Lecture Notes in Computer Science 2917
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Elisa Quintarelli

Model-Checking Based Data Retrieval

An Application to Semistructured and Temporal Data

Springer

Author

Elisa Quintarelli
Politecnico di Milano
Dip. di Elettronica e Informazione
Piazza Leonardo da Vinci 32, 20133 Milano, Italy
E-mail: quintare@elet.polimi.it

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): H.2, H.3, H.4

ISSN 0302-9743
ISBN 3-540-20971-9 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Boller Mediendesign
Printed on acid-free paper SPIN: 10977132 06/3142 5 4 3 2 1 0

To Alessia

Foreword

This thesis deals with the problems of characterizing the semantics of and assuring efficient execution for database query languages, where the database contains semistructured and time-varying information. This area of technology is of much interest and significance for databases and knowledge bases; it also presents many challenging research problems deserving an in-depth investigation. Thus, the topic of Elisa Quintarelli's dissertation is well chosen and totally appropriate to the current research trends.

In her thesis, Elisa addresses a number of related problems. However, her work and contributions concentrate on two main problems. The first is the definition of an effective graph-based approach to the formalization of query languages for semistructured and temporal information. In her approach, query execution is viewed as the process of matching the query graph with the database instance graph; therefore, query execution reduces to searching the database for subgraphs that are similar to the given query graph. The search for such matches can be supported through the computational process of bisimulation. This approach is used to define the semantics of several languages, including graphical languages, such as G-Log and GraphLog, semistructured information languages, such as Lorel, and temporal languages, such as TSS-QL. Both graph-based approaches and bisimulation had been used by previous authors for defining query languages and their semantics; however, this work goes well beyond previous approaches by integrating and refining these techniques into a flexible and powerful paradigm that Elisa demonstrates to be effective on a spectrum of languages and a suite of alternative semantics.

The second research challenge tackled by Elisa in her thesis is that of efficient implementation. This is a nontrivial problem since bisimulation can, in the worst case, incur an exponential time complexity. Her original solution to this difficult problem consists of modeling graphical queries as formulas of modal logic and interpreting database instance graphs as Kripke transition systems. In this way, the problem of solving graphical queries is reduced to the model-checking problem for which efficient decision algorithms exist. In particular, the thesis focuses on CTL formulae for which efficient model checkers are available; this allows Elisa to demonstrate the efficiency of her proposed approach with experimental results.

In conclusion, the thesis represents an interesting piece of research, characterized by novel contributions and an in-depth expertise on several topics. Indeed, the author brings together ideas and techniques from different areas, and displays a solid expertise in computing, in general, and information systems, in particular. In my role as her advisor I had the privilege to see Elisa's growth as a researcher; overall, I am still impressed with the depth and breadth of her work, the originality of her results, and her research maturity. Therefore, I am very happy to see that her Ph.D. thesis has been chosen to be published as a book in these Lecture Notes in Computer Science.

Milan, 10/10/2003 *Letizia Tanca*

Preface

This book contains the research covered by my Ph.D. Thesis, which was developed at the Dipartimento di Elettronica of the Politecnico di Milano, in collaboration with Prof. Agostino Dovier at the Dipartimento di Informatica of the Università di Verona.

The main topic of the book is the study of appropriate, flexible and efficient search techniques for querying semistructured and WWW data, also taking into account the time dimension.

This work was motivated by the fact that in recent years a lot of attention has been given by the database research community to the introduction of methods for representing and querying *semistructured data*. Roughly speaking, this term is used for data that have no absolute schema fixed in advance, and whose structure may be irregular or incomplete.

A common example in which semistructured data arise is when data are stored in sources that do not impose a rigid structure, such as the World Wide Web, or when they are extracted from multiple heterogeneous sources. It is evident that an increasing amount of semistructured data is becoming available to users and, thus, there is a need for Web-enabled applications to access, query and process heterogeneous or semistructured information, flexibly dealing with variations in their structure.

This work proposes a formalization to provide suitable graph-based semantics to languages for semistructured data, supporting both data structure variability and topological similarities between queries and document structures. A suite of semantics based on the notion of bisimulation is introduced both at the concrete level (instances) and at the abstract level (schemata) for a graph-based query language, but the results are general enough to be adapted to other existing proposals as well. Moreover, complexity results on the matching techniques, which are required to find a sort of similarity between a graphical query and a semistructured database, have stimulated our interest in investigating alternative approaches to solve such graphical queries on databases. The main idea here is to solve the data retrieval problem for semistructured data by using techniques and algorithms coming from the model-checking research field: experimental results are presented in this work to confirm the possibility of effectively applying the proposed method based on model-checking algorithms.

The book is structured in the following way: Chap. 1 describes the current scenario, the motivations and the contributions of this work.

Chapter 2 sets the formal content, by presenting G-Log, a graph-based query language showing a three-level semantics based on the notion of bisimulation; the relationships between instances and schemata are investigated by using the theory of abstract interpretation, which provides a systematic approach to guarantee the correctness of operating on schemata with respect to the corresponding concrete computations on instances. We chose G-Log because it is a general graphical language that combines the expressive power of logic, the modeling power of objects, and the representation power of graphs for instances, schemata and logical inferences (i.e., queries and rules). The chapter also describes the main features of two well-known query languages, namely Graph-Log and UnQL, in order to show the applicability and generality of the proposed study to other SQL-like and graphical languages for semistructured data.

Chapter 3 describes the novel idea of this work. We propose an approach to associate a *modal* logic formula to a graphical query, and to interpret database instance graphs as *Kripke Transition Systems* (*KTS*). In this way, the problem of finding subgraphs of the database instance graph that match the query can be performed by using *model-checking* algorithms. In particular, we identify a family of graph-based queries that represent *CTL* formulae. This is very natural and, as an immediate consequence, an effective procedure for efficiently querying semistructured databases can be directly implemented on a model-checker and the query retrieval activity can be performed in polynomial time. In Chap. 3 we focus on a graphical query language very similar to G-Log: we consider only the main constructs of this language that have a natural translation into temporal logic, but we emphasize the possibility of expressing universal conditions in a very compact way. In this chapter we also show some experimental results that confirm our proposal.

Starting with the potential of the *CTL* language for time-based representation, Chap. 4 extends standard techniques for modeling and accessing static information in order to model and retrieve temporal aspects of semistructured data, such as, for example, evolution on simple values contained in semistructured documents. In particular we present a generic graphical data model suitable for representing both static and dynamic aspects of semistructured data, and we introduce a very simple SQL-like query language to compose temporal inferences. In this chapter we do not extend G-Log because it is more natural to express temporal conditions with SQL properties; however, we informally propose the possibility of encoding such properties into graphical queries by adapting the model originally introduced for G-Log in a very natural way. We show also the fragment of this SQL-like language that can be translated into the logic *CTL*, and thus we propose applying model-checking algorithms to solve temporal queries as well. In the same chapter we concentrate our attention not only on the classical notion of the time concept, as it is deeply known in the database field, but we further investigate the possibility of considering more than one dynamic aspect of semistructured data. In particular, we represent with the same data model valid time and interaction time. The first notion is used to consider the validity of facts in the represented reality, whereas the second one is related to user browsing history while navigating Web sites.

Finally, in Chap. 5 we compare this work with others known in the literature, while Chap. 6 summarizes the main results of the thesis and proposes further developments of our piece of research.

Acknowledgments

At the end of this experience it seems only right and proper to give special mention to the people who accompanied me on this long trip.

I would like to take this opportunity to express my deepest thanks to my advisor, Letizia Tanca, for her immense help and encouragement over the last three years. Thanks for always being ready with advice on any possible problem (from scientific questions to very personal problems) and for your efforts to improve this book.

A special thanks to Agostino Dovier for all the things he taught me: how to think and do research, how to write a paper, and how to deal with a formal proof. Thanks for the many corrections to the things we wrote together.

I would like to thank my examiner, Prof. Carlo Zaniolo, for his comments and suggestions on the manuscript.

I am also indebted to Ernesto Damiani for his contribution to some aspects of this work, and to Fabio Grandi for the stimulating discussion on temporal databases during Time 2001. I thank also Carlo Combi for the help he has given me on time-related topics.

I would like to thank Barbara, for being my best friend, for the times she shared with me, for making me laugh on the many occasions she knows very well, and for having convinced me to make some decisions ... Thanks Barbara for all the experiences we shared together during this Ph.D., I will always treasure them.

I thank Nico for challenging me with the first steps in the model-checking field.

I would like to say "thank you" to the Ph.D. students and friends of the DEI department, who have made my life on the first floor most pleasant. Among them, I especially want to mention Vincenzo, Giovanni, Mattia, Matteo, Marco, and Fabio for their enjoyable company, and Sara for her constant support and friendship.

I cannot forget Angela, for her efforts to improve my English, and some special friends: Nadia and Francesca.

I would like to express my gratitude to Rosalba for being a true friend to me.

I would like to give a special thanks to my family: to my parents Federico and Paola for raising me, and for encouraging me to go on with my studies.

This thesis is also for my sisters Giulia and Marta; their constant love and support was very important to me.

Last, but definitely not least, I would like to thank Massimo, for sharing all my achievements with me, for his precious advice in difficult moments, and especially for keeping me connected to the real world; thanks for being with me at all times. A very special thanks to my lovely baby Alessia who makes every day of my life unpredictable.

Milan, 10/10/2003 *Elisa Quintarelli*

Contents

1. Introduction

An increasing amount of data is becoming available electronically to humans and programs. Such data are managed via a large number of data models and access techniques, and may come from relational or object-oriented databases (*structured data*), or consist of simple collections of text or image files (*unstructured data*). Many of them can be seen as *semistructured*: with the term semistructured data we intuitively refer to data whose schema is not fixed in advance, and whose structure may be irregular or incomplete.

We can cite two most common contexts in which semistructured data arise: when data are stored in sources that do not strictly impose a rigid structure, such as the World Wide Web, and when data are combined from several heterogeneous, possibly structured, data sources with different structures.

As a consequence of the large volume and nature of data available to the casual users and programs, studying flexible and efficient techniques for automatically extracting, browsing or querying semistructured data, and in particular data from Web pages, remains a complex and important task in the database community.

In this book we base our work on the use of graph-based representations for semistructured data and queries, and address two specific problems in this data context: the first concerns the study and definition of a suite of semantics for the graph based query language, that allows query composers to relax or restrict topological requirements at their choice. The second main problem addressed is the usage of model-checking based techniques to efficiently solve graphical queries on static and dynamic semistructured documents.

1.1 Motivations

With this work we would like to bring some innovative contributions to semistructured and temporal data management. In particular, we concentrate on a general graph-based data model which offers strong expressive and modeling power, and on a user friendly graphical language. We deeply study a method to add flexibility to the data retrieval activity when static and temporal queries are naturally expressed by labeled graphs. Moreover, we consider also complexity results on the required procedures and therefore focus on a technique based on model-checking algorithms to efficiently solve a subset of the queries.

In the recent years a number of research projects have addressed the problem of accessing in a uniform way *semistructured information* often represented by using data models based on directed labeled graphs. Among these, we can cite LOREL [66], UnQL [12], WG-Log [32], WebSQL [67], WebOQL [7] and, StruQL [50].

Intuitively, the term semistructured data refers to data with (some of) the following features:

– the schema is not given in advance and may be implicit in the data;
– the schema is relatively large w.r.t. the size of the data and may change frequently;
– the schema is descriptive rather than prescriptive, i.e. it describes the current state of the data, but violations of the schema are still tolerated;
– the data are not strongly typed, i.e. for different objects, the values of the same attribute may be of different types.

Effectiveness and efficiency are mandatory requirements when accessing this kind of information; therefore, also in this case appropriate search techniques are more than necessary. Pure keyword-based search techniques have often proved to be ineffective, since in that setting only the document lexicon is taken into account, while the intrinsic semantics conveyed by the document structure is lost; in practice, often this leads to the retrieval of too many documents, since also the ones that do not share the required structure are often included into the result. It is natural to study techniques that go beyond the basic information retrieval paradigm supported by most search engines, and consider also the (partial) structure of documents.

The claim of this work is that, in order to take full advantage of the document structure in the semistructured setting, its hierarchical (or, more generally, topological) organization should be somehow exploited, by introducing some notion of query like the one used in database systems, being still aware of the fact that the document structure is far from being as strict as in the usual database context. In particular, we focus on *flexible* and *efficient* methods for *searching static* and *dynamic information* on the Web.

Indeed, efficient searching techniques can be naturally extended in order to verify not only static properties but also changes in semistructured data. In the past years temporal database management was a productive field of research, that is now covered by textbooks (see for example [48, 3, 99]) and applied in various contexts (see for example the RapidBase system [97]). Several data models, which usually extended the relational one [29, 74, 86], were proposed in order to consider the time-varying nature of data; however, in the semistructured database context there are few works on this research direction, although the problem of representing changes appears more stimulating in this setting for the irregularity, incompleteness and lack of schema that often characterize semistructured data. Moreover, we think that it is worthwhile to investigate the dynamic aspects of this kind of data that arise from different notions of the *time* concept. In particular:

- **Valid Time.** the *valid time* (VT) of a fact is the time when the fact is true in the modeled reality [57].
- **Transaction Time.** the *transaction time* (TT) of a fact is the time when the fact is current in the database and may be retrieved[57].

As a matter of fact, if we restrict our study on semistructured data that arise in the World Wide Web context, we can easily observe that the concept of time has a more general meaning w.r.t. the classical notion used in temporal databases, and does not include only the natural evolution of data through time.

In this work we also investigate another time dimension, strictly related to web data: **Interaction Time** is valid time relative to user browsing. When a user browses through a document (for example a hypermedia representation) they choose a particular path in the graph representing semistructured information and in this way they define their visit order between objects. By appropriately collecting and querying such information, we can create a view depending on each user's choices, that can be useful to personalize data presentation.

We think that temporal aspects of semistructured data are related to the main issues that the research community studying techniques to adapt applications to users is now considering. Indeed, in order to customize the presentation of information it is therefore very important to capture the *history of interactions* between users and the applications they browse.

These considerations have motivated our work that studies a novel and efficient method based on model-checking techniques for automatically extracting semistructured and time-based data, in particular w.r.t. Valid Time and Interaction Time.

1.2 Overview of the Book

Semistructured data are often represented by labeled graphs, a data model which can be made less rigid than the tabular one: note that the web itself may be seen as a large, graph-like database. In addition, a lot of recent proposals have concentrated on issues concerning techniques for introducing flexible and expressive query languages.

This book contains several contributions toward this end. In Chapter 2 we formalize a method to provide suitable graph-based semantics to languages for semistructured data, supporting both data structure variability and different topological similarities between queries and document structures.

We start by observing that the data retrieval activity requires the development of graph algorithms, since queries (graphical or not) are expected to extract information stored in labeled graphs.

In order to do that, it is required to perform a kind of *matching* of the *query graph* with the *database instance* graph. More in detail, we need to find subgraphs of the instance of the database that match (e.g. they are isomorphic or somehow similar to) the query graph.

Our approach to meet the effectiveness and efficiency requirements in querying semistructured data is to make the required topological similarity flexible, in order to support different similarity levels. Therefore, one aim of the chapter is to illustrate effective techniques that allow the query composer to relax or restrict topological requirements at their choice. One main contribution is the design of a suite of semantics for the graphical language G-Log [81], based on the notion of bisimulation [70] (see also [12, 37, 64]), given both at the instance and at the schema level. In particular, we discuss in full detail the benefits of tuning the semantics by enforcing or relaxing requirements of the bisimulation relation.

We choose G-Log because it is a general graphical language which combines the expressive power of logic, the modeling power of objects, and the representation power of graphs. It was initially conceived as a *deductive language for complex objects with identity*, since its data model captures most of the modeling capabilities of the object-oriented query languages (i.e. structural and semantic complexity) but lacks the data abstraction features, also typical of the object-oriented languages and databases, that are more related to the dynamic aspects of the system. Moreover, in G-Log the representation of logical inference is also graphical.

Our work started as a part of the WG-Log project [32], which addressed the problem of Web information querying by enriching G-Log [83, 81] with constructs typical of hypermedia, like entry points, indexes, navigational edges, etc. A subsequent project [18, 31, 42], still in the area of semistructured information querying, addressed the problem of querying XML-specified information, and still investigated the possibilities of flexible graph-based query formulation. To this aim, in [78] the XML-GL language is translated into G-Log, in order to take advantage of the parametric semantics defined here. The results presented here for G-Log can thus be easily extended to WG-Log and XML-GL as well.

As a typical situation with semistructured information is that the databases lack an a-priori schema, we used the theory of abstract interpretation to derive a common schema for a given (set of) document bases, and keep trace of changes made to the underlying instances. Thus, as schemata can evolve gracefully with the evolution of their instances in the extended setting of WG-Log and XML-GL, and more in general in the graph-based languages similar to G-Log, this will allow us to trace the evolution of documents and Web pages by keeping trace of the history of their DTD's or schemata. This possibility to easily keep trace of schemata evolution is useful to apply queries for finding out information about the structure of Web sites.

The suite of semantics defined for G-Log, all based on the bisimulation concept, allows us to support different similarity levels, but in general is not applicable to real-life size problems, because even if the problem of establishing whether two graphs are bisimilar or not is polynomial in time [60, 79], the task of finding subgraphs isomorphic or bisimilar to a given one is NP-complete [44].

At this point, our main problem consists in finding algorithms to efficiently implement the data retrieval activity for semistructured data. In order to overcome this limiting but crucial issue, we observe that graphical queries can be easily translated into logic formulae. In fact it is possible to find many attempts to give a graph-based representation of first-order logic or to translate graphical queries into logical formulae (see for example [81, 33, 10, 61]). In general, techniques for translating graphs in formulae have also been exploited in non-database related literature [14].

The novel idea of this work is to associate a *modal* logic formula Ψ to a graphical query, and to interpret database instance graphs as *Kripke Transition Systems (KTS)*. In this way we reduce the problem of solving a graphical query w.r.t. a semistructured database to an instance of the *model-checking problem.*

Model-checking [27, 73, 28] has emerged in the last decade as a promising and powerful approach to the automatic verification of systems. Intuitively, a *model-checker* is a procedure that decides whether a given structure M is a model of a logical formula φ. More in detail, M is an (abstract) model of a system, typically a finite automata-like structure, and φ is a modal or temporal logic formula describing a property of interest.

In Chapter 3, we use a modal logic with the same syntax as the temporal logic *CTL*. In this way, finding subgraphs of the database instance graph that match the query can be performed by finding nodes of the KTS derived from the database instance graph that satisfy the formula Ψ. This is an instance of the *model-checking* problem, and it is well-known that if the formula Ψ belongs to the class *CTL* of formulae, then the problem is decidable and algorithms running in linear time on both the sizes of the KTS and the formula can be employed [27].

