[3] = false;
      rules_RightHandedPhilosopher(); }
 :: else -> goto RCV;
 od;
}

inline rules_Fork () {
 if
 :: (msg_Received == msg_Fork_Release) ->
     match_ReleaseFork = true;
     match (attr_Fork_ForkHolder_1_1.type, class_Philosopher, match_ReleaseFork, index);
     match_ReleaseFork = match_ReleaseFork &&
               (attr_Fork_ForkHolder_1_1.channel == par_Fork_Release_Philosopher_2.channel);
     if
     :: match_ReleaseFork ->
         d_step {
             event_RuleName = rule_Fork_ReleaseFork;
             event_Fork = opc_Fork.id;
         }
         tmp_attr_Fork_ForkHolder_1_1.channel = par_Fork_Release_Table_1.channel;
         tmp_attr_Fork_ForkHolder_1_1.type    = par_Fork_Release_Table_1.type;
         tmp_attr_Fork_ForkHolder_1_1.id      = par_Fork_Release_Table_1.id;
         attr_Fork_ForkHolder_1_1.channel     = tmp_attr_Fork_ForkHolder_1_1.channel;
         attr_Fork_ForkHolder_1_1.type        = tmp_attr_Fork_ForkHolder_1_1.type;
         attr_Fork_ForkHolder_1_1.id          = tmp_attr_Fork_ForkHolder_1_1.id;
     :: else ->
        if
        :: busy[0] == false ->
           opcb_Fork[0]!msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                      par_Fork_Acquire_Philosopher_1;
           busy[0] = true; inspected[0]=true;
        :: else ->
           if
           :: busy[1] == false ->
              opcb_Fork[1]!msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                         par_Fork_Acquire_Philosopher_1;
              busy[1] = true; inspected[1]=true;
           :: else ->
              if
              :: busy[2] == false ->
                 opcb_Fork[2]!msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                            par_Fork_Acquire_Philosopher_1;
                 busy[2] = true; inspected[2]=true;
              :: else ->
                 if
                 :: busy[3] == false ->
```

```
                  opcb_Fork[3]!msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                           par_Fork_Acquire_Philosopher_1;
                  busy[3] = true; inspected[3]=true;
               :: else -> assert(false);
         fi; fi; fi; fi; fi;

 :: (msg_Received == msg_Fork_Acquire) ->
     match_AcquireFork = true;
     match (attr_Fork_ForkHolder_1_1.type, class_Table, match_AcquireFork, index);
     if
     :: match_AcquireFork ->
         d_step {
             event_RuleName = rule_Fork_AcquireFork;
             event_Fork = opc_Fork.id;
         }
         tmp_attr_Fork_ForkHolder_1_1.channel = par_Fork_Acquire_Philosopher_1.channel;
         tmp_attr_Fork_ForkHolder_1_1.type    = par_Fork_Acquire_Philosopher_1.type;
         tmp_attr_Fork_ForkHolder_1_1.id      = par_Fork_Acquire_Philosopher_1.id;
         attr_Fork_ForkHolder_1_1.channel     = tmp_attr_Fork_ForkHolder_1_1.channel;
         attr_Fork_ForkHolder_1_1.type        = tmp_attr_Fork_ForkHolder_1_1.type;
         attr_Fork_ForkHolder_1_1.id          = tmp_attr_Fork_ForkHolder_1_1.id;
         assert (nfull(par_Fork_Acquire_Philosopher_1.channel));
         if
         :: par_Fork_Acquire_Philosopher_1.type == class_Philosopher ->
             par_Fork_Acquire_Philosopher_1.channel!msg_Philosopher_Got, opc_Fork;
         :: par_Fork_Acquire_Philosopher_1.type == class_LeftHandedPhilosopher ->
             par_Fork_Acquire_Philosopher_1.channel!msg_LeftHandedPhilosopher_Got, opc_Fork;
         :: par_Fork_Acquire_Philosopher_1.type == class_RightHandedPhilosopher ->
             par_Fork_Acquire_Philosopher_1.channel!msg_RightHandedPhilosopher_Got, opc_Fork;
         fi;
     :: else ->
        if
        :: busy[0] == false ->
           opcb_Fork[0]!msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                      par_Fork_Acquire_Philosopher_1;
           busy[0] = true; inspected[0]=true;
        :: else ->
           if
           :: busy[1] == false ->
              opcb_Fork[1]!msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                         par_Fork_Acquire_Philosopher_1;
              busy[1] = true; inspected[1]=true;
           :: else ->
              if
              :: busy[2] == false ->
                 opcb_Fork[2]!msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                            par_Fork_Acquire_Philosopher_1;
                 busy[2] = true; inspected[2]=true;
              :: else ->
                 if
                 :: busy[3] == false ->
                    opcb_Fork[3]!msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                               par_Fork_Acquire_Philosopher_1;
                    busy[3] = true; inspected[3]=true;
                 :: else -> assert(false);
     fi; fi; fi; fi; fi;
    :: else ->
       if
       :: busy[0] == false ->
          opcb_Fork[0]!msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                           par_Fork_Acquire_Philosopher_1;
          busy[0] = true; inspected[0]=true;
       :: else ->
          if
          :: busy[1] == false ->
             opcb_Fork[1]!msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                        par_Fork_Acquire_Philosopher_1;
             busy[1] = true; inspected[1]=true;
          :: else ->
             if
             :: busy[2] == false ->
                opcb_Fork[2]!msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                                 par_Fork_Acquire_Philosopher_1;
              busy[2] = true; inspected[2]=true;
           :: else ->
              if
              :: busy[3] == false ->
                 opcb_Fork[3]!msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                            par_Fork_Acquire_Philosopher_1;
                 busy[3] = true; inspected[3]=true;
              :: else -> assert(false);
   fi; fi; fi; fi; fi;
}


proctype Fork (object opc_Fork; object attr_Fork_ForkHolder_1_1) {
   mtype  msg_Received;
   object nil_object;

   bool   match_AcquireFork;
   bool   match_ReleaseFork;

   int    index;

   object par_Fork_Release_Table_1;
   object par_Fork_Release_Philosopher_2;
   object par_Fork_Acquire_Philosopher_1;

   object tmp_attr_Fork_ForkHolder_1_1;

   bool   busy [BSIZE] = false;
   bool   inspected [BSIZE] = false;
   chan   opcb_Fork [BSIZE] = [1] of {mtype, object, object, object};

   RCV: opc_Fork.channel?msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                  par_Fork_Acquire_Philosopher_1;
        opc_Fork.channel!msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                  par_Fork_Acquire_Philosopher_1;
   do
   :: (len(opc_Fork.channel) > 0) ->
       inspected [0] = false;
       inspected [1] = false;
       inspected [2] = false;
       inspected [3] = false;
       if
       :: busy[0] == false ->
           opc_Fork.channel?msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                  par_Fork_Acquire_Philosopher_1;
           opcb_Fork[0]!msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                  par_Fork_Acquire_Philosopher_1;
           busy[0] = true;
       :: else ->
          if
          :: busy[1] == false ->
              opc_Fork.channel?msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                  par_Fork_Acquire_Philosopher_1;
              opcb_Fork[1]!msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                  par_Fork_Acquire_Philosopher_1;
              busy[1] = true;
          :: else ->
             if
             :: busy[2] == false ->
                 opc_Fork.channel?msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                        par_Fork_Acquire_Philosopher_1;
                 opcb_Fork[2]!msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                        par_Fork_Acquire_Philosopher_1;
                 busy[2] = true;
             :: else ->
                if
                :: busy[3] == false ->
                    opc_Fork.channel?msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                           par_Fork_Acquire_Philosopher_1;
                    opcb_Fork[3]!msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                           par_Fork_Acquire_Philosopher_1;
                    busy[3] = true;
                :: else -> assert(false);
        fi; fi; fi; fi;
   :: (busy[0] == true && inspected[0]==false) -> atomic {
        opcb_Fork[0]?msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                       par_Fork_Acquire_Philosopher_1;
        busy[0] = false;
        rules_Fork();
       }
   :: (busy[1] == true && inspected[1]==false) -> atomic {
        opcb_Fork[1]?msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                       par_Fork_Acquire_Philosopher_1;
        busy[1] = false;
        rules_Fork();
       }
   :: (busy[2] == true && inspected[2]==false) -> atomic {
        opcb_Fork[2]?msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                       par_Fork_Acquire_Philosopher_1;
        busy[2] = false;
        rules_Fork();
       }
   :: (busy[3] == true && inspected[3]==false) -> atomic {
        opcb_Fork[3]?msg_Received, par_Fork_Release_Table_1, par_Fork_Release_Philosopher_2,
                                       par_Fork_Acquire_Philosopher_1;
        busy[3] = false;
        rules_Fork();
       }
   :: else -> goto RCV;
   od;
}