In this work we identify a family of graph-based queries that represent *CTL* formulae and discuss the expressive limits of this propositional temporal logic used as a query language. Observe that in this first setting, the notion of different instants of *time* represents the number of links the user needs to follow to reach the information of interest.

Our translation from graphical queries to modal formulae is very natural and, as immediate consequence, an effective procedure for efficiently querying semistructured databases can be directly implemented on a model checker. We use a "toy" query language called $\mathbb{W}$ mainly based on the G-Log language: we do not consider some details of G-Log (e.g. the possibility to generate new relations in a given database by means of rules) but add the possibility to express universal conditions in a graph-based setting. We can say that $\mathbb{W}$ is a representative of several approaches in which queries are graphical or can be easily seen as graphical (cf. e.g. Lorel [4], G-Log [81], GraphLog [33], and UnQL [50]). We will relate $\mathbb{W}$ to UnQL, GraphLog and G-Log and show the applicability of the method for implementing (parts of) these languages. The model-checking technique for data retrieval could also be applied to non-graphical query languages: formula extraction from SQL-like queries can be done by using similar ideas.

We have effectively tested the approach by using the model-checker NuSMV [26] to confirm that our technique can be really applied.

It is known that model-checking has been used in the past years as a powerful approach to the automatic verification of systems, and is especially applied to verify *temporal properties* on reactive systems [49].

Leaving from the potential of the *CTL* language for time-based representation, in Chapter 4 we adapt the technique introduced for accessing static information stored in graphical databases to retrieve temporal aspects of semistructured data as well.

Most applications of database technology are temporal in nature. In fact, in many applications, information about the *history* of data and their *dynamic* aspects are just as important as static information. Consider, for example, records of various kinds: financial, personnel, medical and scheduling applications such as airline, train and hotel reservations.

These kinds of applications rely on *temporal databases*, which record time-referenced data. The main concepts of classical temporal databases can therefore be applied and adapted to semistructured databases, which are usually represented by means of labeled graphs instead of flat tables. Thus, our first step for studying temporal aspects of semistructured data is to extend graph-based data models to keep trace of historical information. In order to represent dynamic aspects of this kind of data and to allow queries about their evolution through time, we chose to add to the graph the information about the *time interval* of objects and relations. This kind of information represents the *valid time* of objects and relations, i.e. the time when the objects or relations are true in the modeled reality [57].

In particular, in this work we analyze the possible contexts where semistructured applications arise, and consider several dynamic aspects of semistructured data representation, based on different uses of the concept of “time interval”.

We first define a graphical model allowing to represent and retrieve temporal information about semistructured data according to the concepts of Valid Time and Interaction Time.

The relevance of Valid Time is widely accepted, since, as in the classical database field ([90], [25], [24]), also in the context of semistructured data it is interesting to take into account the dynamic aspects of data.

Interaction Time is a somehow novel field of discussion, as nowadays the explosive growth of information sources available on the World Wide Web, has made necessary for users to utilize automated tools to find quickly the desired information resources, and for site developers to track and analyze their usage patterns. These factors are deeply studied in a research area called Web Mining [34] that can be broadly defined as the automatic discovery and analysis of useful information from the World Wide Web.

In this work we show that a general temporal model can be adapted to keep trace of usage patterns while navigating web sites. The analysis of how users are accessing a site is critical for optimizing the structure of the Web site and can be placed among the pre-processing activities required for mining before the mining algorithms can be run [34]. Our proposals is thus to use a database approach for storing sequences of page references before any mining is done.

Results on such historical analysis can be successively used for improving and customizing site content presentation: in fact the graph-based temporal representation is used not only to keep track of user's preferences, but also to pose all sorts of different queries by means of a SQL-like query language based on path expressions, called TSS-QL [77].

This language is not graphical because in our opinion the addition of the temporal dimension makes it difficult to directly express generic temporal properties in a graph-based form (e.g. with G-Log graphs); however, a graphical version has also been designed, which is an extension of the G-Log language. This encoding allows to reduce the problem of solving a query on a graph-based model to the problem of performing a *matching* of the "query graph" with the "site graph". Again, in order to overcome complexity limits of algorithms for solving the matching problem between subgraphs, we find the fragment of our language that can be correctly translated into the logic *CTL* for applying model-checking techniques also to solve temporal TSS-QL queries.

1.3 Contributions

This book investigates static and dynamic properties of semistructured data by concentrating on a new perspective to efficiently solve queries on such kind of data. In particular, this work is a proposal to deal with the necessity to add flexibility and efficiency to methods for solving graphical queries by considering approaches in quite different research areas, going from standard database techniques to temporal databases, Web Mining [9, 23, 34, 100] and Model-Checking [49].

We differ from other works in the literature in several respects:

1. We study a search technique that overcomes the basic information retrieval paradigm supported by most search engines and consider also the

structure of documents, to use for adding flexibility to query methods. In particular, we show that by considering different notions of similarity between query graphs and instance graphs it is possible to relax or restrict the set of retrieved information. We also formalize the relationship between semistructured instances and their schemata by the application of the theory of Abstract Interpretation; this approach allows us to emphasize the ability though schemata whenever it is useful to infer information about the structure in order to query databases modeled by labeled graphs.

2. In the semistructured database context, we extend a general model based on labeled graphs in order to store changes to data. The model is general enough to be applied in different contexts; in fact we consider more than one dynamic aspect of semistructured data, and by the same model we can store information either on the natural evolution of data through time or on user activities while navigating Web sites.
3. This work is also an attempt to formalize the relationships between the data retrieval activity in the semistructured context and the model-checking problem. This well-known technique for verifying temporal properties on Kripke Transition Systems is an efficient procedure to solve (graphical) queries on semistructured databases. In particular, we show how to translate queries into *CTL* formulae and graphical instances into Kripke Transition Systems; some experimental results are reported to confirm the feasibility of this technique. We explain the main restrictions of the proposed approach and the expressive power limits of propositional temporal logics used as query languages.

To sum up, we may say that ours is a very comprehensive approach to problems related to modeling and querying semistructured data, more global than most of the ones present in the literature, and benefit from the advantages of techniques originally designed in different research areas (Databases, Logics, Web Mining and Model-Checking) to deal with the need of flexible and efficient search methods for retrieving static as well as dynamic information.

2. Semantics Based on Bisimulation

Computational models based on graph transformation are used as semantic domains for various kinds of formalisms and languages like, for example, actor systems, the π-calculus, functional programming, database query and manipulation languages, and neural networks.

Also semistructured data are often represented by using data models based on directed labeled graphs: among these we can cite, UnQL [12], LOREL [66], WebSQL [67], WebOQL [7], StruQL [50], GraphLog [33] and G-Log [81]. Effectiveness and efficiency are mandatory requirements when accessing this kind of information; therefore, appropriate search techniques are more than necessary and a lot of work has been done to face the problem of accessing in a uniform way this kind of data with graph-based queries. In some approaches queries are really graphs [50, 33, 81] while, in others, queries can be written in extended SQL languages [12, 4, 16].

In this work, we mainly refer to the graphical representation and query style of G-Log, a well-known database language for complex objects [83, 81]. The reason of the choice of G-Log stands on observing that most of the models and languages for representing and querying semistructured information share an analytical approach to data representation, lacking a synthetic notion of schema. Conversely, G-Log models semistructured information by using a concept very close to that of *database schema*, that in this context enables the user to formulate a query in an easier way. Nevertheless, the use of a schema-like facility, however desirable, should not be mandatory, since we may well imagine a situation where the user is not fully aware of the document's exact organization. In this case, assuming a strict matching between the document and the required topological structure may lead to miss some still interesting documents that do not adhere precisely to the query structure.

Our approach to attack these problems is to make the required topological similarity flexible, in order to support different similarity levels. Therefore, the aim of this chapter is to illustrate effective techniques that allow the query formulator to relax or restrict topological requirements at their choice. Its main contribution is the design of a suite of semantics for G-Log, based on the notion of bisimulation [70] (see also [64, 12]), given both at the instance and at the schema level. In particular, we discuss in full detail the benefits of tuning the semantics by enforcing or relaxing requirements on the bisimulation relation.

The relationship between instances and schemata is investigated using the theory of Abstract Interpretation [38], which provides a systematic approach to guarantee the correctness of operating on schemata with respect to the corresponding concrete computations on instances.

The revisitation of the semantics of G-Log also clarifies some subtle ambiguities in the initial semantics of G-Log queries. Since the semantics is based on the notion of bisimulation, the implementation of the language will inherit all the algorithmic properties studied in the literature [1]. In particular, Kanellakis and Smolka in [60] relate the bisimulation problem with the general (relational) coarsest partition problem, and they propose an algorithmic solution and pointed out that the partition refinement algorithms in [79, 45] can serve, and more efficiently, to the same task. Applicability of our approach is strongly based on this efficient implementation of the bisimulation tests.

Alternative approaches to the semantics of graphical languages have been introduced in the literature. For instance, the semantics of the language GraphLog is given via rewriting into DATALOG. Our choice of giving directly a graph-based semantics is not only justified by the fact that this is a typical approach for visual languages, but also, and more significantly, by the fact that the expressive power of G-Log is higher than that of DATALOG.

Our approach may remind previous works on the two languages UnQL [12] and GraphLog [33]; differences between G-Log and them are mainly related to the expressive power: for instance, G-Log allows to express cyclic information and queries, and achieves its high expressive power by allowing a fully user-controlled non-determinism. Addition of entities, other than the addition of relations allowed in GraphLog, is admitted in G-Log which, in its full form, is Turing complete [81].

This issue is further dealt with in Chapter 5, where it becomes clear that our results can be extended to these languages as well, and in general to any graphical language whose main aim is to query and transform a graph-oriented data model by using graph-based rules.

In this chapter, Section 2.1 introduces the language G-Log and Section 2.2 explains the (concrete) semantics of the language, showing the three-level semantics which introduces flexibility. In Section 2.3 some results for the

[1] Bisimulation is usually attributed to Park, who introduced the term in [82], extending a previous notion of automata simulation by Milner ([68]). Milner employs bisimulation as the core for establishing observational equivalence of the Calculus of Communicating Systems (CCS) ([69, 70]). In the Modal Logic/Model Checking areas this notion was introduced by van Benthem (cf. [94]) as an equivalence principle between Kripke Structures. In Set Theory, it was introduced as a natural principle replacing extensionality in the context of non well-founded sets [5].

semantics proposed are given in detail; here different types of rules are analyzed and the differences between the three semantics are highlighted. In Section 2.4 the notion of abstract graphs (corresponding to schemata) is introduced, and the concepts of abstract interpretation are applied. In some cases query applicability can be tested directly on schemata; this means that they represent instances correctly. Moreover, operating on schemata is useful whenever one wants to investigate on general properties on the structure of a given (semistructured) database. This section also shows how schemata can be derived by abstraction (in $n \log n$ time) from instance sets, thus allowing to deduce a common scheme or DTD from a set of documents. As schemata can evolve gracefully with the evolution of their instances (applying the theory of abstract interpretation), in the extended setting of WG-Log [32] and XML-GL [18], and more in general in the graph-based languages similar to G-Log, this will allow to trace the evolution of documents and Web pages by keeping trace of the history of their DTD's or schemata. Section 2.5 introduces a logical and model theoretic view of G-Log graphs and the relationships with the concrete semantics are analyzed. In Section 2.6 we point out the relationships between our three-levels semantics and the G-Log original semantics defined in [81]. To conclude, Section 2.9 presents UnQL and GraphLog, two query languages for semistructured databases that we will consider in this work to show the applicability of the model-checking based data retrieval approach to both SQL-like and graphical query languages.

2.1 G-Log: a Language for Semistructured Data

2.1.1 An Informal Presentation

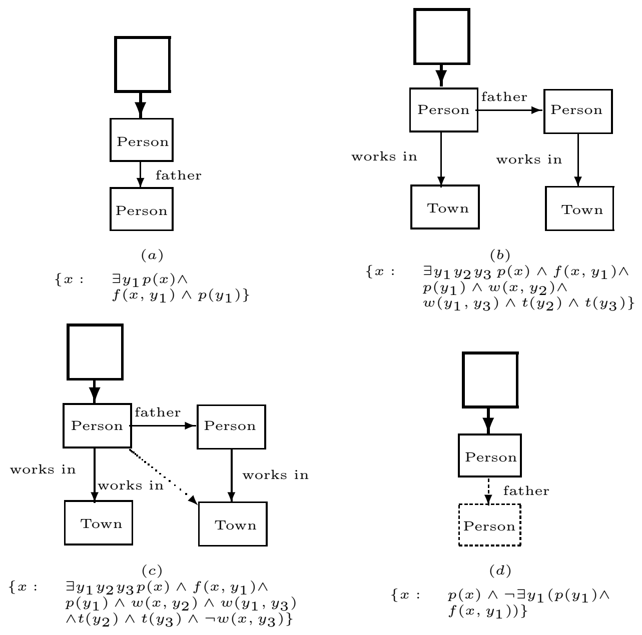
In this section we introduce some intuitive examples of queries in the language G-Log, in order to appreciate its expressive power and to emphasize some ambiguities we are going to tackle later on.

Consider the graph depicted in Fig. 2.1 (a). It represents the query 'collect all the people that are fathers of someone'. Intuitively, the boldface part of the graph (also called the 'green part') is what you try to get from the database, while you match the rest of the graph (also called the 'red part') with a graph representing the database instance.

The query (b) of Fig. 2.1 can be read as 'collect all the workers having (at least) one son that works in some town'.

Also negative requirements can be introduced in a query by means of dashed edges and nodes. This is depicted by query (c) of Fig. 2.1 whose meaning is 'collect all the workers having (at least) one son that works in a town different from that where his father works'. Moreover, the meaning of query (d) is 'collect all the people who are not father of someone'.

The translation of queries $(a), (b), (c), (d)$ into logical formulas is also illustrated in Fig. 2.1 (with abbreviations for predicate symbols). As observed



(a) $\{x : \exists y_1 p(x) \wedge f(x, y_1) \wedge p(y_1)\}$

(b) $\{x : \exists y_1 y_2 y_3\, p(x) \wedge f(x, y_1) \wedge p(y_1) \wedge w(x, y_2) \wedge w(y_1, y_3) \wedge t(y_2) \wedge t(y_3)\}$

(c) $\{x : \exists y_1 y_2 y_3 p(x) \wedge f(x, y_1) \wedge p(y_1) \wedge w(x, y_2) \wedge w(y_1, y_3) \wedge t(y_2) \wedge t(y_3) \wedge \neg w(x, y_3)\}$

(d) $\{x : p(x) \wedge \neg \exists y_1 (p(y_1) \wedge f(x, y_1))\}$

Figure 2.1. Sample queries

in [81], G-Log offers the expressive power of logic, the modeling power of object-oriented DBs, and the representation power of graphs.

However, the modeling power of G-Log is heavily constrained by some arguable choices in its semantics [81]. Consider, for instance, query (*e*) of Fig. 2.2: it can be intuitively interpreted in three different ways:

– collect the people having two children, not necessarily distinct;
– collect the people having exactly two (distinct) children;
– collect the people having at least two (distinct) children.

The semantics of G-Log as given in [81] uniquely selects the first option. As a consequence, queries (*a*) and (*e*) become equivalent, so there is no way to express ‘collect the people that have more than one child’ without making use of negative information (negated equality edges in G-Log [81]).

An even deeper problem arises when considering query (*f*): in G-Log it has exactly the same meaning as query (*b*). In other words, it is not possible to express a query like ‘collect the people that work in the same town as (at least) one of their children’ in a natural fashion. Actually, such a query can be expressed in G-Log, but not in a straightforward way. Of course, these problems are further emphasized when combined with negation.

In order to address this kind of ambiguities, in the Section 2.2 we revisit the semantics of G-Log taking advantage of the use of the well-known concept of bisimulation. Furthermore, we apply the abstract interpretation approach to the semantics defined in this way, in order to clarify the relationship between concrete (instances) and abstract (schemata) data representations.

2.1.2 Syntax of G-Log

In this section we introduce the basic aspects of the syntax of the G-Log language. Definitions are based on the concept of directed labeled graph, and, differently from [81, 83], rules, programs, and queries are defined independently of the context in which they are used. This simplifies the notation and allows the study of algebraic properties of programs. However, the semantics (cf. Section 2.2) will be given in such a way that the practical use is coherent with that of [81, 83].

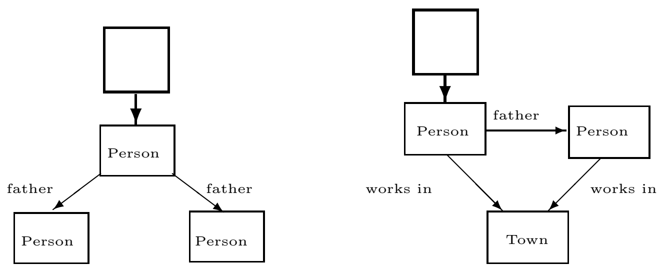
Definition 2.1.1. *A* G-Log graph *is a directed labeled graph* $\langle N, E, \ell \rangle$*, where* N *is a (finite) set of nodes,* E *is a set of labeled edges of the form* $\langle m, label, n \rangle$*, where* $m, n \in N$ *and label is a pair of* $\mathcal{C} \times (\mathcal{L} \cup \{\bot\})$*, while* $\ell : N \longrightarrow (\mathcal{T} \cup \{\bot\}) \times \mathcal{C} \times (\mathcal{L} \cup \{\bot\}) \times (\mathcal{S} \cup \{\bot\})$*.* $\bot$ *means 'undefined', and:*

– $\mathcal{T} = \{$ *entity, slot* $\}$ *is a set of* types *for nodes;*
– $\mathcal{C} = \{$ *red, green, black* $\}$ *is a set of* colors*;*
– $\mathcal{L}$ *is a set of* labels *to be used as entity, slot, and relation names;*
– $\mathcal{S}$ *is a set of* strings *to be used as concrete values.*

ℓ *is the composition of four single-valued functions* $\ell_{\mathcal{T}}, \ell_{\mathcal{C}}, \ell_{\mathcal{L}}, \ell_{\mathcal{S}}$*. When the context is clear, if* $e = \langle m, \langle c, k \rangle, n \rangle$*, with abuse of notation we say that* $\ell_{\mathcal{C}}(e) = c$ *and* $\ell_{\mathcal{L}}(e) = k$*. Moreover, we require that:*

– $(\forall x \in N)(\ell_{\mathcal{T}}(x) \neq slot \rightarrow \ell_{\mathcal{S}}(x) = \bot)$ *(i.e., values are associated to* slot *nodes only),*
– $(\forall \langle m, label, n \rangle \in E)(\ell_{\mathcal{T}}(m) \neq slot)$ *(i.e., slot nodes are leaves).*

Observe that two nodes may be connected by more than one edge, provided that edge labels be different.