init {

   atomic {

       object nil_object;

       inheritance[0].primitive = class_ForkHolder;
       inheritance[0].derived   = class_Table;

       inheritance[1].primitive = class_ForkHolder;
       inheritance[1].derived   = class_Philosopher;

       inheritance[2].primitive = class_Philosopher;
       inheritance[2].derived   = class_LeftHandedPhilosopher;

       inheritance[3].primitive = class_Philosopher;
       inheritance[3].derived   = class_RightHandedPhilosopher;

       inheritance[4].primitive = class_ForkHolder;
       inheritance[4].derived   = class_LeftHandedPhilosopher;

       inheritance[5].primitive = class_ForkHolder;
       inheritance[5].derived   = class_RightHandedPhilosopher;

       chan channel_Socrate  = [BSIZE] of {mtype, object};
       chan channel_Plato    = [BSIZE] of {mtype, object};
       chan channel_Nietzche = [BSIZE] of {mtype, object};
       chan channel_Hegel    = [BSIZE] of {mtype, object};
       chan channel_Kant     = [BSIZE] of {mtype, object};

       chan channel_Fork1 = [BSIZE] of {mtype, object, object, object};
       chan channel_Fork2 = [BSIZE] of {mtype, object, object, object};
       chan channel_Fork3 = [BSIZE] of {mtype, object, object, object};
       chan channel_Fork4 = [BSIZE] of {mtype, object, object, object};
       chan channel_Fork5 = [BSIZE] of {mtype, object, object, object};

       chan channel_DinnerTable = [BSIZE] of {mtype};

       object obj_Socrate;
       obj_Socrate.channel = channel_Socrate;
       obj_Socrate.type = class_RightHandedPhilosopher;
       obj_Socrate.id = Socrate;

       object obj_Plato;
```

```
        obj_Plato.channel = channel_Plato;
        obj_Plato.type = class_RightHandedPhilosopher;
        obj_Plato.id = Plato;

        object obj_Nietzche;
        obj_Nietzche.channel = channel_Nietzche;
        obj_Nietzche.type = class_RightHandedPhilosopher;
        obj_Nietzche.id = Nietzche;

        object obj_Hegel;
        obj_Hegel.channel = channel_Hegel;
        obj_Hegel.type = class_RightHandedPhilosopher;
        obj_Hegel.id  = Hegel;

        object obj_Kant;
        obj_Kant.channel = channel_Kant;
        obj_Kant.type = class_RightHandedPhilosopher;
        obj_Kant.id = Kant;

        object obj_Fork1;
        obj_Fork1.channel = channel_Fork1; obj_Fork1.type = class_Fork; obj_Fork1.id = Fork1;

        object obj_Fork2;
        obj_Fork2.channel = channel_Fork2; obj_Fork2.type = class_Fork; obj_Fork2.id = Fork2;

        object obj_Fork3;
        obj_Fork3.channel = channel_Fork3; obj_Fork3.type = class_Fork; obj_Fork3.id  = Fork3;

        object obj_Fork4;
        obj_Fork4.channel = channel_Fork4; obj_Fork4.type = class_Fork; obj_Fork4.id = Fork4;

        object obj_Fork5;
        obj_Fork5.channel = channel_Fork5; obj_Fork5.type = class_Fork; obj_Fork5.id  = Fork5;

        object obj_DinnerTable;
        obj_DinnerTable.channel = channel_DinnerTable;
        obj_DinnerTable.type = class_Table;
        obj_DinnerTable.id  = DinnerTable;

        run RightHandedPhilosopher (obj_Socrate,  obj_DinnerTable,  obj_Fork1, obj_Fork2);
        run RightHandedPhilosopher (obj_Plato,    obj_DinnerTable,  obj_Fork2, obj_Fork3);
        run RightHandedPhilosopher (obj_Nietzche, obj_DinnerTable,  obj_Fork3, obj_Fork4);
        run RightHandedPhilosopher (obj_Hegel,    obj_DinnerTable,  obj_Fork4, obj_Fork5);
        run RightHandedPhilosopher (obj_Kant,     obj_DinnerTable,  obj_Fork5, obj_Fork1);