(e) $\{x : \exists y_1 y_2 p(x) \wedge f(x, y_1) \wedge p(y_1) \wedge f(x, y_2) \wedge p(y_2)\}$ (f) $\{x : \exists y_1 y_2 p(x) \wedge f(x, y_1) \wedge p(y_1) \wedge w(x, y_2) \wedge w(y_1, y_2) \wedge t(y_2)\}$

Figure 2.2. Problematical queries

Red (RS) and *black* edges and nodes are graphically represented by thin lines (this does not originate confusion, since there can not be red and black nodes and edges in the same graph), while *green* (GS) by thick lines. Red and green nodes are used together in queries.

Colors red and green are chosen to remind traffic lights. Red edges and nodes add constraints to a query (= stop!), while green nodes and edges correspond to the data we wish to derive (= walk!).

Result nodes play a particular rôle in queries: they have the intuitive meaning of requiring the collection of all objects fulfilling a particular property. Moreover, result nodes can occur in (instances of) web-like databases to simulate web pages connecting links. In this paper, if an entity node is labeled by *result*[2] it will be simply represented by small squares, and its outcoming edges implicitly labeled by *'connects'*. In general, an *entity* (*slot*) node n will be represented by a rectangle (oval) containing the label $\ell_{\mathcal{L}}(n)$.

As an instance, consider the graph (e) of Fig. 2.2. Let 1 be the topmost node, 2 the center node, 3 the leftmost, and 4 the rightmost node. Then

$$\begin{array}{rcllcll}
G & = & \langle & N & = & \{ & 1,2,3,4\}, \\
 & & & E & = & \{ & \langle 1, \langle GS, \mathit{connects}\rangle, 2\rangle, \\
 & & & & & & \langle 2, \langle RS, \mathit{father}\rangle, 3\rangle, \\
 & & & & & & \langle 2, \langle GS, \mathit{father}\rangle, 4\rangle\}\ , \\
 & & & \ell & = & \{ & 1 \mapsto \langle \mathit{entity}, GS, \mathit{result}, \bot\rangle, \\
 & & & & & & 2 \mapsto \langle \mathit{entity}, RS, \mathit{Person}, \bot\rangle, \\
 & & & & & & 3 \mapsto \langle \mathit{entity}, RS, \mathit{Person}, \bot\rangle, \\
 & & & & & & 4 \mapsto \langle \mathit{entity}, RS, \mathit{Person}, \bot\rangle\} \quad \rangle
\end{array}$$

Definition 2.1.2. *Let $G = \langle N, E, \ell\rangle$ and $G' = \langle N', E', \ell'\rangle$ be G-Log graphs. We say that G is a* labeled subgraph *of G', denoted $G \sqsubseteq G'$, if $N \subseteq N'$, $E \subseteq E'$, and $\ell = \ell'|_N$ (i.e., for all $x \in N$ it holds that $\ell(x) = \ell'(x)$).*

With ε we denote the (empty) G-Log graph $\langle \emptyset, \emptyset, \emptyset\rangle$. It is immediate to see that given a G-Log graph G, then

$$\langle \{G' \text{ is a G-Log graph } : G' \sqsubseteq G\}, \sqsubseteq\rangle$$

is a *complete lattice*, where $\top \equiv G$, $\bot \equiv \varepsilon$. Moreover, given two G-Log graphs $G_1 = \langle N_1, E_1, \ell_1\rangle \sqsubseteq G$ and $G_2 = \langle N_2, E_2, \ell_2\rangle \sqsubseteq G$, where $\ell_1|_{N_2} = \ell_2|_{N_1}$, their l.u.b. and g.l.b. can be computed as[3]

$$\begin{array}{rcl}
G_1 \sqcup G_2 & = & \langle N_1 \cup N_2, E_1 \cup E_2, \ell_1 \cup \ell_2\rangle \\
G_1 \sqcap G_2 & = & \langle N_1 \cap N_2, E_1 \cap E_2, \ell_1 \cap \ell_2\rangle
\end{array}$$

[2] [41] uses *entry point* nodes for this purpose.

[3] As a side remark, notice that, if G is the (complete) graph $\langle N, N \times \{\bot\} \times N, \ell\rangle$ and $n = |N|$, then the lattice contains: $\sum_{i=0}^{n} \binom{n}{i} 2^{i^2} = O(n 2^{n^2})$ subgraphs. If G is not of this form, it is difficult to find the exact number; however, if $|E| = O(|N|^2)$, then the upper bound remains the same as the complete case.

We recall that a *complete lattice* is a non-empty ordered set P such that the the least upper bound and the greatest lower bound exist for each subset S of P.

Definition 2.1.3. *Given a G-Log graph $G = \langle N, E, \ell\rangle$, and a set C of colors, $C \subseteq \mathcal{C}$, consider the sets*

$$N' = N \cap \ell_{\mathcal{C}}^{-1}(C) \text{ and } E' = \{\langle m, \langle c, k\rangle, n\rangle \in E : c \in C\}.$$

The restriction of G to the colors in C, denoted $G|_C$ is defined as $G|_C = \langle N', E', \ell|_{N'}\rangle$.

Observe that $G|_C$ is not necessarily a graph, since, for instance, it may contain only edges and no nodes.

Now we introduce the notions of concrete graph, rule, and program. Through them, we aim at characterizing the instances of a semi-structured database like a WWW site.

Definition 2.1.4. *A* G-Log concrete graph *is a G-Log graph such that:*

1. $(\forall x \in N \cup E)(\ell_{\mathcal{C}}(x) = \mathit{black})$, *and*
2. $(\forall n \in N)(\ell_{\mathcal{T}}(n) = \mathit{slot} \rightarrow \ell_{\mathcal{S}}(n) \neq \bot)$.

With $\mathcal{G}^c$ we denote the set of G-Log concrete graphs.

Definition 2.1.5. *A* G-Log rule $R = \langle N, E, \ell\rangle$ *is a G-Log graph such that:*

1. $(\forall x \in N \cup E)(\ell_{\mathcal{C}}(x) \neq \mathit{black})$,
2. $R|_{\{GS\}} \neq \emptyset$,
3. $(\forall e = \langle m, \mathit{label}, n\rangle \in E)(\ell_{\mathcal{C}}(e) = RS \rightarrow \ell_{\mathcal{C}}(n) = RS)$, *and*
4. $(\forall n, n' \in N)((\ell_{\mathcal{C}}(n) = GS \wedge \ell_{\mathcal{C}}(n') = RS) \rightarrow (\ell_{\mathcal{L}}(n) \neq \ell_{\mathcal{L}}(n') \vee \ell_{\mathcal{T}}(n) \neq \ell_{\mathcal{T}}(n')))$.

The third condition is necessary to avoid the possibility of dangling red solid edges, while the fourth condition is introduced to avoid the possibility of infinite generation of nodes (cf. Remark 2.3.1).[4] Notice that it can be the case that $R|_{\{RS\}} = \emptyset$. This corresponds to an *unconditional* query or update. In general, we can split the notion of rule in two concepts: *query* and *update*. Basically, queries are expected to extract information from a graph (i.e., no existing *class* of objects is modified), whereas updates are expected to build up new instances of the graph (i.e., classes and relationships can be modified).

Remark 2.1.1. Observe that no restriction is imposed on the structure of the graphs. Graphs can contain cycles of any kind.

[4] This corresponds to the well-known problem of OID generation in logical object oriented DB's [1].

Definition 2.1.6. *Given a G-Log rule $R = \langle N, E, \ell \rangle$ and a graph $G = \langle N', E', \ell' \rangle$, The rule R is a* query with respect to G *if the following conditions hold:*

1. $(\forall n \in N)(\ell_{\mathcal{C}}(n) = GS \rightarrow (\forall n' \in N')(\ell_{\mathcal{L}}(n) \neq \ell_{\mathcal{L}}(n') \vee \ell_{\mathcal{T}}(n) \neq \ell_{\mathcal{T}}(n')))$,
2. $(\forall e \in E)(\ell_{\mathcal{C}}(e) = GS \rightarrow (\forall e' \in E')(\ell_{\mathcal{L}}(e) \neq \ell_{\mathcal{L}}(e')))$.

The rule R is an update *with respect to G if it is not a query.*

As a matter of fact, this formal notion does not correspond exactly to the common usage of the word "query": we further distinguish two kinds of queries: *generative* queries retrieve some objects and relationships and, based on them, construct new concepts. For instance, from the notion of parent, a generative query (the transitive closure) can construct the notion of ancestor. *Pure retrieval* queries associate links to a number of objects enjoining a common property. The last notion captures the common intuition of *query.* The previous one is more related to the usual notion of *view.*

A computation can be seen as a sequence of applications of a sequence of rules to a graph. This leads to the following definition of program.

Definition 2.1.7. *A* G-Log program *is a finite list of sets of G-Log rules.*

G-Log allows the expression of complex queries by means of *programs.* Sometimes it is worthwhile to have the possibility of expressing *transitive properties*: in G-Log a set of two rules is enough.

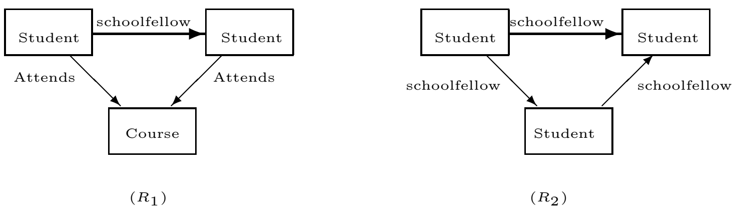


Figure 2.3. A G–Log program

For instance, the program of Fig. 2.3 expresses the following transitive property: 'if two students attend the same course, they are schoolfellows. And, if a student x is a schoolfellow of a student y and y is a schoolfellow of a student z then x is z's schoolfellow'.

Now we revisit the semantics of G-Log by using the bisimulation relation to define the concept of similarity between the query graph and the database instance graph.

2.2 Bisimulation Semantics of G-Log

We present a three-level semantics, based on the concept of bisimulation. In Section 2.6 we compare this approach with that based on the concept of *embedding* of a graph used in [81].[5]

First, let us remind the well-known concept of bisimulation [82, 70] adapted to our setting:

Definition 2.2.1. *Given two G-Log graphs $G_0 = \langle N_0, E_0, \ell^0 \rangle$ and $G_1 = \langle N_1, E_1, \ell^1 \rangle$, a relation $b \subseteq N_0 \times N_1$ is said to be a* bisimulation *between G_0 and G_1 if and only if:*

1. *for $i = 0, 1$, $(\forall n_i \in N_i)(\exists n_{1-i} \in N_{1-i}) n_0\ b\ n_1$,*
2. *$(\forall n_0 \in N_0)(\forall n_1 \in N_1)(n_0\ b\ n_1 \rightarrow \ell^0_{\mathcal{T}}(n_0) = \ell^1_{\mathcal{T}}(n_1) \wedge \ell^0_{\mathcal{L}}(n_0) = \ell^1_{\mathcal{L}}(n_1) \wedge \ell^0_{\mathcal{S}}(n_0) \doteq \ell^1_{\mathcal{S}}(n_1))$ (where $\doteq$ means that if both labels are defined—i.e., different from $\perp$—they must be equal), and*
3. *for $i = 0, 1$, $(\forall n \in N_i)$, let*

$$M_i(n) =_{def} \{\langle m, label \rangle : \langle n, \langle color, label \rangle, m \rangle \in E_i\}.$$

Then, $(\forall n_0 \in N_0)(\forall n_1 \in N_1)$ such that $n_0\ b\ n_1$, for $i = 0, 1$ it holds that

$$\begin{array}{l}(\forall \langle m_i, \ell_i \rangle \in M_i(n_i))(\exists \langle m_{1-i}, \ell_{1-i} \rangle \in M_{1-i}(n_{1-i})) \\ (m_0\ b\ m_1 \wedge \ell_i = \ell_{1-i}).\end{array}$$

We write $G_0 \overset{b}{\sim} G_1$ ($G_0 \overset{b}{\not\sim} G_1$) if b is (not) a bisimulation between G_0 and G_1. We write $G_0 \sim G_1$ ($G_0 \not\sim G_1$) if there is (not) a bisimulation between G_0 and G_1: in this case we also say that G_0 is bisimilar to G_1.

A brief explanation of the conditions above may be useful. Condition 1 is obvious: no node in the two graphs can be left out of the relation b. Condition 2 states that two nodes belonging to relation b have same type and same label, exactly. Moreover, if they are just slots, then either one of their values is undefined, or they have also the same value. Finally, condition 3 deals with edge correspondences. If two nodes n_0, n_1 are in relation b, then every edge having n_0 as endpoint should find as a counterpart a corresponding edge with n_1 as endpoint.

As an example, consider the graphs in Fig. 2.4:

- $G_0 \sim G_1 \sim G_2$, as well as the reflexive, symmetric and transitive closure of this fact, since $\sim$ is an equivalence relation (cf. Lemma 2.2.1);
- $G_0 \not\sim G_3$ since it is impossible to 'bind' a node labeled by P with one labeled by Q. Thus, condition (2) cannot be verified;
- $G_0 \not\sim G_4$ since it is impossible to verify condition (3).

[5] In this chapter we do not face the problem of negation (dashed nodes and edges). A line for future work is drawn in Section 2.7.

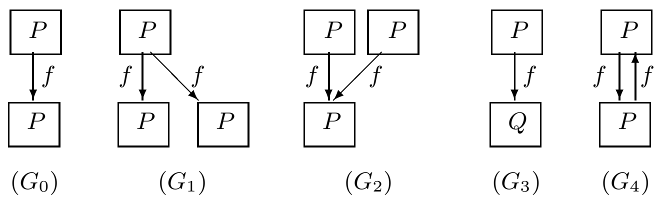


Figure 2.4. Bisimilar and not bisimilar graphs

Notice that colors (represented by the $\ell_\mathcal{C}$ function) are not taken into account in the bisimulation definition, while the *value* fields of the label are considered only when they are defined. The last feature will allow to apply bisimulations between *schemata* and *instances* (see Section 2.4).

The bisimulation relation can be further refined by introducing additional conditions:

Definition 2.2.2. *Given two G-Log graphs* $G_0 = \langle N_0, E_0, \ell_0 \rangle$ *and* $G_1 = \langle N_1, E_1, \ell_1 \rangle$*, and* $b \subseteq N_0 \times N_1$ *we say that*

- *b is a* directional bisimulation*, denoted by* $G_0 \overset{b}{\thicksim} G_1$*, if* $G_0 \overset{b}{\sim} G_1$ *and b is a function from* N_0 *to* N_1*. We say that* $G_0 \thicksim G_1$ *if there is a b such that* $G_0 \overset{b}{\thicksim} G_1$.
- *b is a* bidirectional bisimulation*, denoted by* $G_0 \overset{b}{\equiv} G_1$*, if* $G_0 \overset{b}{\sim} G_1$ *and b is a bijective function from* N_0 *to* N_1*. We say that* $G_0 \equiv G_1$ *if there is a b such that* $G_0 \overset{b}{\equiv} G_1$.

Again in Fig. 2.4, we have that:

- $G_1 \thicksim G_0$, $G_2 \thicksim G_0$, while the converse is not true.
- $G_i \equiv G_i$ for $i = 0, 1, 2, 3, 4$, while $G_i \not\equiv G_j$ for $i \neq j$.

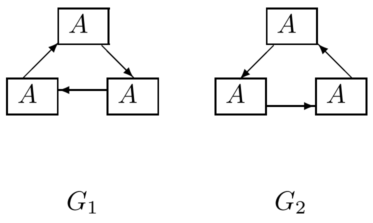
The three relations defined above will be used to define the semantics that we are going to study in the rest of the chapter:

$\sim$ is used to build a semantics based on bisimulation;
$\thicksim$ is used to build a semantics based on the concept of bisimulation that is also a function;
$\equiv$ is used to build a semantics based on *graph isomorphism* (*injective embeddings* [81] or, equivalently, *bisimulations that are bijections*).

Some basic properties of these relations are emphasized in the following lemma.

Lemma 2.2.1. *1.* $\sim$, $\thicksim$, *and* $\equiv$, *are reflexive and transitive relations;*
2. Both $\sim$ *and* $\equiv$ *are symmetric relations and thus, equivalence relations;*
3. $\thicksim$ *is a preorder (and not an ordering).*
4. $G \equiv G'$ *if and only if* $G \thicksim G'$ *and* $G' \thicksim G$.

Proof. Follows immediately from the definition for 1 and 2. For 3, consider the graphs:



It holds that $G_1 \boldsymbol{\sim} G_2$ and vice versa. However, the two graphs are not the same graph. For proving statement (4), it is sufficient to observe that, by the requirement (1) of the definition of bisimulation (Definition 2.2.1), the two functions ensuring that $G \boldsymbol{\sim} G'$ and $G' \boldsymbol{\sim} G$ are onto. This means that $|N| = |N'|$ and, thus, both functions are bijections. □

Remark 2.2.1. The semantics of the original definition of the language G-Log [81] is based on a different notion of matching of graphs: the so-called *embedding*. Relationships between our proposals and the embedding are explained in Section 2.6.

2.2.1 Semantics of Rules

The first two notions that we define are the applicability of a rule and the satisfiability of a graph, given a rule. Note that in this chapter we deal with the fragment of G-Log without negation.

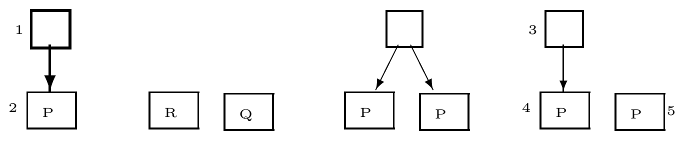
Definition 2.2.3. *Let G be a concrete graph and R a rule. For ξ in $\{\sim, \boldsymbol{\sim}, \equiv\}$, R* is ξ-applicable in *G if $(\exists G_1 \sqsubseteq G)(R_{\{RS\}} \xi G_1)$.*

Definition 2.2.4. *Let G be a concrete graph and R a rule. For ξ in $\{\sim, \boldsymbol{\sim}, \equiv\}$, G* ξ-satisfies *R ($G \models_\xi R$) if for all $G_1 \sqsubseteq G$ such that there is b_1 with $R_{\{RS\}} \overset{b_1}{\xi} G_1$, there exist $G_2 \sqsubseteq G$ and $b_2 \supseteq b_1$ that satisfy the following conditions:*

(i) $G_1 \sqsubseteq G_2$;
(ii) $R_{\{RS,GS\}} \overset{b_2}{\xi} G_2$;
(iii) $(\forall G_3 \sqsubseteq G_2)(G_1 \sqsubseteq G_3 \wedge R_{\{RS\}} \xi G_3 \rightarrow G_3 = G_1)$.

Intuitively, G satisfies R if for any subgraph G_1 of G matching (with respect to ξ) the red part of the rule (i.e. the pre-condition), there is a way to 'complete' G_1 into a graph $G_2 \sqsubseteq G$ such that the whole rule R matches G_2. Condition *(iii)* is necessary to avoid the possibility of using other parts of G, matching with $R_{\{RS\}}$ independently, to extend G_1.