        run Fork (obj_Fork1, obj_DinnerTable);
        run Fork (obj_Fork2, obj_DinnerTable);
        run Fork (obj_Fork3, obj_DinnerTable);
        run Fork (obj_Fork4, obj_DinnerTable);
        run Fork (obj_Fork5, obj_DinnerTable);

        run Table (obj_DinnerTable);

        assert(nfull(obj_Socrate.channel));
        obj_Socrate.channel!msg_RightHandedPhilosopher_Thinking, nil_object;

        assert(nfull(obj_Plato.channel));
        obj_Plato.channel!msg_RightHandedPhilosopher_Thinking, nil_object;

        assert(nfull(obj_Nietzche.channel));
        obj_Nietzche.channel!msg_RightHandedPhilosopher_Thinking, nil_object;

        assert(nfull(obj_Hegel.channel));
        obj_Hegel.channel!msg_RightHandedPhilosopher_Thinking, nil_object;

        assert(nfull(obj_Kant.channel));
        obj_Kant.channel!msg_RightHandedPhilosopher_Thinking, nil_object;

    }
}
```

APPENDIX F GRAMÁTICAS DE GRAFOS ORIENTADOS A OBJETO

F.1 Introdução

Programas de computador são utilizados em quase todas as atividades da vida moderna e deles dependemos de maneira muitas vezes crucial. A complexidade dos sistemas de software atuais fez com que novas técnicas de desenvolvimento emergissem. Contudo, os paradigmas nos quais estas técnicas estão baseadas (em especial orientação a objeto, eventos e concorrência), apesar de tornarem o processo de modelagem e codificação mais rápido, fazem com que as fases de teste e validação se tornem ainda mais complexas. O cenário atual, devido às pressões de tempo de desenvolvimento e qualidade do produto, requer técnicas de desenvolvimento mais ágeis que garantam que o produto final seja consistente e completo em relação à sua especificação, que sejam formais, incrementais, executáveis e utilizáveis por não especialistas em métodos formais.

Visualmente simples, diagramas são geralmente construídos a partir de regras sintáticas bem definidas. Se essa sintaxe também for equipada com uma semântica formal, tem-se uma linguagem visual de especificação que pode ser integrada mais facilmente no processo de desenvolvimento. Diagramas usados em processos de especificação podem ser modelados por *grafos*.

Grafos são estruturas algébricas capazes de transmitir uma quantidade significativa de informação de uma maneira compacta, visual e clara. A especificação de sistemas computacionais com grafos oferece duas vantagens que são, em geral, mutuamente exclusivas: (i) sendo estruturas matemáticas, grafos apresentam uma semântica bem definida e (ii) contando com uma apresentação diagramática, especificações baseadas em grafos podem ser mais facilmente entendidas e produzidas por não especialistas na área de Computação. Assim, técnicas de especificação baseadas em grafos formam uma base sólida para a integração de métodos formais de especificação e verificação de sistemas no processo de desenvolvimento de software (BARESI; PEZZÈ, 2000).

Sistemas baseados em regras, por outro lado, são capazes de descrever computações a partir de transformações locais. Transformações de grafos através de regras é uma maneira de combinar grafos – para descrição de estruturas complexas – com regras, para manipulação destas estruturas. Sistemas de *transformação* (ou *reescrita*) *de grafos* combinam as vantagens de grafos e regras num único paradigma computacional (EHRIG et al., 1996). A *abordagem algébrica para gramáticas de grafos* (EHRIG; PFENDER; SCHNEIDER, 1973) faz uso de construções categoriais para definir os aspectos relevantes das computações de grafos. Esta abordagem é atualmente conhecida como *double-pushout* porque as derivações são baseadas em duas construções do tipo *pushout* na categoria de grafos e morfismos de grafos. Na abordagem *single-pushout* (LÖWE, 1991) uma derivação é caracterizada como uma construção deste mesmo tipo na categoria de grafos e morfismos parciais de grafos. Esta abordagem é uma extensão própria da primeira (EHRIG et al., 1996). A escolha da abordagem algébrica baseou-se na generalidade fornecida pela Teoria de Categorias (MACLANE, 1998), onde vários resultados poderiam ser herdados caso fosse provado que as construções desejadas existiam nas categorias estudadas.

Os princípios por trás do paradigma da orientação a objeto — *encapsulamento de dados e código*, *oclusão da informação*, *herança* e *polimorfismo* — servem perfeitamente aos propósitos do desenvolvimento modular de sistemas, teste distribuído e reutilização de software, necessários para lidar com o tamanho e a complexidade dos sistemas atuais. A *herança* é um mecanismo central na programação orientada a objeto, suportando o projeto incremental de classes e reúso de especificações já escritas. O *polimorfismo* pode ser descrito como a associação de diferentes significados ou usos para algo em contextos diferentes. O polimorfismo de subclasses (CARDELLI; WEGNER, 1985) assegura que um objeto pode pertencer a diferentes classes, não necessariamente disjuntas, tornando-se especialmente importante quando a ligação dinâmica de métodos é implementada, permitindo que a decisão de qual código executar quando um método é chamado possa ser adiada para o tempo de execução. Programas orientados a objeto fazem uso de herança e sobrescrita de métodos para atingir seus objetivos. Sendo assim, deve-se esperar que os formalismos para a construção de especificações orientadas a objeto reflitam estes conceitos, caso contrário aspectos chave do desenvolvimento poderão acabar negligenciados.

Gramáticas de grafos têm sido usadas para especificação de vários tipos de sistemas de software (EHRIG et al., 1997), onde grafos são usados para representar estados e produções de grafos para operações ou transformações destes sistemas (EHRIG; LÖWE, 1993). Tais especificações freqüentemente utilizam grafos rotulados ou tipados para representar diferentes entidades (ANDRIES et al., 1999), (BLOSTEIN; FAHMY; GRBAVEC, 1995), (CORRADINI; MONTANARI; ROSSI, 1996), (DOTTI; RIBEIRO, 2000), (KORFF, 1995), (RIBEIRO, 1996), (TAENTZER, 1996a). Contudo, nem rotulação nem morfismos de tipagem são capazes de traduzir o significado da herança e do polimorfismo em especificações orientadas a objeto (FERREIRA; RIBEIRO, 2003).

Existe na literatura uma vasta quantidade de métodos formais e semi-formais para especificação de sistemas orientados a objeto. O propósito deste trabalho não é indicar uma solução "melhor", visto que este conceito é relativo e dependente do problema em mãos. A contribuição desta tese está focada na área de gramática de grafos, e nos seu uso para especificação e verificação formal de sistemas orientados a objeto. As contribuições principais desta tese resumem-se como:

- o desenvolvimento de uma extensão para a abordagem *single-pushout* de gramáticas de grafos — gramáticas de grafos orientados a objeto ou *object-oriented graph grammars* — que abarque as características principais (herança, polimorfismo, ligação dinâmica, encapsulamento e oclusão da informação) da orientação a objeto;

- o desenvolvimento de uma semântica observacional baseada em sistemas de transições rotulados pelos eventos (aplicações de regras) sobre elementos (objetos) do grafo do sistema, no caso de ambos fazerem parte do conjunto de entidades visíveis do sistema, fazendo com que computações silenciosas e computações observáveis de uma gramática de grafos orientados a objeto possam ser descritas;

- uma tradução definida formalmente de modelos expressos neste novo formalismo para programas Promela (a linguagem de entrada do verificador de modelos (CLARKE; GRUMBERG; PELED, 1999) SPIN (HOLZMANN, 1997)), para permitir a verificação formal de propriedades (escritas em lógica temporal LTL (HUTH; RYAN, 2000)) sobre estados e eventos da especificação construída.

Dada uma especificação de sistema, pode-se estar interessado em duas coisas: o sistema em si ou o seu comportamento. A primeira parte deste trabalho garante os meios para a construção de especificações orientadas a objeto executáveis enquanto que a segunda provê o ferramental necessário para a análise de propriedades dos sistemas assim especificados. O restante deste capítulo está organizado da seguinte forma: a Seção F.2 apresenta os principais resultados relacionados com a modelagem estática de sistemas orientados a objeto; a Seção F.3 discute a computação de sistemas especificados de acordo com o formalismo proposto; a Seção F.4 esboça como a verificação de propriedades de especificações é realizada. As conclusões são apresentadas na Seção F.5.

F.2 Especificações de sistemas orientados a objeto