Example 2.2.1. For instance, consider the graphs below.



(*R*) (G_1) (G_2) (G_3)

Rule R is not ξ-applicable to G_1 (so G_1 ξ-satisfies R trivially) and G_2 ξ-satisfies R, for ξ in $\{\sim, \thicksim, \equiv\}$. Observe the necessity of condition (*iii*), in the case of $\sim$, to ensure that G_3 does not $\sim$-satisfy R. As a matter of fact, with $b_1 = \{\langle 2,5\rangle\}$ $R_{RS} \overset{b_1}{\sim} G'$ where G' consists of the unique node 5. However, with $b_2 = \{\langle 2,4\rangle, \langle 2,5\rangle, \langle 1,3\rangle\}$ it holds that $R \overset{b_2}{\sim} G_3$. To achieve this bisimulation, however, b_1 has been 'unnaturally extended'. This is avoided by the last condition of Definition 2.2.4.

Problems like those seen in Example 2.2.1 come from the fact that functions can be extended to relations. This is not possible for $\thicksim$, and $\equiv$ because their extensions must be functions, by definition. Thus, Definition 2.2.4 can be significantly simplified for $\thicksim$ and $\equiv$:

Lemma 2.2.2. *Let G be a concrete graph and R a rule. For ξ in $\{\thicksim, \equiv\}$, G ξ-*satisfies *R ($G \models_\xi R$) if and only if for all $G_1 \sqsubseteq G$ and for all b_1 such that $R_{\{RS\}} \overset{b_1}{\xi} G_1$ $\exists G_2 \sqsubseteq G$ and $\exists b_2 \supseteq b_1$:*

1. *$G_1 \sqsubseteq G_2$ and*
2. *$R_{\{RS,GS\}} \overset{b_2}{\xi} G_2$.*

□

The notion of applicability is a pre-condition for an effective application of a rule to a concrete graph, whose precise semantics is given below:

Definition 2.2.5. *Let R be a rule. Its* semantics *$[\![R]\!]_\xi \subseteq \mathcal{G}^c \times \mathcal{G}^c$ is defined as follows: for ξ in $\{\sim, \thicksim, \equiv\}$, $\langle G, G'\rangle \in [\![R]\!]_\xi$ if and only if :*

1. *$G \sqsubseteq G'$, $G' \models_\xi R$, and*
2. *G' is minimal with respect to property (1), namely there is no graph G'' such that $G \sqsubseteq G''$, $G'' \sqsubset G'$, and $G'' \models_\xi R$.*

Intuitively, a rule, if applicable, extends G in such a way that G satisfies R. Moreover, it is required that the extension is minimal. If R is not applicable in G, then G satisfies R trivially and $\langle G, G\rangle \in [\![R]\!]_\xi$.

Example 2.2.2. Consider the graph G and the rules R_1 and R_2 of Fig. 2.5. The application of Rule R_2 leaves the graph G unchanged. The application of Rule R_1 uniquely adds the grandfather relation. The application of Rule R_2 after that of Rule R_1 further adds the grandchild relation. Thus, rule application is not commutative.

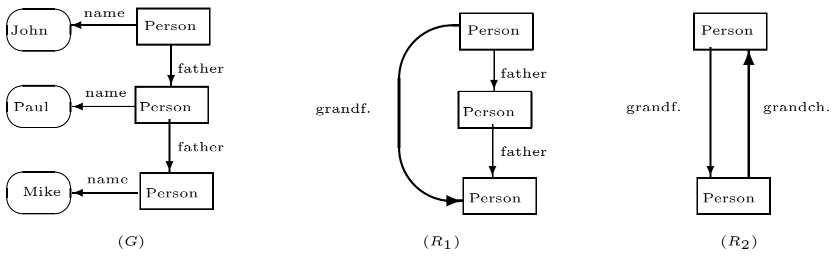


Figure 2.5. Non commutativity of rule applications

Example 2.2.3. Consider graphs of Fig. 2.6. It holds that $\langle G, G_1\rangle$ and $\langle G, G_2\rangle$ belong to $[\![R]\!]_{\sim}$. $\langle G, G_3\rangle \notin [\![R]\!]_{\sim}$ since $G_3 \not\models_{\sim} R$: this is due to condition (iii) in the Definition 2.2.4. Notice that $G_1 \sim G_2 \sim G_3$.

Definition 2.2.6. *Given a G-Log graph G, then for ξ in $\{\sim, \backsim, \equiv\}$, $[\![\cdot]\!]_{\xi}(G)$ is a function from the set of the rules to the powerset of the set of G-Log graphs, defined as follows:*

$$[\![R]\!]_{\xi}(G) =_{def} \{G' : \langle G, G'\rangle \in [\![R]\!]_{\xi}\}$$

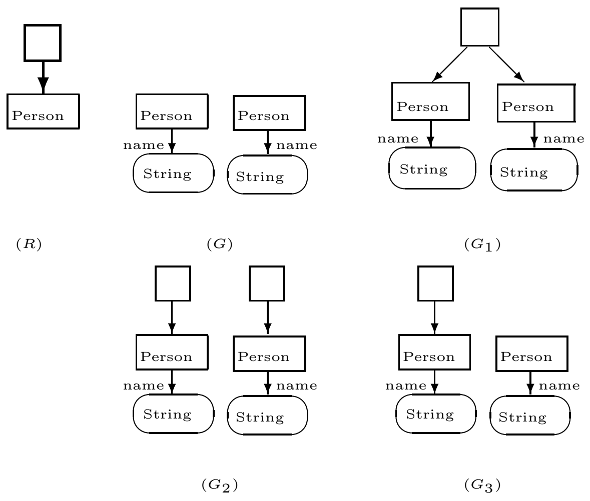


Figure 2.6. Application of a rule R to a graph G

Rules can be combined to build programs according to Definition 2.1.7:

Definition 2.2.7. *Let $S = \{R_1, \ldots, R_n\}$ be a set of rules. Then, for ξ in $\{\sim, \backsim, \equiv\}$, $\langle G, G'\rangle \in [\![S]\!]_{\xi}$ if*

1. *$G \sqsubseteq G'$, $G' \models_\xi R_i$, for $i = 1, \ldots, n$, and*
2. *G' is minimal with respect to property (1).*

Let P be a program $\langle S_1, \ldots, S_n \rangle$. For ξ in $\{\sim, \backsim, \equiv\}$, $\langle G_0, G_n \rangle \in [\![P]\!]_\xi$ if and only if there are $G_1, \ldots, G_{n-1}$ such that $\langle G_i, G_{i+1} \rangle \in [\![S_{i+1}]\!]_\xi$, for $i = 0, \ldots, n-1$.

The following notion is useful in practical querying:

Definition 2.2.8. *Let R be a rule and G be a concrete graph such that $G \models R$. For ξ in $\{\sim, \backsim, \equiv\}$, the ξ-*view *of G using R, denoted by $G|_R$ is the union of all the graphs $G' \sqsubseteq G$ such that $R \ \xi \ G'$. The* unfolded ξ-view *of G using R ($G^\uplus|_R$) is the disjoint union of all the G'.*

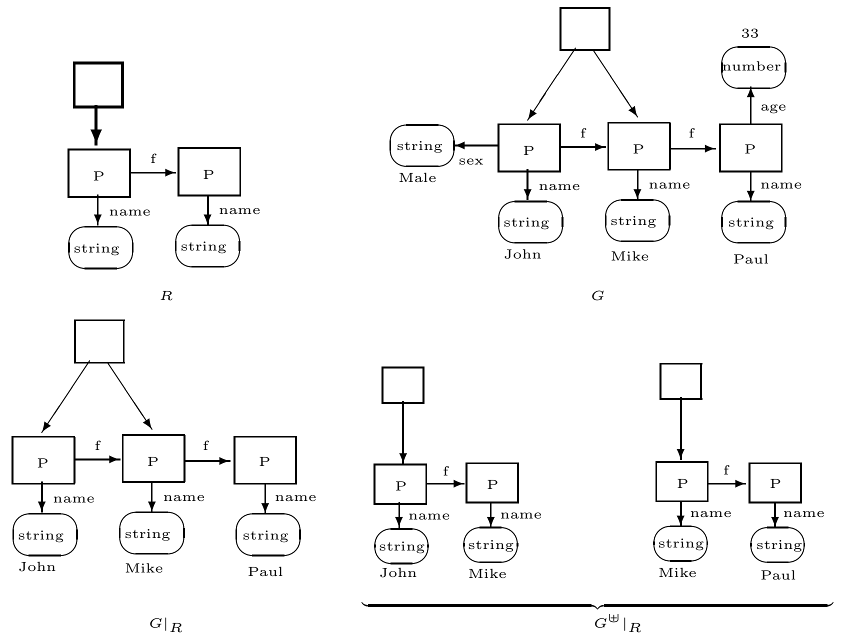


Figure 2.7. Rules, concrete graphs, and views

Fig. 2.7 shows a concrete graph G satisfying a rule R, its view using R (in this case there is no difference adopting different semantics), and its unfolded view.

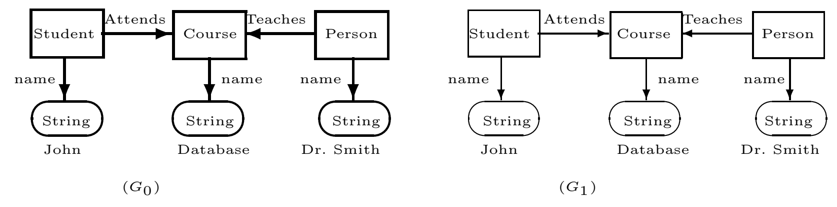


Figure 2.8. An update and a concrete graph

2.2.2 Programming in G-Log

Let us show now how to build up a database using the G-Log language and then, how to query it.

Suppose we want to create a new database which contains informations about *students*, the *courses* they attend and *teachers*. We use an unconditional rule, i.e. a rule without red part, since we wish to build up the database and not to modify an existing one.

For example, the graph G_0 depicted in Fig. 2.8 is a *G–Log generative unconditional query* (its color is only green solid); it creates a simple database with three entities: John is a student who attends the Database course and Dr. Smith is the teacher of the same course. If we apply G_0 to the initial empty concrete graph, we build up the G-Log concrete graph G_1 of Fig. 2.8 which *ξ-satisfies* G_0. Given G_1 we can either querying it or add more information to it.

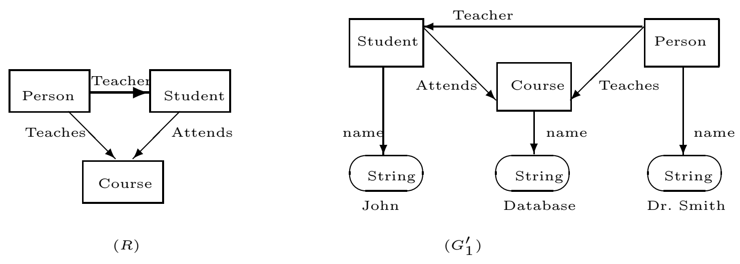


Figure 2.9. A G-Log rule and a concrete graph

Now, suppose we apply the rule R of Fig. 2.9 *'if a person teaches a course and a student attends that course then the person is a student's teacher'* to G_1. R is *ξ-applicable* to G_1 for ξ in $\{\sim, \thicksim, \equiv\}$. As a matter of fact, for each ξ there is a *ξ-relation* between the red solid part of R and a subgraph of G_1; therefore, for each ξ, there is a way to expand the concrete graph in order to obtain a new graph G_1' of Fig. 2.9 that matches the whole rule. In this particular case, the extension is the same.

This way, using G-Log rules, we can query a database to obtain information and complete its concrete graph adding new nodes or edges.

Moreover, G-Log allows the expression of complex queries by means of *programs* which are sequences of rules. Sometimes it is worthwhile to have the possibility of expressing *transitive properties*: in G-Log a set of two rules is enough.

2.3 Basic Semantic Results

In this section we analyze the main results concerning the proposed parametric semantics, in order to point out G-Log rules of a form ensuring desirable properties, first of all program determinism.

2.3.1 Applicability

Proposition 2.3.1. *For each G-Log rule R,*

1. *if R is $\equiv$-applicable, then it is $\boldsymbol{\sim}$-applicable;*
2. *if R is $\boldsymbol{\sim}$-applicable, then it is $\sim$-applicable.*

Proof. Immediate, by definition. □

Relations $\sim$, $\boldsymbol{\sim}$, and $\equiv$ have different expressivity and thus they can be compared to form an ordering (cf. also, Section 2.6). The ordering is strict, as follows from Fig. 2.10: R_1 is ξ-applicable to G_1 only when ξ is $\sim$. R_2 is ξ-applicable to G_2 for $\sim$ and $\boldsymbol{\sim}$, but not for $\equiv$.

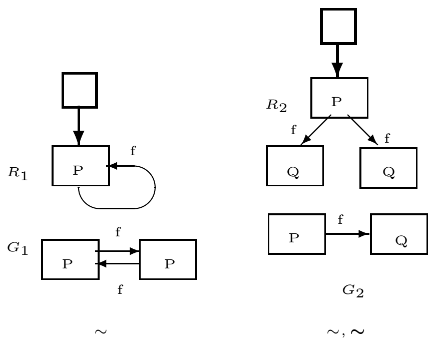


Figure 2.10. Applicability differences

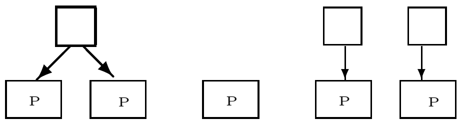
2.3.2 Satisfiability

The situation is a bit more intricate as far as the concept of *satisfiability* is concerned:

Proposition 2.3.2. *There are rules R such that the sets $\{G : G \models_{\sim} R\}$, $\{G : G \models_{\thicksim} R\}$, and $\{G : G \models_{\equiv} R\}$ are pairwise distinct.*

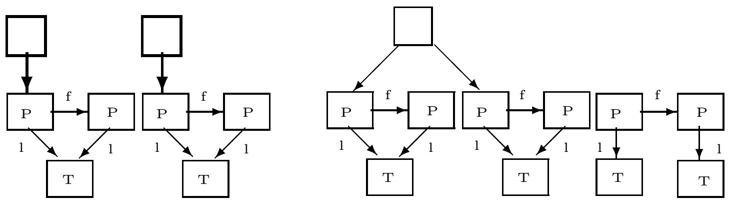
Proof. Consider rule R_1 and the graph G_1 below. G_1 does not satisfy R for $\sim$ and $\thicksim$. However, since R_1 is not $\equiv$-applicable in G_1, trivially $G_1 \models_{\equiv} R_1$.

On the other hand, we can note that rule R_1 is $\equiv$-applicable (hence, $\sim$- and $\thicksim$-applicable, thanks to Proposition 2.3.1) to graph G_2. However, it only holds that $G_2 \models_{\thicksim} R_1$.



(R_1) (G_1) (G_2)

Now we prove that for some R, $\{G : G \models_{\thicksim} R\}$ may contain elements that are not in the other two sets. Consider rule R_2 and graph G_3 below:



(R_2) (G_3)

R_2 is $\equiv$-applicable to G_3 but it is not $\equiv$-satisfied. Similarly, R_2 is $\sim$-applicable to G_3 but it is not $\sim$-satisfied, due to the rightmost subgraph. Instead, G_3 $\thicksim$-satisfies R_2. □

Thus, none of the three sets is included in the other.

Proposition 2.3.3. *Let G be a G-Log graph and R a G-Log rule. For ξ in $\{\sim, \thicksim, \equiv\}$, if R is ξ-applicable to G, then there is a G' such that 1) $G \sqsubseteq G'$, and 2) $G' \models_{\xi} R$, and G' is minimal with respect to the properties (1) and (2).*

Proof. Consider a rule R ξ-applicable to G. The existence of a G' fulfilling (1) and (2) is clearly ensured: for each $G_i \sqsubseteq G$ such that $G_i \xi R_{RS}$ (existing by hypothesis), consider G'_i obtained by augmenting G_i with new nodes and edges 'copying' R_{GS}. Consider $G' = G \cup \bigcup_i G'_i$. The assumption (3) of the definition of rule (Definition 2.1.5) ensures that the process cannot enter into loop, thus ensuring the finiteness of the graph G'.

To get one (among the various possible) minimal graph it is sufficient to remove some of the new edges and nodes (possibly collapsing them) while the satisfiability property still holds. □

In other words, if R is ξ-applicable to G, then $[\![R]\!]_{\xi}(G)$ is not empty.

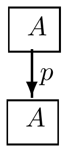
Corollary 2.3.1. *For $\xi \in \{\sim, \boldsymbol{\sim}, \equiv\}$, for any rule R and graph G, it holds that $[\![\, R \,]\!]_\xi(G) \neq \emptyset$.*

Proof. If R is not $\equiv$-applicable in G, then $[\![\, R \,]\!]_\xi(G) = \langle G, G \rangle$ by definition. Otherwise, the result follows from Proposition 2.3.3. □

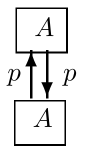
2.3.3 Simple Edge-Adding Rules

We analyze now the effect of some simple rules, *edge-adding rules*, and prove the determinism of their semantics. First of all we point out an ambiguity hidden in the graphical language.

Consider the following rule in which the green part is composed only by one edge connecting two nodes with the same label:



Its intuitive meaning is: any time you have two entity nodes labeled by A (necessarily distinct if we are using the $\equiv$ semantics) add an edge labeled p between them, unless one edge of this form is already present. The meaning is exactly the same as that of the following rule:



The first rule, in a sense, hides a cycle of green edges. Notice that this happens even if the two rules are not bisimilar: the existence of a bisimulation between two rules is not required for the two rules to have the same expressive power.

Table 2.1 shows the differences of the semantics of rules R_1, R_2, R_3 admitting cycles of green edges involving equivalent nodes on simple concrete graphs G_1, G_2, G_3.