Uma hierarquia de classes de um modelo orientado a objeto é, neste trabalho, representada por um hipergrafo rotulado denominado *grafo de classes* (*class-model graph*). A novidade desta abordagem consiste na exigência que ambos os conjuntos de nodos (classes) e arcos (atributos e mensagens) do grafo sejam conjuntos parcialmente ordenados. Hiperarcos podem ser rotulados como *mensagens* ou *atributos*. Um hiperarco de tipo mensagem tem somente um nodo destino (o objeto ao qual a mensagem em questão se destina), mas pode ter qualquer número (finito) de nodos de origem, que correspondem aos parâmetros da mensagem. Um arco de tipo mensagem pode ser visto, então, como a especificação da assinatura de um método. Já um hiperarco de tipo atributo possui somente um nodo de origem (o objeto ao qual o atributo pertence), e um número qualquer de nodos destino. Um atributo com mais do que um nodo de destino representa um agrupamento de dados numa única estrutura sintática.

As relações subjacentes aos já mencionados conjuntos de nodos e arcos de um grafo de classes são denominadas *relações estritas* (*strict relations*). Uma relação estrita pode ser visualmente modelada por um conjunto de árvores, que representa as possíveis hierarquias de classes em programas escritos em linguagens orientadas a objeto que possuem somente herança simples. Formalmente, uma relação estrita é irreflexiva, acíclica, funcional e nenhuma cadeia com respeito à relação é infinita (embora a relação em si possa ser). Mostra-se que o fecho transitivo e reflexivo de uma relação estrita é uma relação de ordem parcial, o que é importante para herdar toda a teoria já desenvolvida neste domínio. Relações estritas possuem várias propriedades, e mostra-se que elas formam uma categoria, juntamente com seus morfismos. Morfismos de relações estritas são tais que "colam" dois conjuntos ordenados de maneira que possam ser combinados de forma consistente. Apesar desta categoria não ser cocompleta, colimites binários podem ser construídos a partir de duas relações estritas e de um morfismo entre elas.

Grafos de classes são estruturas algébricas e podem, por esta razão, ser relacionados através de morfismos. Um morfismo de grafo de classes é um morfismo usual de hipergrafos rotulados em que as funções que mapeiam nodos e arcos são morfismos de relações estritas. Mostra-se neste trabalho que grafos de classes e seus morfismos formam uma categoria. Esta categoria não é cocompleta uma vez que coequalizadores de morfismos de seus conjuntos de nodos e arcos não podem ser arbitrariamente construídos. No entanto, grafos de classes podem ser compostos dois a dois, usando um morfismo para identificação dos elementos pertinentes. A composição de sistemas orientados a objeto pode ser formalmente definida como o colimite do diagrama que contém os sistemas que estão sendo compostos e o morfismo que os conecta. Dois elementos do grafo de classes devem ser relacionados pelo morfismo se eles representam o mesmo elemento, que é repetido em duas especificações distintas (seja uma classe inteira ou somente alguns de seus atributos ou métodos). A construção do colimite une os elementos relacionados de forma e que eles sejam identificados na especificação composta. Como um colimite em qualquer categoria é uma estrutura única (a menos de isomorfismo), conclui-se que a composição de sistemas neste contexto é única e bem definida. Mostra-se ainda como as formas mais usuais de extensão de sistemas orientados a objeto, que são a especialização por herança e a agregação de objetos, são casos especiais de composição de grafos de classes. Este resultado permite uma visão consistente e uniforme da extensão de sistemas orientados a objeto, uma vez que as formas existentes para que isso seja efetuado podem ser todas formalizadas pela mesma construção categorial.

F.3 Computações de sistemas orientados a objeto

Gramáticas de grafos orientados a objeto devem modelar o comportamento dinâmico de sistemas orientados a objeto. O ponto de partida para a definição dessas gramáticas são *grafos $\mathcal{C}$-tipados* e seus morfismos. Grafos $\mathcal{C}$-tipados são hipergrafos tipados sobre um grafo de classes, mas o morfismo de tipagem é mais flexível do que o tradicional (CORRADINI; MONTANARI; ROSSI, 1996), no sentido que os hiperarcos mapeados precisam preservar as *relações* existentes entre suas origens e seus destinos e não suas origens e destinos propriamente ditos. Esta flexibilidade permite que a herança de elementos seja implementada, uma vez que os objetos podem fazer uso dos arcos de quaisquer outros objetos dos quais sejam derivados na hierarquia de herança. Morfismos de grafos $\mathcal{C}$-tipados também diferem dos morfismos de grafos tipados tradicionais, no sentido de que os tipos não precisam ser preservados, mas somente a relação entre eles. Essa característica é utilizada para implementação de polimorfismo de subclasses em computações. Mostra-se que grafos $\mathcal{C}$-tipados e seus morfismos formam uma categoria cocompleta $\mathbf{CGraphP}(\mathcal{C})$.

Grafos orientados a objeto são grafos $\mathcal{C}$-tipados restritos. Esta restrição diz respeito ao fato de que todas as mensagens endereçadas a um objeto devem ser tipadas de acordo com o paradigma: um objeto somente pode receber uma mensagem se ela for o elemento mínimo na cadeia de redefinição à qual a mensagem pertence. Esta característica reflete a oclusão de métodos que tenham sido redefinidos em classes derivadas.

A categoria **OOGraphP**($\mathcal{C}$) possui grafos orientados a objeto como objetos e morfismos entre grafos $\mathcal{C}$-tipados como setas. Os mapeamentos entre grafos orientados a objeto asseguram que o polimorfismo de subclasses ocorre automaticamente: sempre que um objeto de uma classe é esperado, um objeto de uma classe dela derivada pode aparecer em seu lugar. O polimorfismo de subclasses é implementado pelas características do morfismo: um nodo x pode ser mapeado para outro nodo y se o tipo de y estiver relacionado com o tipo de x na relação existente entre os nodos no grafo de classes subjacente.

Regras orientadas a objeto respeitam os princípios de oclusão da informação existentes no paradigma. Um objeto, para tratar uma mensagem recebida, somente pode ter conhecimento de seus próprios atributos e dos parâmetros recebidos pela mensagem, ao passo que somente pode modificar valores de seus próprios atributos. A manipulação dos objetos conhecidos deve ser realizada através do envio de mensagens a eles. Adicionalmente, os morfismos de regras orientadas a objeto devem ser invertíveis, o que significa que um objeto não pode ter seu tipo alterado pelas derivações da gramática.

Uma derivação direta (ou aplicação de regra) é definida nos mesmos moldes que derivação em gramáticas de grafos (*single-pushout*) usuais, com exceção da tipagem do grafo resultante da derivação, que é realizada levando em conta as relações de ordem no grafo de classes subjacente. Mostrou-se que uma derivação é o *pushout* dos morfismos correspondentes à regra e à ocorrência do lado esquerdo da regra no grafo do sistema, na categoria **OOGraphP**($\mathcal{C}$). Este resultado é significativo, visto que uma grande parte da teoria algébrica de gramáticas de grafos está suportada sobre o fato de derivações corresponderem a uma construção categorial do tipo (*single-* ou *double-*) *pushout*. Desta forma, todos os resultados já atingidos podem ser imediatamente aproveitados.

Gramáticas de grafos orientados a objeto são o primeiro desenvolvimento conhecido de gramáticas de grafos para especificação de sistemas orientados a objeto que leva em consideração herança, polimorfismo e ligação dinâmica do paradigma de orientação a objeto.

F.4 Verificação de sistemas orientados a objeto

Especificações em gramáticas de grafos orientados a objeto podem ser traduzidas para outras linguagens formais. Neste trabalho é apresentada uma tradução daquelas especificações para programas na linguagem Promela (*PROcess/PROtocol MEta LAnguage*), que é a linguagem de definição de modelos do verificador formal SPIN.

Objetos no grafo inicial da gramática são traduzidos em processos na linguagem Promela, cujos parâmetros reais são os atributos do objeto (isto é, os elementos que são os destinos dos arcos de atributos do objeto em questão). Mensagens são modeladas através de elementos colocados em canais de comunicação associados a cada objeto. A semântica da aplicação de regras é preservada pela escolha não determinística da mensagem a consumir em cada canal. O não determinismo na escolha das mensagens a consumir, uma vez que os canais de comunicação em Promela possuem uma estratégia FIFO (*first in, first out*), é implementado com o auxílio de um buffer de mensagens local a cada processo. As mensagens recebidas são retiradas em ordem do canal de comunicação principal e colocadas num espaço do buffer local através de um escolha não determinística. A retirada das mensagens do buffer local também é não determinística, o que significa que a ordem de recebimento das mensagens não é necessariamente a ordem de seu consumo, como requer o formalismo.