We observe that:

1. The three semantics are all equivalent with respect to rule R_1;
2. For the other rules, there are always differences;
3. Rules R_2 and R_3 may be non-$\equiv$-applicable for graphs with too few nodes. This is due to the constraint on cardinality required by the graph isomorphism relation $\equiv$;

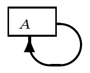
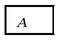
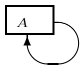
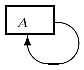
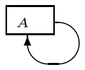
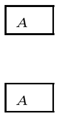
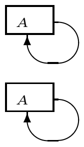
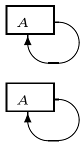
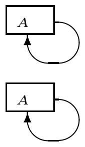
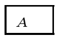
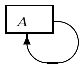
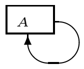
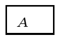
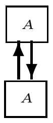
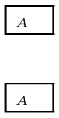
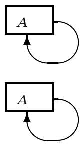
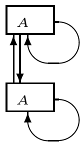
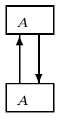
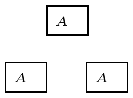
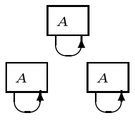
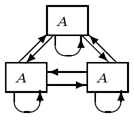
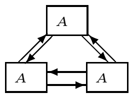
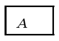
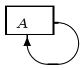
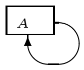
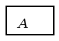
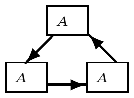
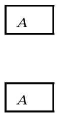
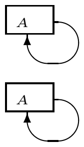
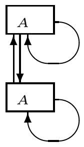
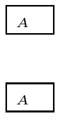
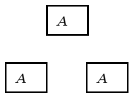
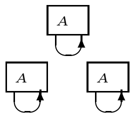
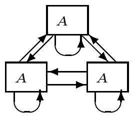
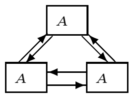
Rule	Graph	Semantics		
		$\sim$	$\boldsymbol{\sim}$	$\equiv$
 (R_1)	 (G_1)	 $\sim$	 $\boldsymbol{\sim}$	 $\equiv$
	 (G_2)	 $\sim$	 $\boldsymbol{\sim}$	 $\equiv$
	 (G_1)	 $\sim$	 $\boldsymbol{\sim}$	 $\equiv$
 (R_2)	 (G_2)	 $\sim$	 $\boldsymbol{\sim}$	 $\equiv$
	 (G_3)	 $\sim$	 $\boldsymbol{\sim}$	 $\equiv$
	 (G_1)	 $\sim$	 $\boldsymbol{\sim}$	 $\equiv$
 (R_3)	 (G_2)	 $\sim$	 $\boldsymbol{\sim}$	 $\equiv$
	 (G_3)	 $\sim$	 $\boldsymbol{\sim}$	 $\equiv$

Table 2.1. Effects of 'cyclic' rules

4. The semantics based on bisimulation ($\sim$) does not distinguish the three rules R_1, R_2, and R_3. This is due to the possibility given by bisimulation (a relation in general—not necessarily a function) to bind one node with a family of nodes.
5. The semantics based on $\boldsymbol{\sim}$ can not distinguish rules R_2 and R_3;
6. The semantics based on $\equiv$ distinguishes all the rules.
7. In all the examples, the application of rules is a function.

Actually, the same conclusions can be drawn whenever all the nodes belonging to a cycle are roots of ξ-equivalent and disjoint G-Log graphs. R_1 and R_2 are:

– $\sim$-equivalent if $R_1 \sim R_2$,
– $\equiv$-equivalent if $R_1 \equiv R_2$, and
– $\boldsymbol{\sim}$-equivalent if for all I, $R_1 \boldsymbol{\sim} I$ if and only if $R_2 \boldsymbol{\sim} I$.

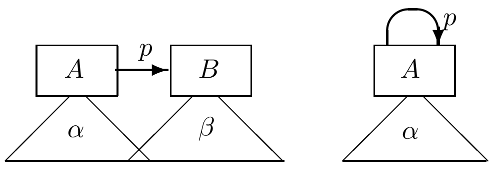
Let us study some more general properties of rule application for simple rules. We begin with the simple cases in which R_{GS} consists only of edges.

Lemma 2.3.1. *For each G-Log rule R, if R_{GS} consists only of one edge and no nodes, then $[\![R]\!]_\xi$ is a function, for ξ in $\{\sim, \boldsymbol{\sim}, \equiv\}$.*

Proof. We need to prove that for each G-Log graph G, there is exactly one G' such that $\langle G, G' \rangle \in [\![R]\!]_\xi$.

If R is not ξ-applicable, then the result holds by definition choosing G' as G.

Assume R is ξ-applicable in G, for any ξ in $\{\sim, \boldsymbol{\sim}, \equiv\}$. By hypothesis, R can have only one of the following two forms:



(1) (2)

where the nodes labeled by A and B in (1) are distinct nodes (but not necessarily label A is different from label B) or in (2) they are the same node. We prove first the fact when ξ is $\sim$.

Let R be of the form (1). Its semantics is exactly that of introducing *all possible* edges labeled by p connecting subgraphs bisimilar to A, α and B, β of G, unless they are already present. This kind of extension of G is clearly unique.

Let R be of the form (2). As shown in Example 2.2.1, condition (*iii*) in Definition 2.2.4 ensures that the only (minimal) way to generate a graph satisfying the rule is that of adding a self-loop for each node matching with A, α.

When ξ is $\boldsymbol{\sim}$ or $\equiv$, the situation is similar (and easier than for $\sim$ as concerns case (2)). □

Now we extend the above lemma to the case in which R contains several edges.

Proposition 2.3.4. *For each edge-adding rule R, $[\![R]\!]_\xi$ is a function, for ξ in $\{\sim, \boldsymbol{\sim}, \equiv\}$.*

Proof. R contains n green edges, with $n > 0$, by definition of rule. Consider the rules R_i, $i = 1, \ldots, n$, obtained by removing from R the green edges $1, \ldots, i-1, i+1, \ldots, n$. Each R_i is of the form analyzed by Lemma 2.3.1. Let G be a G-Log graph. Consider the following procedure, parametric with respect to ξ in $\{\sim, \sim, \equiv\}$:

```
$G'' := G$;
repeat
    $G' := G''$;
    for $i = 1$ to $n$ do
        let $G''$ be the result of $R_i$ applied to $G''$ with respect to $\xi$
until $G'' = G'$;
```

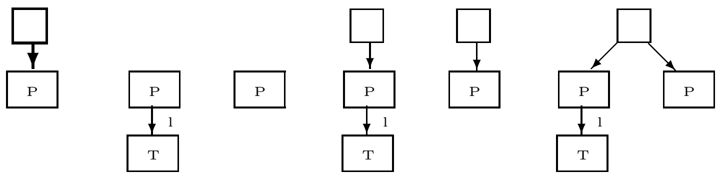
The procedure is clearly terminating. Moreover, by induction on the number n of green edges, it is easy to prove that the procedure is ensured to return a unique graph G' (use Lemma 2.3.1), that G' ξ-satisfies R and it is the minimum graph extending G fulfilling such a property. □

2.3.4 Very Simple Queries

We consider simple rules, actually very used in practice:

Definition 2.3.1. *A rule R is said to be a* very simple rule *if R_{GS} consists in one node μ and one edge $\langle \mu, lab, \nu \rangle$, with $\ell_{\mathcal{C}}(\nu) = RS$.*

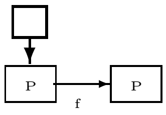
In this section we are interested in very simple rules that are queries (very simple queries—VSQ). In general, $[\![R]\!]_\xi$ applied to a concrete graph G is not a function, even when R is a very simple query with respect to G. Consider the following diagrams:[6]



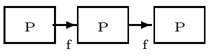
(R_1) (G_1) (G'_1) (G''_1)

Then, $[\![R_1]\!]_\xi(G_1) = \{G'_1, G''_1\}$. However, the views of G'_1 and G''_1 with respect to R_1 (see Definition 2.2.8) are bisimilar. So we could guess that $[\![\cdot]\!]_\xi$ is a function modulo bisimulation, at least with respect to a 'structured' subgraphs of G, i.e., graphs filtered by a rule. This does not hold in general, as follows from the following example concerning grandfathers, fathers, and sons:

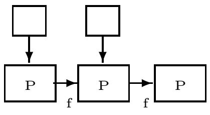
[6] As usual, square nodes can be read as *Result* nodes and outgoing edges as *connects* edges.



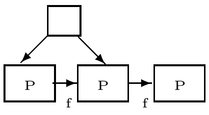
(R_2)



(G_2)



(G'_2)



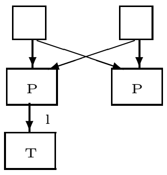
(G''_2)

It holds that $[\![R_2]\!]_\xi(G_2) = \{G'_2, G''_2\}$. However, G'_2 and G''_2 are not bisimilar. The uniqueness (modulo bisimulation) is ensured only when the various parts of G matching with R_{RS} are all independent (unfolded views).

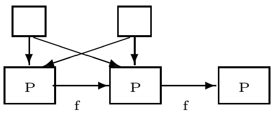
However, a sort of regularity of the semantics of rule application can be obtained by considering

$$G^R_\xi =_{def} \bigsqcup_{G' \in [\![R]\!]_\xi(G)} G'$$

Such a graph is unique (up to isomorphism) and it is a sort of *skeleton* from which all elements of $[\![R]\!]_\xi(G)$ can be obtained. As an instance, for the examples above:



($G^{R_1}_1$)



($G^{R_2}_2$)

Lemma 2.3.2. *Let G be a G-Log graph and R be a VSQ with respect to G. Then, for ξ in $\{\sim, \thicksim, \equiv\}$, there is a unique graph (up to isomorphism) G^R_ξ such that:*

1. *$G^R_\xi \models R$,*
2. *$\forall G' \in [\![R]\!]_\xi(G)$ it holds that $G' \sqsubseteq G^R_\xi$, and*
3. *G^R_ξ is minimal with respect to properties (1) and (2).*

Proof. If R is not ξ-applicable to G, then choose G^R_ξ as G. Otherwise since, by definition of query, no node labeled by *result* and edge labeled by *connects* is in G, whenever there is a subgraph G' of G such that $R_{RS}\xi G'$, add a node μ and an edge $\langle \mu, lab, \nu \rangle$. Moreover, keep track of all nodes μ, ν of this kind. When all the G' of that form have been processed, add edges from all nodes μ to all nodes ν, uniquely obtaining the graph G^R_ξ. □

Proposition 2.3.5. *Let G be a G-Log graph and R be a VSQ with respect to G. For ξ in $\{\sim, \thicksim, \equiv\}$, define*

$$views(G, R, \xi) = \{G'^{\uplus}|_R : \exists G' \in [\![R]\!]_\xi(G)\}$$

Then for each $I_1, I_2 \in views(G, R, \xi)$, it holds that I_1 is isomorphic to I_2.

Proof. Assume $I_1, I_2 \in views(G, R, \xi)$, I_1 and I_2 distinct graphs. This means that there are two graphs G_1 and G_2 in $[\![R]\!]_\xi(G)$ such that $I_1 = G_1^{\uplus}|_R$ and $I_2 = G_2^{\uplus}|_R$. Since $G_i \in [\![R]\!]_\xi(G)$ it holds that for each $G' \sqsubseteq G$ such that $R_{RS}\xi G'$ there is an edge between a *Result* node (not occurring in G) and a node of G. This means, by definition of unfolded view, that a graph exactly composed by G' and the just mentioned node and edge is both in I_1 and in I_2, and, moreover, this is an isolated subgraph of both I_1 and I_2. Nodes and edges are introduced in I_1 and I_2 only in this way. This ensures that $I_1 \equiv I_2$. □

To reach a more convincing deterministic result for the semantics, we suggest to add determinism to the definition: we define the *deterministic semantics of R*:

$$[\![R]\!]_\xi^{det}(G) = G'$$

for $G' \in [\![R]\!]_\xi(G)$ and G' contains at most one node more than G.

Proposition 2.3.6. *Let G be a G-Log graph and R be a VSQ with respect to G. For ξ in $\{\sim, \thicksim, \equiv\}$, then $[\![R]\!]_\xi^{det}(G)$ is well-defined, i.e., $[\![R]\!]_\xi^{det}(G)$ is a function.*

Proof. Assume, by contradiction, that $G_1, G_2 \in [\![R]\!]_\xi(G)$ and that they differ by G for at most one node.

If R is not ξ-applicable, then $G_1 = G_2 = G$ by definition.

Assume R is ξ-applicable. Since R is a query with respect to G, no result nodes are in G. Thus both G_1 and G_2 contains exactly one (result) node μ more than G. Without loss of generality, we can assume that it is the same node in the two graphs. New (connects) edges have been introduced from μ to the various subgraphs equivalent to R_{RS}. It is immediate to check that an edge of this form belongs to G_1 if and only if it belongs to G_2, unless one of them is not in $[\![R]\!]_\xi(G)$. □

$[\![R]\!]_\xi^{det}(G)$ can therefore be seen as a *privileged* answer to a query. Actually, it contains exactly all the information we need and does not introduce redundant nodes.

We conclude this section with a consideration that explains the rationale behind the condition (3) of being a rule.

Remark 2.3.1. Consider the graph R in Fig. 2.11, that does not fulfill requirement (3) of being a rule. It intuitively says that for all nodes labeled A you need to have a node labeled by A connected with it by an edge labeled by p. The application of R to the trivial graph G generates a denumerable family $G'', G''', \ldots$ of graphs satisfying R. However, none of them is minimal. Moreover, notice that the graph G' does not satisfy R, as condition (3) of the definition of bisimulation is not satisfied.

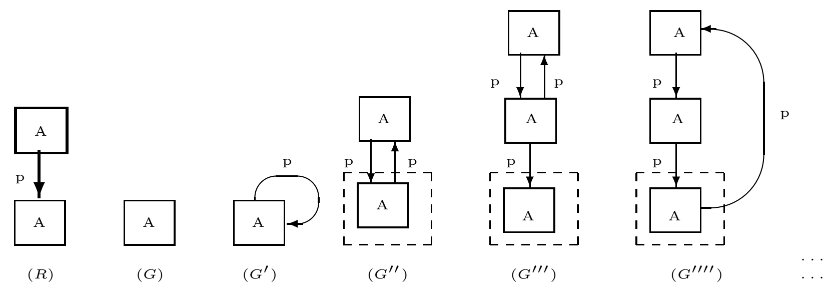


Figure 2.11. Infinite generation

2.4 Abstract Graphs and Semantics

In order to represent sets of instances sharing the same structure, we introduce now the notion of abstract graph. Following the Abstract Interpretation approach [38, 51], we see that abstract graphs can be used as a domain to abstract the computation of G-Log programs over concrete graphs. In fact, Abstract Interpretation is a theory of semantics approximation which is mainly used for the construction of semantics-based program analysis algorithms, the comparison of formal semantics, the design of proof methods. As a first approximation, abstract interpretation can be understood as a non standard semantics, in which the domain of values (usually called concrete values) is replaced by a domain of descriptions of values (called abstract values), and in which the operators are given a corresponding non standard interpretation.

Moreover, our approach to obtain abstract graphs can also be seen as an alternative view of reasoning on schemata and instances of a database or a WWW site, coherently with the *Dataguide* approach of [54]. We recall that operating on schemata may be useful whenever the user wants to investigate on general properties on the structure of a given semistructured database.

Definition 2.4.1. *A* (G-Log) abstract graph *is a G-Log graph such that:*

1. $(\forall x \in N \cup E)(\ell_{\mathcal{C}}(x) = black)$,
2. $(\forall x \in N)(\ell_{\mathcal{S}}(x) = \bot)$, *i.e., an abstract graph has no values.*

With $\mathcal{G}^{\mathcal{A}}$ *we denote the set of G-Log abstract graphs.*

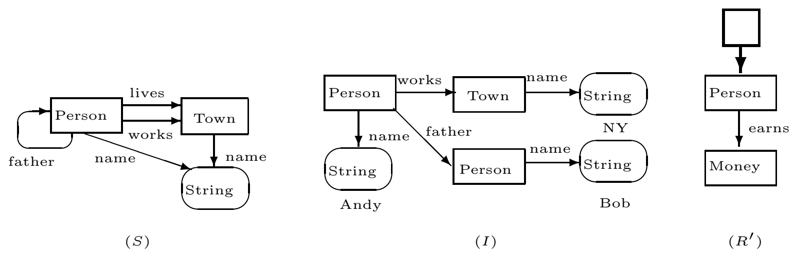


Figure 2.12. A schema, an instance, and a rule

Let us use once more the notion of bisimulation to re-formulate the G-Log concepts of instance and schema. Intuitively, an abstract graph represents a concrete graph if it contains its skeleton while disregarding multiplicities and values.

Definition 2.4.2. *A concrete graph I is* an instance *of an abstract graph G if $(\exists I' \sqsupseteq I)(G \sim I')$. In this case G is said to be a* schema *for I. I' is said to be a* witness *of the relation schema-instance.*

In Fig. 2.12 there is an example of application of the definition above. (S) represents (I). To build the witness (I'), add to (I) an edge labeled by *works* linking the entity node *Person* of *Bob* with the *entity* node *Town*. Moreover, add edges labeled by *lives* from the two nodes labeled *Person* to the node labeled *Town*, and add also an edge reverse to the *father* edge. It is easy to check that a bisimulation from S to I' is uniquely determined.

The notions of *applicability* and *satisfiability* for abstract graphs are the same as in Definitions 2.2.4 and 2.2.2. This also holds for the semantics definitions for rules and programs. Anyway, the semantics based on bisimulation $\sim$ is, in a sense, the less precise and, thus, it is the most suited for abstract computations.

The following properties can be immediately derived from the definitions above.

Lemma 2.4.1.

(a) If I is an instance of G with witness I', then for all I'' such that $I \sqsubseteq I'' \sqsubseteq I'$ it holds that I'' is an instance of G.
(b) If I is a concrete graph, G is an abstract graph, with $I \sim G$, then I is an instance of G.
(c) If I is a concrete graph, G, G' are abstract graphs, with $G \sim G'$, then I is an instance of G if and only if I is an instance of G'.

We have already observed after Definition 2.1.2 that given a G-Log graph G_0, the set $\langle \{G \text{ is a G-Log graph} \,:\, G \sqsubseteq G_0\}, \sqsubseteq \rangle$ is a *complete lattice.*[7] In the rest of this section we assume that every (concrete and abstract) graph belongs to this lattice. Under this hypothesis we may properly deal with the $\sqsubseteq$ relation between graphs.

A Galois connection ([38]),[8] which is used in our context to approximate in the best way any concrete graph by an abstract one, between $\mathcal{G}^{\mathcal{A}}$ and $\wp(\mathcal{G}^{\mathcal{C}})$ can be obtained by considering the concretization function $\gamma : \mathcal{G}^{\mathcal{A}}_{/\sim} \longrightarrow \wp(\mathcal{G}^{\mathcal{C}})$:

$$\gamma(G) = \{I \,:\, I \text{ is an instance of } G\}$$

and its adjoint abstraction function $\alpha : \wp(\mathcal{G}^{\mathcal{C}}) \longrightarrow \mathcal{G}^{\mathcal{A}}_{/\sim}$ defined by

$$\alpha(S) = \sqcup\{G \in \mathcal{G}^{\mathcal{A}} \,:\, \gamma(G) \subseteq S\}.$$

Algorithmically, given a set of instances S, we may build up its abstraction $\alpha(S)$ by taking the union of all their nodes and edges. Moreover, applying standard techniques, we can build the minimum graph $\sim$-equivalent to that graph. This graph is unique up to isomorphism, and it can be computed without knowing G_0. Using the techniques in [79], this simplification can be performed in time $O(m \log n + n)$, where m is the number of edges and n the number of nodes.