A linguagem Promela, originalmente, não possui nenhum suporte para orientação a objeto. Herança, polimorfismo e ligação dinâmica são codificados no programa Promela de diferentes formas. A hierarquia de herança é codificada em um *array* de pares, que guardam a informação referente ao fecho transitivo da relação de herança pertencente à gramática. Esta informação é global, e portanto visível a todos os objetos. Polimorfismo de subclasses é implementado pela inspeção desta estrutura, para garantir que regras definidas para tipos primitivos possam ser aplicadas a tipos derivados. O disparo de mensagens utiliza reflexão computacional, no sentido de que cada objeto conhece seu próprio tipo (i.e., a classe ao qual pertence), para reproduzir a ligação dinâmica. Como esta informação é pública, o objeto que envia a mensagem pode decidir, em tempo de execução, qual é a mensagem mais adequada em função do tipo do objeto que é o seu destino.

A tradução apresentada neste trabalho garante que a verificação possa ser realizada tanto sobre estados como sobre eventos. Como a maior parte dos verificadores de modelos, SPIN somente permite que fórmulas sobre os estados do modelo sejam definidas. Para permitir a verificação sobre eventos, existem variáveis globais que definem a existência de um evento (aplicação de regra) e a identificação do objeto que sofreu essa ação. Fórmulas da lógica temporal LTL podem ser especificadas sobre os objetos originais do grafo inicial, e sobre as regras aplicáveis em cada objeto. O programa traduzido serve então de fonte para o verificador, que realiza a verificação da propriedade especificada automaticamente. Caso a propriedade não seja verdadeira no modelo, um contra-exemplo gráfico é gerado, a partir do traço de saída do verificador.

A compatibilidade semântica entre a gramática original e o programa traduzido é também discutida. A semântica usada para comparações também foi proposta neste trabalho, e é baseada em sistemas de transição onde algumas entidades são observáveis e outras não. Esta noção de observação em gramática de grafos orientados a objeto é compatível com a execução de programas orientados a objeto, onde em geral não se tem acesso a todos os elementos do programa. Computações da gramática podem ser comparadas com traços de execução do programa Promela traduzido. Argumenta-se que para cada computação da gramática existe um traço correspondente no programa, e que o contrário também é verdadeiro (obedecidas algumas restrições de traços de programa).

F.5 Conclusões

Esta tese apresenta um método formal baseado em grafos para a especificação e verificação de sistemas orientados a objeto. Mais especificamente, uma extensão da abordagem algébrica *single-pushout* para gramáticas de grafos tipadas é proposta para contemplar aspectos concernentes ao paradigma. Os morfismos de tipagem dos grafos respeitam as relações de ordem que definem as relações de herança entre classes (nodos) e sobrescrita de métodos (hiperarcos). Os morfismos entre grafos tipados implementam a noção de polimorfismo de subclasses, e a derivação direta implementa a ligação dinâmica de métodos. O encapsulamento de dados e funções em uma mesma entidade é implementado via restrições estruturais nos grafos de classe. A oclusão da informação é implementada via restrições na estrutura das regras.

Uma semântica observacional para gramática de grafos orientados a objeto, baseada em sistemas de transição, é proposta. Esta semântica é mais abstrata que a usual, é baseada na noção de *entidades visíveis* (objetos ou mensagens) e é compatível com a execução de programas orientados a objeto. A partir desta semântica, é definida uma tradução de especificações escritas em gramática de grafos orientados a objeto para programas Promela. A novidade desta abordagem é a tradução de construções de orientação a objeto para uma linguagem de verificação que não as implementa. A verificação de propriedades, expressas na lógica temporal LTL pode ser conduzida automaticamente usando o verificador SPIN.