The abstraction function for rules can be obtained exactly in the same way as for concrete graphs. Thus, when R is a rule, $\alpha(R)$ denotes the graph obtained by deleting values (i.e., $\ell_{\mathcal{S}} = \bot$) from R and then computing the minimum graph $\sim$-equivalent to it.

The following two auxiliary results will be useful in order to prove monotonicity and injectivity of the function γ just defined.

Lemma 2.4.2. *If $G_1 = \langle N_1, E_1, \ell_1 \rangle \sqsubseteq G_2 = \langle N_2, E_2, \ell_2 \rangle$ and $G_1 \sim G' = \langle N', E', \ell' \rangle$, then there is $G'' \sqsupseteq G'$ such that $G'' \sim G_2$.*

Proof. Without loss of generality, assume that $N' \cap N_2 = \emptyset$. Let b' such that $G_1 \overset{b'}{\sim} G'$. Let $N'' = N'$, $E'' = E'$, $\ell'' = \ell'$, $b'' = b'$.

- for all $n \in N_2 \setminus N_1$ let $N'' = N'' \cup \{n\}$, $b'' = b'' \cup \{(n, n)\}$, and $\ell'' = \ell'' \cup \{(n, \ell_2(n))\}$;
- for all $\langle m, \lambda, n \rangle$ in $E_2 \setminus E_1$, let $E'' = E'' \cup \{\langle \mu, \lambda, \nu \rangle : m b'' \mu, n b'' \nu\}$.

It is immediate to check that $G_2 \overset{b''}{\sim} G'' = \langle N'', E'', \ell'' \rangle$. □

[7] Assume, for instance, to deal with Web sites. G_0 can be chosen as the graph obtained by disjoint union of all the instances and schemata of all Web sites.

[8] Given two posets $\langle A, <^A \rangle$ and $\langle C, <^C \rangle$, a *Galois connection* is a pair of maps (α, γ) such that: $\alpha : A \mapsto C$, $\gamma : C \mapsto A$ and $\forall a \in A, \forall c \in C : \alpha(a) <^C c \Leftrightarrow a <^A \gamma(c)$.

Lemma 2.4.3. *If $G \sqsubseteq G_1 \sim G_2 \sqsubseteq G_3 \sim G$, then $G \sim G_1 \sim G_2 \sim G_3$.*

Proof. It is sufficient to prove that $G \sim G_1$; the remaining part of the claim follows by the fact that $\sim$ is an equivalence relation. Let a and b be the two bisimulations such that $G_1 \overset{a}{\sim} G_2$ and $G_3 \overset{b}{\sim} G$. By Lemma 2.4.2, there is G_0 such that $G_1 \sqsubseteq G_0$ and $G_0 \overset{a'}{\sim} G_2$, where a' extends a as in the proof of that lemma. We will refer to a' simply as a and we call $c = a \circ b$. It is easy to verify that c is monotonic; thus we can build two infinite descending chains:

$$\begin{array}{l} G_1 \sqsupseteq c(G_1)(\sqsupseteq G) \sqsupseteq c(c(G_1)) \sqsupseteq c(c(c(G_1)))\dots \\ G_0 \sqsupseteq c(G_0)(= G) \sqsupseteq c(c(G_0)) \sqsupseteq c(c(c(G_0)))\dots \end{array}$$

Since the graphs are finite, there must be an integer n such that

$$c^n G_1 = c^{n+1} G_1 \wedge c^n G_0 = c^{n+1} G_0$$

(assuming that the graph G is nonempty, the fixed points above must be non empty). Moreover, by construction, it holds that:

$$\begin{array}{l} G_1 \sim c(G_1) \sim c(c(G_1)) \sim c(c(c(G_1)))\dots \\ G_0 \sim c(G_0)(= G) \sim c(c(G_0)) \sim c(c(c(G_0)))\dots \end{array}$$

In particular, $G_1 \sim c^n(G_1)$ and $G \sim c^n(G_0)$. Since the application of relation c is monotonic, it holds that

$$c^i(G_1) \sqsubseteq c^i(G_0),$$

Thus, in particular, $c^n(G_1) \sqsubseteq c^n(G_0)$. On the other hand, since $c(G_0) = G$ and $G \sqsubseteq G_1$, it holds that

$$c^{i+1}(G_0) \sqsubseteq c^i(G_1)\,.$$

Thus, $c^n(G_0) = c^{n+1}(G_0) \sqsubseteq c^n(G_1)$. This means that $c^n(G_1) = c^n(G_0)$ and, moreover, that $G \sim c^n(G_1) \sim G_1$. □

Theorem 2.4.1. *Function γ is monotonic, i.e. for any pair of abstract graphs G, G', $G \sqsubseteq G'$ implies $\gamma(G) \subseteq \gamma(G')$.*

Proof. Let $I \in \gamma(G)$. By definition of γ, there exists $I' \sqsupseteq I$ such that $I' \sim G \sqsubseteq G'$. By applying Lemma 2.4.2, there exists $I'' \sqsupseteq I'$ such that $I'' \sim G'$. By transitivity of the ordering relation, we get $I'' \sqsupseteq I$. Hence, $I \in \gamma(G')$. □

Theorem 2.4.2. *Function γ is injective, i.e. for any pair of abstract graphs G, G', $G \not\sim G'$ implies $\gamma(G) \neq \gamma(G')$.*

Proof. Assume that $\gamma(G_1) = \gamma(G_2)$, and let $I_1 \in \gamma(G_1)$ with $I_1 \sim G_1$. By the assumption, $I_1 \in \gamma(G_2)$ too. Hence, by the definition of γ, $\exists I'_1 \sqsupseteq I_1$ such that $I' \sim G_2$. Therefore,

$$G_1 \sim I_1 \sqsubseteq I_1' \sim G_2.$$

Now, let $I_2 \in \gamma(G_2)$ with $I_2 \sim G_2$. By the same reasoning, there exists $I_2' \sqsupseteq I_2$ such that $I_2' \sim G_1$. Therefore

$$G_1 \sim I_1 \sqsubseteq I_1' \sim G_2 \sim I_2 \sqsubseteq I_2' \sim G_1.$$

By lemma 2.4.3 we immediately get $G_1 \sim G_2$, concluding the proof.

□

Theorem 2.4.3. *(Correctness) Let G, G' be abstract graphs and R a rule such that $\langle G, G' \rangle \in [\![\alpha(R)]\!]_\sim$. If $I \in \gamma(G)$ and $\langle I, I' \rangle \in [\![R]\!]_\sim$, then $I' \in \gamma(G')$, i.e., the following diagram commutes:*

$$\begin{tikzcd} G \arrow[r, "\alpha(R)"] \arrow[d, "\gamma"] & G' \arrow[d, "\gamma"] \\ I \arrow[r, "R"] & I' \end{tikzcd}$$

Proof. By the hypotheses and by Lemma 2.4.2, there exist $\hat{I}$ and $\hat{I}'$ such that the following diagram holds:

$$\begin{array}{ccccccccc} R & & \dashv & I' & & \hat{I}' & \sim & G' & \models & \alpha(R) \\ & & & \sqcup & & \sqcup & & \sqcup & & \\ R|_{\{RS\}} & \sqsubseteq & & I & \sqsubseteq & \hat{I} & \sim & G & \sqsupseteq & \alpha(R)|_{\{RS\}} \end{array}$$

By the definition of satisfiability of the rule R, we may build I'' such that $I' \sqsubseteq I'' \sim \hat{I}'$. In order to build such a I'', extend I' only with arcs and nodes belonging to $\hat{I}'$; minimality conditions on G' (and thus on $\hat{I}'$) avoid redundancies. Hence, from the diagram above we get $I' \sqsubseteq I'' \sim \hat{I}' \sim G'$, i.e. $I' \in \gamma(G')$.

□

Theorem 2.4.3 guarantees the correctness of abstract computations: the application of a rule abstraction to an abstract graph safely represents the application of the corresponding concrete rule to any of its instances. The practical impact of this result is quite interesting. Consider the abstract graph S and the rule R' in Fig. 2.12. Since $\alpha(R')$ is not applicable to S, we can immediately conclude that the same rule is not applicable to any instance of S. Therefore, we may apply rules to abstract graphs in order to build complex queries, and then, once checked that they are applicable to the abstract graph we can turn to the concrete cases to get the desired answer. This is particularly interesting when the instance resides on a remote site.

Moreover, suppose we use G-Log rules to specify site instance evolution during the site life. Then, the application of the same rule to the site schema returns automatically the schema corresponding to the new site instance.[9]

[9] Of course, in this context we are interested in those site updates that would affect the schema, since schema-invariant updates do not need to be traced.

Remark 2.4.1. According to standard definitions of schema, the concept introduced in Definition 2.4.1 may be further enforced with the condition:

3. $(\forall x, y \in N)(\ell_{\mathcal{T}}(x) \neq \ell_{\mathcal{T}}(y) \vee \ell_{\mathcal{L}}(x) \neq \ell_{\mathcal{L}}(y))$, i.e. there is no repetition of nodes.

In this case, given a set S of instances, we may build up its abstraction $\alpha'(S)$ computing $\alpha(S)$ and then by collapsing all the nodes in it having the same type and label. The same technique can be applied to a rule R. α' can be seen as an abstraction less precise than α; by construction, it holds that $\alpha'(\alpha(S)) = \alpha'(S)$. The concretization function γ remains the same.

When R is a query, Theorem 2.4.3 still holds using α' in place of α. In Fig. 2.13 we present the concrete graph I, the abstract graph $G = \alpha(I)$ and the schema $S = \alpha'(I)$. For each rule R_i, $\alpha(R_i) = \alpha'(R_i)$ is obtained by removing the concrete value label (in these cases, Udine and Verona). It holds that:

– $\{I'\} = [\![R_1]\!](I)$, $\{I\} = [\![R_2]\!](I) = [\![R_3]\!](I)$;
– $\{G'\} = [\![\alpha(R_1)]\!](G) = [\![\alpha(R_2)]\!](G)$, $\{G\} = [\![\alpha(R_3)]\!]G$;
– $\{S'\} = [\![\alpha'(R_1)]\!](S) = [\![\alpha'(R_2)]\!](S) = [\![\alpha'(R_3)]\!](S)$.

When, as in this case, rule application is deterministic, the two levels of abstractions can be summarized by the following diagram:

$$\begin{tikzcd}
S \arrow[r, "\alpha'(R)"] & S_f \arrow[d, phantom, sloped, "\sqsupseteq"] \\
 & \alpha'(G_f) \\
G \arrow[uu, "\alpha'"] \arrow[r, "\alpha(R)"] & G_f \arrow[u, "\alpha'"] \arrow[d, phantom, sloped, "\sqsupseteq"] \\
 & \alpha(I_f) \\
I \arrow[uu, "\alpha"] \arrow[r, "R"] & I_f \arrow[u, "\alpha"]
\end{tikzcd}$$

This leads to a hierarchy of abstraction in the spirit of [51].

2.5 Logical Semantics of G-Log

Aim of this section is to provide a model theoretic characterization of the language G-Log. First, we show how to automatically extract a first-order formula from a G-Log graph. Then we show that G-Log concrete graphs are simply representations of Herbrand structures: they are models of the formulae associated with the rules they satisfy.

As said in Section 2.1.2, *result* nodes and their outgoing edges labeled by *connects* are represented without writing explicitly the labels.

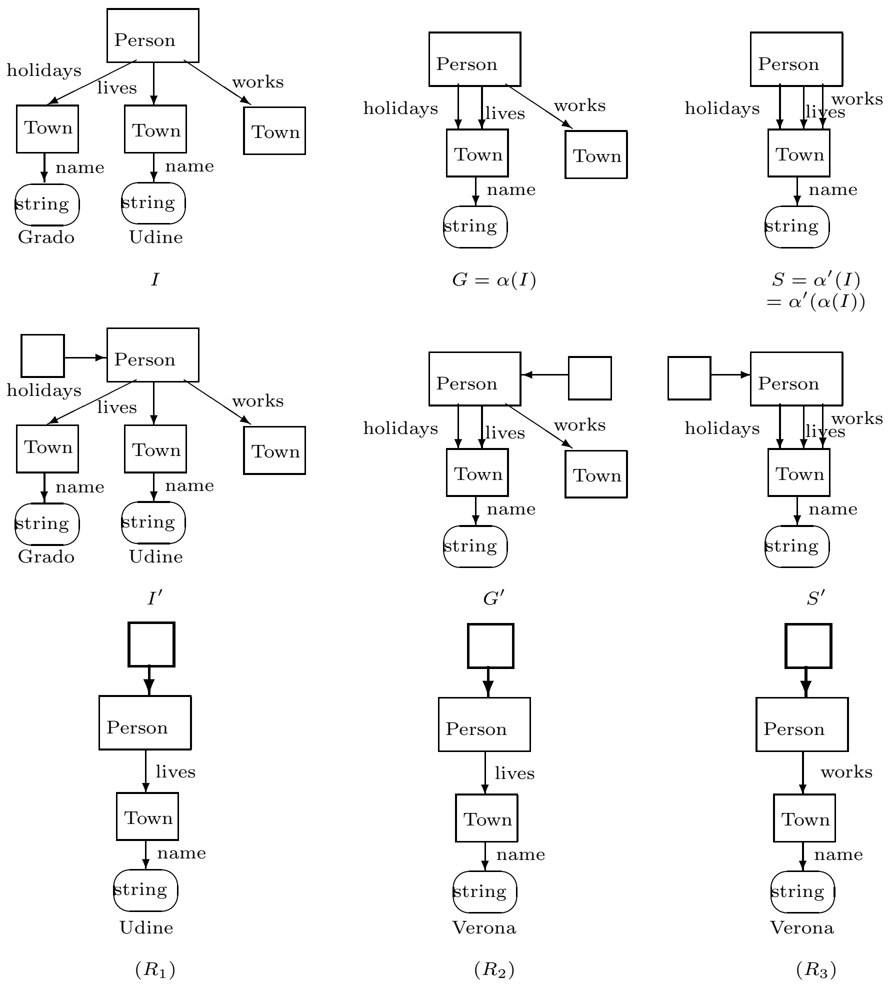


Figure 2.13. Two levels of abstraction

We write $\phi(x_1, \ldots, x_n)$ to denote that ϕ is a first-order formula with free variables among $x_1, \ldots, x_n$. Moreover, $[\ell_{\mathcal{N}}(n)](x_1, \ldots, x_n)$ denotes the atom $p(x_1, \ldots, x_n)$ where p is $\ell_{\mathcal{N}}(n)$. Similarly for $\ell_{\mathcal{L}}$ and $\ell_{\mathcal{S}}$.

2.5.1 Formulae for G-Log Rules

In this subsection we describe how to obtain a first-order formula from each G-Log rule without negation either in its nodes or in its edges.

Definition 2.5.1. *A* G-Log formula *is a closed first-order formula of the following form:*

$$\forall x_1 \cdots x_h \, (B_1(x_1, \ldots, x_h) \to \exists z_1 \cdots z_k \, B_2(x_1, \ldots, x_h, z_1, \ldots, z_k))$$

where x_i, z_i are variables and the B_i are conjunctions of atoms.

Remark 2.5.1. Observe that the existential quantification of the variable on the r.h.s. of the implication causes that the formula can not be encoded as a simple Horn clause of DATALOG.

We show how to associate a G-Log formula to every solid G-Log rule. We begin with the semantics based on directional bisimulation $\sim$; then we show how to modify the technique according to the other two semantics.

Definition 2.5.2. *Let $R = \langle N, E, \ell \rangle$ be a G-Log rule; we define the G-Log formula $\Phi_R^{\sim}$ and the formula $\Psi_R^{\sim}$ as follows:*

1. *$\forall n \in N$ associate a distinct variable $\nu(n)$.*
2. *$\forall n \in N$ let φ_n be the formula:*
$$[\ell_{\mathcal{L}}(n)](\nu(n)) \wedge [\ell_{\mathcal{T}}(n)](\nu(n)) \wedge [\ell_{\mathcal{S}}(n)](\nu(n)).$$
If $\ell_{\mathcal{S}}(n) = \bot$, then the last conjunct is omitted.
3. *$\forall e = \langle m, \langle c, \ell_{\mathcal{L}}(e) \rangle, n \rangle \in E$ let φ_e be the formula:*
$$[\ell_{\mathcal{L}}(e)](\nu(m), \nu(n)).$$
4. *Let $n_1, \ldots, n_h$ be the nodes of N such that $\ell_{\mathcal{C}}(e) = RS$,*
5. *Let $n'_1, \ldots, n'_k$ be the nodes of N such that $\ell_{\mathcal{C}}(e) = GS$,*
6. *The formula $\Phi_R^{\sim}$ is:*
$$\forall \nu(n_1) \cdots \nu(n_h) \left(\begin{array}{l} \left(\bigwedge_{n \in N, \ell_{\mathcal{C}}(n) = RS} \varphi_n \wedge \bigwedge_{e \in E, \ell_{\mathcal{C}}(e) = RS} \varphi_e \right) \to \\ \exists \nu(n'_1) \cdots \nu(n'_k) \left(\begin{array}{l} \bigwedge_{n \in N, \ell_{\mathcal{C}}(n) = GS} \varphi_n \wedge \\ \bigwedge_{e \in E, \ell_{\mathcal{C}}(e) = GS} \varphi_e \end{array} \right) \end{array} \right)$$
7. *The formula $\Psi_R^{\sim}$ is:*
$$\exists \nu(n_1) \cdots \nu(n_h) \left(\bigwedge_{n \in N, \ell_{\mathcal{C}}(n) = RS} \varphi_n \wedge \bigwedge_{e \in E, \ell_{\mathcal{C}}(e) = RS} \varphi_e \right)$$

For instance, the formula $\Phi_R^{\sim}$ associated to the rule R depicted in Figure 2.14 is:[10]

$$\forall x_1 x_2 x_3 \left(\begin{array}{l} \text{Person}(x_1) \wedge \text{Town}(x_2) \wedge \text{Lives}(x_1, x_2) \wedge \text{Town}(x_3) \wedge \\ \text{Studies}(x_1, x_3) \to \exists z_1 (\text{result}(z_1) \wedge \text{connects}(z_1, x_1)) \end{array} \right)$$

[10] We omit the type information for the sake of readability.

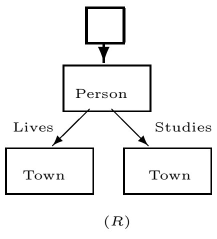


Figure 2.14. G–Log rule

Logical formulae corresponding to G-Log graphs are different if we study the other two semantics. With the semantics based on the concept of graph isomorphism $\equiv$, rule R of Fig. 2.14 represents the query 'collect all the people living and studying in two *different* towns'.

In the construction of $\Phi_R^{\equiv}$ we need to force the fact that the two towns must be distinct. This can be done by adding an inequality constraint between the variables identifying the nodes x_2 and x_3:

$$\forall x_1 x_2 x_3 \left(\begin{array}{l} \text{Person}(x_1) \wedge \text{Town}(x_2) \wedge \text{Lives}(x_1, x_2) \wedge \text{Town}(x_3) \wedge \\ \text{Studies}(x_1, x_3) \wedge x_2 \neq x_3 \ \rightarrow \\ \exists z_1 (\text{result}(z_1) \wedge \text{connects}(z_1, x_1)) \end{array} \right)$$

More generally, we will require that all nodes of the graph are distinct:

Definition 2.5.3. *Given a G-Log rule $R = \langle N, E, \ell \rangle$, and the formula*

$$\Phi_R^{\sim} = \forall \nu(n_1) \cdots \nu(n_h) \left(B_1 \rightarrow \exists \nu(n'_1) \cdots \nu(n'_k) B_2 \right) ,$$

then the formula $\Phi_R^{\equiv}$ is:

$$\forall \nu(n_1) \cdots \nu(n_h) \left(\begin{array}{l} \left(B_1 \wedge \displaystyle\bigwedge_{1 \leq i < j \leq h} \nu(n_i) \neq \nu(n_j) \right) \rightarrow \\ \exists \nu(n'_1) \cdots \nu(n'_k) \left(\begin{array}{ll} B_2 \wedge \displaystyle\bigwedge_{1 \leq i < j \leq k} & \nu(n'_i) \neq \nu(n'_j) \\ \wedge \displaystyle\bigwedge_{1 \leq i \leq h, 1 \leq j \leq k} & \nu(n_i) \neq \nu(n'_j) \end{array} \right) \end{array} \right)$$

Similarly, formula $\Psi_R^{\equiv}$ can be obtained by adding $\bigwedge_{1 \leq i < j \leq h} \nu(n_i) \neq \nu(n_j)$ to the conjuncts of $\Psi_R^{\sim}$.

Consider now the semantics based on $\sim$, the rule R, and the concrete graph G of Fig. 2.15. R is $\sim$-applicable (Definition 2.2.3) to (G) but is not ξ-applicable for ξ in $\{\equiv, \sim\}$.

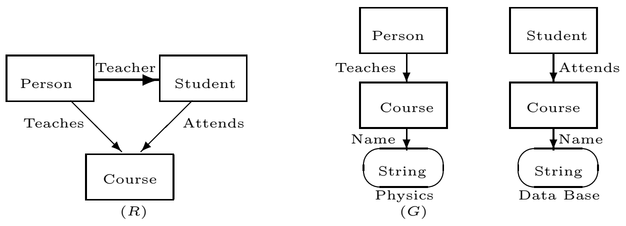


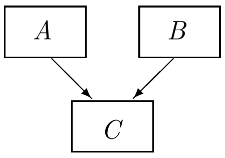
Figure 2.15. A G–Log rule and a concrete graph

The G-Log formula $\Phi^{\sim}_{R}$:

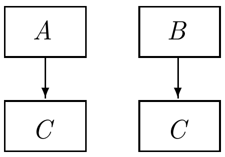
$$\forall x_1 x_2 x_3 \left(\begin{array}{l} \text{Person}(x_1) \wedge \text{Student}(x_2) \wedge \text{Course}(x_3) \wedge \text{Teaches}(x_1, x_3) \\ \wedge\text{Attends}(x_2, x_3) \ \rightarrow \text{Teacher}(x_1, x_2) \end{array} \right)$$

represents the query 'if a person teaches a course and a student attends the *same* course, then the person is that student's teacher'. The semantics based on bisimulation requires a weaker condition, since the constraint 'the same' cannot be forced. This means that, in the $\sim$-semantics the rule requires that whenever a student attends a course and a person teaches some (other?) course, the person is that student's teacher.

Definition 2.5.4. *Given a G-Log rule R, we define* the unfolding of *R, briefly unf(R), to be the graph obtained by replacing every subgraph of R of the form*



with the subgraph:



Then we set $\Phi^{\sim}_{R} = \Phi^{\sim}_{unf(R)}$ *and* $\Psi^{\sim}_{R} = \Psi^{\sim}_{unf(R)}$.

For instance, the formula $\Phi^{\sim}_{R}$ for the rule R of Fig. 2.15 is:

$$\forall x_1 x_2 x_3 x_4 \left(\begin{array}{l} \text{Person}(x_1) \wedge \text{Student}(x_2) \wedge \text{Course}(x_3) \wedge \\ \text{Course}(x_4) \wedge \text{Teaches}(x_1, x_3) \wedge \text{Attends}(x_2, x_4) \rightarrow \\ \text{Teacher}(x_1, x_2) \end{array} \right)$$

that does not constraint the courses to be the same.

In Section 2.5.3 we formally prove that the logical semantics of rules we are describing is consistent with the semantics defined in Section 2.2.

2.5.2 Concrete Graphs as Models

In this subsection we show, independently of the operational rule, how to obtain a first-order formula, from a concrete graph; in particular, we show that concrete graphs are representations of the least Herbrand model (modulo isomorphism) of the Skolemization of that formula.

Definition 2.5.5. *Let $G = \langle N, E, \ell \rangle$ be a concrete graph. As in Definition 2.5.2, to every $n \in N = \{n_1, \dots, n_h\}$ associate a variable $\nu(n)$ and to every node n and edge e associate the formulae φ_n and φ_e, respectively. Then, the formula Φ_G associated to G is:*

$$\exists \nu(n_1) \cdots \nu(n_h) \bigwedge_{n \in N} \varphi(n) \wedge \bigwedge_{e \in E} \varphi(e)$$

Definition 2.5.6. *Let $G = \langle N, E, \ell \rangle$ be a concrete graph. We associate to G the structure $M_G = \langle D, I \rangle$ built as follows:*

1. *for every node $n \in N$ introduce a constant c_n; let D be the set $\{c_n : n \in N\}$.*
2. *$I(p(c_n)) =$ true if and only if $p(\nu(n))$ is a conjunct of Φ_G.*
3. *$I(p(c_m, c_n)) =$ true if and only if $p(\nu(m), \nu(n))$ is a conjunct of Φ_G.*

M_G is the *least* Herbrand model of the Skolemization of the formula Φ_G.

For example, let G be the concrete graph of Fig. 2.15, and c_i, with $i = 1, \dots, n$, the constants introduced for the nodes of the concrete graph. Then M_G can be expressed by the set of facts that are true:

Person(c_1), entity(c_1),	Course(c_2), entity(c_2)
String(c_3), slot(c_3), Physics(c_3),	Student(c_4), entity(c_4),
Course(c_5), entity(c_5)	String(c_6), slot(c_6), Data Base(c_6),
Teaches(c_1, c_2), Name(c_2, c_3),	Attends(c_4, c_5), Name(c_5, c_6)

2.5.3 Model Theoretic Semantics

In this subsection we highlight the relationships between the semantics of Section 2.2 and the logical view of G-Log graphs presented in Subsections 2.5.1 and 2.5.2.

Proposition 2.5.1 (Applicability). *Let G be a concrete graph and R a rule. Then R is ξ-applicable to G if and only if $M_G \models \Psi_R^{\xi}$.*

Proof. We prove first the claim when ξ is $\sim$. The fact that R is $\sim$-applicable to G means that there is $G_1 \sqsubseteq G$ such that $R_{RS} \sim G_1$. Thus, there is a function f from the nodes of R_{RS} to those of G_1 fulfilling the requirements of $\sim$. Using that f we find exactly the constants c_i of the domain D obtained by skolemization of Φ_G to be assigned to the existentially quantified variables $\nu(n_i)$ of $\Psi_R^{\sim}$ to ensure that $M_G \models \Psi_R^{\sim}$. Similarly, starting from an assignment ensuring $M_G \models \Psi_R^{\sim}$ we can build a function f such that $R_{RS} \sim G_1$ for some $G_1 \sqsubseteq G$.

To conclude the proof, notice that when computing $\Psi_{\equiv}^{\xi}$ and $\Psi_{\sim}^{\xi}$ we have kept into account the constraint to map distinct nodes into distinct objects, and the possibility given by the unfolding of a node to be mapped into distinct objects, respectively. □

Proposition 2.5.2 (Satisfiability). *Let G be a concrete graph and R a rule. Then G ξ-satisfies R if and only if $M_G \models \Phi_R^{\xi}$.*

Proof. We prove first the claim when ξ is $\sim$. G $\sim$-satisfies G when for all $G_1 \sqsubseteq G$ such that $R_{RS} \sim G_1$ there is a $G_2 \sqsupseteq G_1$ such that $R \sim G_2$. But this is exactly the meaning of the formula $\Phi_R^{\sim}$.

To conclude the proof, notice that the way to compute Ψ_R^{ξ} when ξ is $\sim$ or $\equiv$ ensures that the result holds. □

Proposition 2.5.3 (Rule application). *Let G be a G-Log concrete graph and R a rule. Then, for $\xi \in \{\sim \sim, \equiv\}$,*

$$\begin{array}{lll} G' \in [\![R]\!]_{\xi}(G) & \rightarrow & (M_G \not\models \Psi_R^{\xi} \wedge G = G') \vee \\ & & (M_G \models \Psi_R^{\xi} \wedge M_G \models \Phi_R^{\xi} \wedge G = G') \vee \\ & & (M_G \models \Psi_R^{\xi} \wedge M_G \not\models \Phi_R^{\xi} \wedge G' \neq G \wedge M_{G'} \models \Phi_R) \end{array}$$

Proof. The proof is by case analysis. Assume $G' \in [\![R]\!]_{\xi}(G)$.

1. If R is not ξ-applicable to G then $G' = G$ by definition. From Proposition 2.5.1 it holds that $M_G \not\models \Psi_R^{\xi}$. Since the l.h.s. of the implication of Φ_R^{ξ} is false, then trivially $M_G \models \Phi_R^{\xi}$.
2. If, conversely, R is ξ-applicable to G (and from Proposition 2.5.1 $M_G \models \Psi_R^{\xi}$) either:
   a) G ξ-satisfies R (and thus, by Proposition 2.5.2, $M_G \models \Phi_R^{\xi}$), or
   b) G does not ξ-satisfy R (and thus, by Proposition 2.5.2, $M_G \not\models \Phi_R^{\xi}$).
   In the former case $G' = G$; in the latter there is a $G' \sqsupseteq G$ such that G' ξ-satisfies R. Thus, by Proposition 2.5.2, $M_{G'} \models \Phi_R^{\xi}$.

□

Notice that it can be the case that

$$M_G \models \Psi_R^{\xi} \wedge M_G \not\models \Phi_R^{\xi} \wedge G' \neq G \wedge M_{G'} \models \Phi_R$$

but $G' \notin [\![R]\!]_{\xi}(G)$. This happens when G' is not a minimal extension of G. Thus, the converse direction of th above proposition is not always true. However:

Corollary 2.5.1. *Let G be a G-Log concrete graph and R a rule. Then, for $\xi \in \{\sim\!\sim, \equiv\}$,*

$$\exists G'(G' \in [\![R]\!]_\xi(G) \wedge G' \neq G) \quad \leftrightarrow \left(\begin{array}{l} M_G \models \Psi_R^\xi \wedge M_G \not\models \Phi_R^\xi \wedge \\ (\exists G' \sqsupseteq G)(M_{G'} \models \Phi_R) \end{array} \right)$$

Proof. The $(\rightarrow)$ direction follows immediately from Proposition 2.5.3. Assume now that $M_G \models \Psi_R^\xi \wedge M_G \not\models \Phi_R^\xi \wedge (\exists G' \sqsupseteq G)(M_{G'} \models \Phi_R)$. From Propositions 2.5.1 and 2.5.2 we know that R is ξ-applicable to G and G does not ξ-satisfy R. Then, by Proposition 2.3.3 this is sufficient to ensure that there is a minimal G' extending G and ξ-satisfying R. □

To sum up, given a rule R and a graph G, the model-theoretic interpretation of the rule application is that of finding a (minimal) $G' \sqsupseteq G$ such that $M_{G'} \models \Phi_R^\xi$.

More generally, the effect of the application of the consecutive rules $R_1, \dots, R_n$ to an initial concrete graph G is that of producing a (nondeterministic) path of the form:

$$G \overset{R_1}{\Rightarrow} G_1 \overset{R_1}{\Rightarrow} \cdots \overset{R_n}{\Rightarrow} G_n$$

where $M_{G_i} \models \Phi_{R_j}^\xi$ for all $j \leq i$.

As a final remark, also for abstract graphs it is possible to develop a logical semantics. However, while a concrete graph leads to an existential formula, an abstract graph leads to universally quantified formulae in which a lot of *closure* properties are to be stored. The difficulty of handling those formulae requires further work.

2.6 Relationship with the Original G-Log Semantics

In this subsection we wish to point out the connections of a bisimulation-based semantics with the embedding-based semantics of [83, 81]. To complete the Definition 2.2.2, b is a directional *pseudo-bisimulation*, denoted by $G_0 \overset{b}{\lll_\sim} G_1$, if there is a function b from the nodes of G_0 to the nodes of G_1 fulfilling conditions 1 and 2 of the definition of bisimulation and, moreover, condition 3 for $i = 0$. We say that $G_0 \lll_\sim G_1$ if there is a b such that $G_0 \overset{b}{\lll_\sim} G_1$. $\lll_\sim$ is used to build a semantics based on the notion of *embedding* as given in [81].

It is immediate to extend the Proposition 2.3.1 proving that if R is $\sim\!\sim$-applicable, then it is $\lll_\sim$-applicable. However, also this implication is strict: in Fig. 2.16 it is represented a rule R applicable to a graph G only by this semantics.

Thus, the naturally induced ordering among relations is depicted by:

$$\begin{tikzcd}
& \equiv \arrow[d] & \\
& \sim \arrow[dl] \arrow[dr] & \\
\ll_{\sim} \arrow[dr] & & \sim \arrow[dl] \\
& \perp_{\sim} &
\end{tikzcd}$$

The bottom element, say $\perp_{\sim}$, of the above graph exists: $G \perp_{\sim} G'$ if there is a relation (not necessarily a function!) b fulfilling conditions 1 and 2 of the definition of bisimulation and, moreover, condition 3 for $i = 0$. The first Example of Fig. 2.10 denotes a case of applicability for $\sim$ but not for $\ll_{\sim}$.

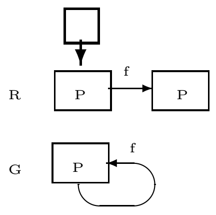


Figure 2.16. R is $\ll_{\sim}$-applicable to G

2.7 G-Log Graphs with Negation

The semantics introduced in this chapter can be extended in order to deal with rules and programs with negation (i.e., containing red dashed nodes and edges—cf. Section 2.1.1).

Intuitively, dashed edges express negative information; consider, for instance, R and G as in Fig. 2.17. R intuitively means 'collect all the people that do not live in a town named Verona'. The fact that graphs G' and G'' satisfy R can be formalized by extending the definitions of [81] concerning negation according to our semantics. However,

- G' can be obtained from G by using a sort of *failure rule* or, almost equivalently, by applying the *Closed World Assumption*: we infer that 'Mike does not live in Verona' from the fact that we cannot derive that this fact is true.
- G'' is obtained by adding the hypothesis 'Mike lives in Verona' that ensures that no subset of G fulfills R.

This kind of non-determinism is dealt with in [81] and [83] by limiting the G-Log programs to queries.

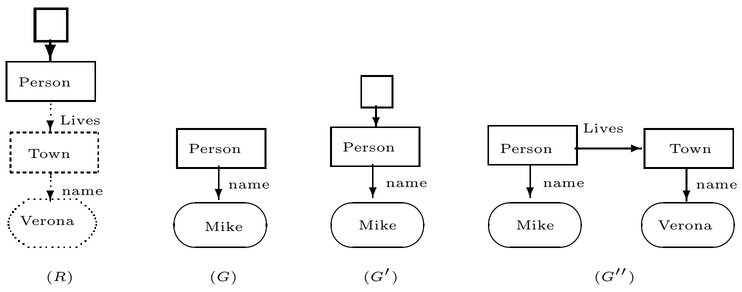


Figure 2.17. Negation as failure

2.8 Computational Issues

During the implementation of the semantics of the language G-Log, two typical graph problems must be faced: given two graphs G_1 and G_2,

1. to verify if $G_1 \xi G_2$, and
2. to verify if there is $G_3 \sqsubseteq G_1$ such that $G_3 \xi G_2$.

In the case of the relation $\equiv$, problem 1 is known as *graph isomorphism*, while problem 2 is the *subgraph isomorphism*. The former is in NP but still it is not known whether it is NP-complete or it is in P.[11] The latter in NP-complete [53].

In the case of bisimulation ($\sim$), in general problem 1 is polynomial ($O(m \log n + n)$, where m is the number of edges and n the number of nodes.; see, e.g. [79]). In particular cases this problem has a linear solution [45]. However, the second one is again NP-complete [44].

In Chapter 3 we will show that for some kinds of graphs we can improve the performances of the old implementation of G-Log based on the notion of embedding. In fact, model-checking allows to apply a polynomial time procedure to solve a subset of graphical queries on semistructured databases.

2.9 Other Languages for Semistructured Data

In this Section we briefly describe two well-known languages for semistructured databases that are respectively based on SQL and graphical queries.

[11] Actually, it is one of the candidates for membership in the (hypothetical) intermediate class NPI [53].

2.9.1 UnQL

UnQL is a language for querying data organized as a rooted directed graph with labeled edges [12]. According to the definition in [12], a rooted labeled graph is a tuple $G = \langle N, E, s, t, \nu, \ell \rangle$, where $\langle N, E, s, t \rangle$ is a multi-graph, $s, t : E \rightarrow N$ denote the source and the target of each edge, respectively; $\nu \in N$ is the root of the graph, and $\ell : E \rightarrow \mathit{Label} \cup \{\bot\}$ is a labeling function for edges.

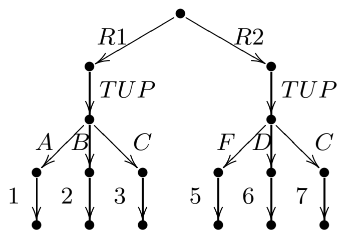


Figure 2.18. An UnQL graph, called DB

Some simple UnQL queries on the DB database of Fig. 2.18, which represents with a rooted labeled graph a relational database, are the following. The expression

$$\textbf{select}\ t\ \textbf{where}\ R1 \Rightarrow \backslash t \leftarrow DB \tag{2.1}$$

computes *the union of all trees t such that DB contains an edge* $R1 \Rightarrow t$ *emanating from the root*. There is only one such edge in DB and therefore, this query simply returns the set of tuples in R1. The returned expression is: $\{Tup \Rightarrow \{A \Rightarrow 1, B \Rightarrow 2, C \Rightarrow 3\}\}$.
The expression

$$\textbf{select}\ t\ \textbf{where}\ \backslash l \Rightarrow \backslash t \leftarrow DB \tag{2.2}$$

uses a label variable $\backslash l$ to match any edge emanating from the root. The result is the union of all tuples in both relations.

The UnQL query

$$\begin{array}{rl} \textbf{select} & \{Tup \Rightarrow \{A \Rightarrow x, D \Rightarrow z\}\} \\ \textbf{where} & R1 \Rightarrow Tup \Rightarrow \{A \Rightarrow \backslash x, C \Rightarrow \backslash y\} \leftarrow DB, \\ & R2 \Rightarrow Tup \Rightarrow \{C \Rightarrow \backslash y, D \Rightarrow \backslash z\} \leftarrow DB \end{array} \tag{2.3}$$

computes *the join of* $R1$ *and* $R2$ *on their common attribute* C *and the projection onto* A *and* D. It is easy to check that its application to the database DB has an empty result.

From the previous examples we note that more expressive power is needed in order to look for data whose depth in the graph is not predetermined by any schema. A simple query that looks arbitrarily deep into the database to find *all edges with a numeric label* is the following:

$$\textbf{select}\ \{l\}\ \textbf{where}\ \_* \Rightarrow \backslash l \Rightarrow \_ \leftarrow DB, \mathit{isnumber}(l) \tag{2.4}$$

Here the $\_*$ is a "repeated wildcard" that matches any path in the DB tree.

This brief introduction about the main features of UnQL will be used in Chapter 3 to show the applicability of the model-checking based approach to the data retrieval activity for SQL-like query languages.

2.9.2 GraphLog

GraphLog is a query language based on a graph representation of both data and queries [33]. Query graphs represent patterns, corresponding to paths in databases, that must be present or absent in database graphs, moreover they define a set of new edges (i.e. new relations) that are added to the database graphs whenever the path is found. GraphLog queries are sets of query graphs.

For example, the graph I in Fig. 2.19 is a representation of a flights schedule database. Each flight number is related to the cities it connects (by the predicates *from* and *to*, respectively) and to the departure and arrival time (by the predicates *departure* and *arrival*).

Formally, GraphLog represents databases by means of *directed labeled multigraphs*. According to the definition in [33] a *directed labeled multigraph* G is a tuple $\langle N, E, L_N, L_E, \iota, \nu, \epsilon \rangle$ where N is a finite set of nodes, E is a finite set of edges, L_N is a set of node labels, L_E is a set of edge labels, ι, the incidence function, is a function from E to N^2 that associates with each edge in E a pair of nodes from N, ν, the node labeling function, is a function from N to L_N that associates with each node in N a label from L_N, and finally ϵ, the edge labeling function, is a function from E to L_E that associates with each edge in E a label from L_E.

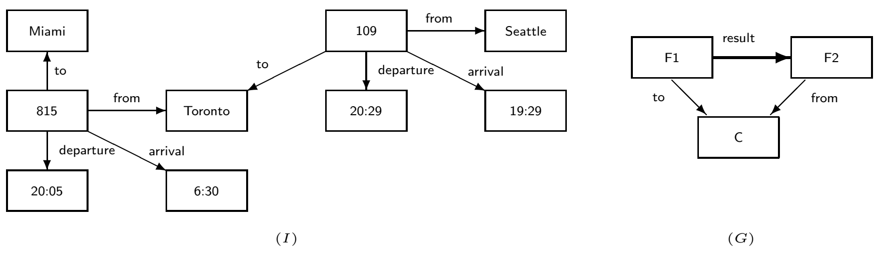


Figure 2.19. GraphLog representation of a database and a query

Moreover, in GraphLog a graphical query is a set of query graphs, each of them defining (constructive part) a set of new edges (relations) that are added to the graph whenever the path (non-constructive part) specified in the query itself is found (or not found for negative requests). More in detail, a query graph [33] is a directed labeled multigraph with distinguished edges (they define new relations that will hold, after the query application, between two objects in a chosen database whenever they fulfill the requirements of the query graph), without isolated nodes, and having the following properties:

1. the nodes are labeled by sequences of variables;
2. each edge is labeled by a literal (i.e. positive or negative occurrence of a predicate applied to a sequence of variables and constants) or by a *closure literal*, which is simply a literal followed by the positive closure operator;
3. the distinguished edge can only be labeled by a positive non-closure literal.

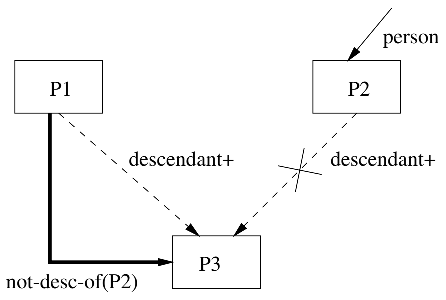


Figure 2.20. The descendants of $P1$ which are not descendants of $P2$

For example, graph (G) in Fig. 2.19 requires to find in a database two flights $F1$ and $F2$ departing from and arriving to the same city C, respectively. The edge 'result' will connect $F1$ and $F2$.

Fig. 2.20 shows another query graph. There are several different kinds of literals labeling the edges. The literal "not-desc-of($P2$)" is the (necessarily positive) literal labeling the distinguished edge. The literal "descendant+" labeling the edge between $P1$ and $P3$ is a positive closure literal. The literal "¬descendant+" labeling the edge between $P2$ and $P3$ is a negative closure literal (the negation is represented by crossing over the edge with that literal). Intuitively, this query graph expresses the query that returns a ternary predicate "not-desc-of($P1$,$P3$,$P2$)" with the descendants $P3$ of person $P1$ who are not descendants of person $P2$.

This short introduction to the language GraphLog will be used in Chapter 3 to show the main limits of the approach based on model-checking techniques for solving graphical queries on semistructured data.

3. Model-Checking Based Data Retrieval

As we said in the previous chapters, semistructured data are often represented by using data models based on directed labeled graphs [13, 84, 18]; users can retrieve information of interest with graph-based queries [50, 33, 81] or with extended SQL languages [12, 4, 16]. In both cases, the data retrieval activity requires the development of graph algorithms. In fact, queries (graphical or not) are expected to extract information stored in labeled graphs. In order to do that, it is required to perform a kind of *matching* of the "query graph" with the "database instance graph". More in detail, we need to find subgraphs of the instance of the database that are somehow similar to the query graph. In Chapter 2 we have shown that even if the problem of establishing whether two graphs are bisimilar or not is polynomial time [60, 79], the task of finding subgraphs isomorphic or bisimilar is NP-complete [44] and hence, not applicable to real-life size problems.

Graphical queries can be easily translated into logic formulae. Techniques for translating graphs in formulae have been exploited in literature [14]. The novel ideas of this chapter are to associate a *modal* logic formula Ψ to a graphical query, and to interpret database instance graphs as *Kripke Transition Systems (KTS)*. We use a modal logic with the same syntax as the temporal logic *CTL*; the notion of different instants of *time* represents the number of links the user needs to follow to reach the information of interest. In this way, finding subgraphs of the database instance graph that match the query can be performed by finding nodes of the KTS derived from the database instance graph that satisfy the formula Ψ. This is an instance of the *model-checking* problem, and it is well-known that if the formula Ψ belongs to the class *CTL* of formulae, then the problem is decidable and algorithms running in linear time on both the sizes of the KTS and the formula can be employed [27].

In particular, we identify a family of graph-based queries that represent *CTL* formulae. This is very natural and, as immediate consequence, an effective procedure for efficiently querying semistructured databases can be directly implemented on a model checker. We use a "toy" query language called $\mathbb{W}$. It can be considered as a representative of several approaches in which queries are graphical or can easily be seen as graphical (cf. e.g. Lorel [4], G-Log [81], GraphLog [33], and UnQL [50]). We will relate $\mathbb{W}$ to

UnQL, GraphLog and G-Log and show the applicability of the method for implementing (parts of) these languages. The model-checking technique for data retrieval could also be applied to non-graphical query language. Formula extraction from SQL-like queries can be done using similar ideas. We have effectively tested the approach using the model-checker NuSMV.

The structure of the chapter is as follows. Sections 3.1 and 3.2 introduce basic definitions of model-checking and of the language $\mathbb{W}$. In Section 3.3 we show how to assign a KTS to a database instance, while in Section 3.4 it is shown how to extract CTL formulae from query graphs. In Section 3.5 we report some basic results on the complexity of the data retrieval problem by using our approach. A complete example with experimental results is given in Section 3.6, while Section 3.7 is devoted to apply the method to the languages UnQL [12], GraphLog [33] and G-Log [81]. To conclude, in Section 3.8 we sum up the main results of the proposed approach.

3.1 An Introduction to Model-Checking

In this section we recall the main concepts and results of model-checking and we introduce the model-checking problem for the branching time temporal logic *CTL* [49]. In Section 3.1.2 we report the linear time algorithm for verifying that a finite-state concurrent system meets a specification expressed in *CTL* introduced in [27].

3.1.1 Transition Systems and CTL

In the last two decades model-checking [27, 73] has emerged as a promising and powerful approach to automatic verification of systems. Intuitively, a *model-checker* is a procedure that decides whether a given structure M is a model of a logical formula φ. More in detail, M is an (abstract) model of a system, typically a finite automata-like structure, and φ is a modal or temporal logic formula describing a property of interest.

Model-checking typically depends on a discrete model of a system, in fact the system's behavior is (abstractly) represented by a graph structure, where the nodes represent the system's states and the arcs represent possible transitions between the states. Graphs alone are usually too weak to provide an interesting description, so they are annotated with more specific information. A very common approach is to use *Kripke Structures* where the nodes are annotated with so-called *atomic propositions* or *Kripke Transition Systems*, where the nodes are annotated with atomic propositions and the arcs are annotated with *actions.*

Definition 3.1.1. *A* Kripke Transition System *(KTS) over a set Π of* atomic propositions *is a structure $\mathcal{K} = \langle \Sigma, \mathcal{A}ct, \mathcal{R}, I \rangle$, where Σ is a set of* *states, $\mathcal{A}ct$ is a set of actions, $\mathcal{R} \subseteq \Sigma \times \mathcal{A}ct \times \Sigma$ is a total transition relation, and $I : \Sigma \rightarrow \wp(\Pi)$ is an interpretation.*

According to [73], we assume that $\Pi \cap \mathcal{A}ct = \emptyset$.

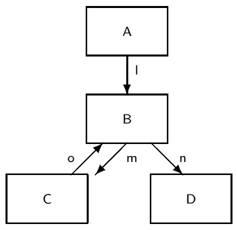


Figure 3.1. A Kripke Transition System

Figure 3.1 shows a Kripke Transition System: the initial state satisfies the atomic proposition "A" and has a transition labeled "l" to a state where the property "B" holds. From this last state there are two different transitions driven by actions "m" and "n", and so on.

The implementation of some model-checkers (e.g. NuSMV [26]) works with the structurally more simple Kripke Structures, but any Kripke Transition System induces in a natural way a Kripke Structure which codes the same information.

Remark 3.1.1. KTS are very expressive structures modeling modal logic. Kripke Structures (KS) are KTS in which transitions are not driven by actions. Standard techniques can be used to transform KTS in KS (see, for example, [73]). The idea is to associate the information about the action exchanged in a transition with the reached state instead of the transition itself.

The interpretation I in a Kripke Transition System defines local properties of states. Often we are also interested in global properties connected to the transitional behavior. For example, we might be interested in reachability properties, like "can we reach from the initial state a state where the atomic proposition P holds?". Temporal logics [49] are logical formalisms introduced for expressing such properties.

There are two main kinds of temporal logics, *linear-time* logics and *branching-time* logics. Linear-time logics are concerned with properties of paths. On the other hand, branching-time logics describe properties that depend on the branching structure of the model.

Computational Tree Logic (*CTL*) was the first temporal logic for which an efficient model-checking procedure was proposed [27]. Its syntax is as follows.

Definition 3.1.2. *Given the sets Π of atomic propositions and $\mathcal{A}ct$, CTL formulae are recursively defined as follows:*

1. *each $p \in \Pi$ is a CTL formula;*
2. *if φ_1 and φ_2 are CTL formulae, $a \subseteq \mathcal{A}ct$, then $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\mathsf{AX}_a(\varphi_1)$, $\mathsf{EX}_a(\varphi_1)$, $\mathsf{AU}_a(\varphi_1, \varphi_2)$, and $\mathsf{EU}_a(\varphi_1, \varphi_2)$ are CTL formulae.*[1]

The *length* of a formula is determined by counting the total number of operands and operators. For example the length of $\neg\varphi$ is $1 + \mathit{length}(\varphi)$.

A and E are the universal and existential path quantifiers, while X (neXt) and U (Until) are the linear-time modalities. Composition of formulae of the form $\varphi_1 \vee \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, and the modalities F (Finally) and G (Generally) can be defined in terms of the *CTL* formulae: $\mathsf{F}(\varphi) = \mathsf{U}(\mathsf{true}, \varphi), \mathsf{G}(\varphi) = \neg\mathsf{F}(\neg\varphi)$ (cf. [49]).

A *path* (called fullpath in [49]) in a KTS $\mathcal{K} = \langle \Sigma, \mathcal{A}ct, \mathcal{R}, I \rangle$ is an infinite sequence $\pi = \langle \pi_0, a_0, \pi_1, a_1, \pi_2, a_2, \ldots \rangle$ of states and actions (π_i denotes the i-th state in the path π) s.t. for all $i \in \mathbb{N}$ it holds that $\pi_i \in \Sigma$ and

– either $\langle \pi_i, a_i, \pi_{i+1} \rangle \in \mathcal{R}$, with $a_i \in \mathcal{A}ct$,
– or there are not outgoing transitions from π_i and for all $j \geq i$ it holds that a_j is the special action $\circlearrowleft$ which is not in $\mathcal{A}ct$ and $\pi_j = \pi_i$.

Definition 3.1.3. Satisfaction *of a CTL formula by a state s of a KTS $\mathcal{K} = \langle \Sigma, \mathcal{A}ct, \mathcal{R}, I \rangle$ is defined recursively as follows:*

– *If $p \in \Pi$, then $s \models p$ iff $p \in I(s)$. Moreover, $s \models \mathsf{true}$ and $s \not\models \mathsf{false}$;*
– *$s \models \neg\phi$ iff $s \not\models \phi$;*
– *$s \models \phi_1 \wedge \phi_2$ iff $s \models \phi_1$ and $s \models \phi_2$;*
– *$s \models \mathsf{EX}_a(\phi)$ iff there is a path $\pi = \langle s, x, \pi_1, \ldots \rangle$ s.t. $x \in a$ and $\pi_1 \models \phi$;*
– *$s \models \mathsf{AX}_a(\phi)$ ($a \subseteq \mathcal{A}ct$) iff for all paths $\pi = \langle s, x, \pi_1, \ldots \rangle$ s.t. $x \in a$, it holds that $\pi_1 \models \phi$;*
– *$s \models \mathsf{EU}_a(\phi_1, \phi_2)$ iff there is a path $\pi = \langle \pi_0, \ell_0, \pi_1, \ell_1 \ldots \rangle$, and $\exists j \in \mathbb{N}$ s.t.*
  – *$\pi_0 = s$,*
  – *$\pi_j \models \phi_2$, and*
  – *$(\forall i < j)$ ($\pi_i \models \phi_1$ and $\ell_i \in a$);*
– *$s \models \mathsf{AU}_a(\phi_1, \phi_2)$ iff for all paths $\pi = \langle \pi_0, \ell_0, \pi_1, \ell_1 \ldots \rangle$ such that $\pi_0 = s$, $\exists j \in \mathbb{N}$ s.t.*
  – *$\pi_j \models \phi_2$ and*
  – *$(\forall i < j)(\pi_i \models \phi_1$ and $\ell_i \in a)$.*

If $\mathcal{K} = \langle \Sigma, \mathcal{A}ct, \mathcal{R}, I \rangle$ is a KTS and φ is a *CTL* formula, we say that $\mathcal{K} \models \varphi$ if and only if for all $s \in I$ it holds that $s \models \varphi$.

[1] When a is of the form $\{m\}$ for a single action m, we simplify the notation writing $\mathsf{AX}_m, \mathsf{EX}_m, \mathsf{AX}_m, \mathsf{EX}_m$.

There are two ways in which the *model-checking problem* can be specified:

Definition 3.1.4. *The* local *model-checking: given a KTS $\mathcal{K}$, a formula φ, and a state s of $\mathcal{K}$, verifying whether $s \models \varphi$. The* global *model-checking: given a KTS $\mathcal{K}$, and a formula φ, finding all states s of $\mathcal{K}$ s.t. $s \models \varphi$.*

The above problems are usually formulated for Σ finite. In this case, the global model-checking problem for a *CTL* formula φ can be solved in linear running time on $|\varphi| \cdot (|\Sigma| + |\mathcal{R}|)$ [27]. Note that the solution of the global model-checking problem comprises the solution of the local problem.

3.1.2 A Linear Time Algorithm to Solve the Model-Checking Problem

In this section we report the algorithm introduced in [27] which has time complexity linear in both the size of the specification expressed in *CTL* and the size of the Kripke Structure modeling the concurrent system to analyze.

Assume that we want to determine whether the formula φ is true in the KS $\mathcal{K} = \langle \Sigma, \mathcal{R}, I \rangle$. The algorithm is designed to operate in more steps: the first step processes all subformulas of φ of length 1, the second step processes all the subformulas of length 2, and so on. At the end of the ith step, each state will be labeled with the set of all subformulas of length less than or equal to i that are true in the state. The expression $label(s)$ denotes this set for the state $s \in \Sigma$. When the algorithm terminates at the end of step $n = length(\varphi)$, we obtain that for all states $s \in \Sigma, s \models f$ iff $f \in label(s)$, for all subformulae $f \in \varphi$.

Note that the length of a formula is determined by counting the total number of operands and operators.

In the description of the algorithm we use the following primitives for manipulating formulae and accessing the labels associated with states:

- $arg1(\varphi)$ and $arg2(\varphi)$ give the first and second arguments of a two-argument temporal operator; thus, if $\varphi = \mathsf{AU}(f_1, f_2)$, then $arg1(\varphi) = f_1$ and $arg2(\varphi) = f_2$.
- $labeled(s, f)$ will return true (false) if state s is (is not) labeled with the formula f.
- add-labeled(s, f) adds formula f to the current label of state s.

We explicitly show the algorithm to solve the model-checking problem for the case in which the formula is $\varphi = \mathsf{AU}(f_1, f_2)$, because the other cases are either simpler or similar [27]. In the case we are presenting, the algorithm (i.e. the **procedure** label-graph(φ)) uses a *depth-first search* to explore the state graph.

```
procedure label-graph($\varphi$)
begin
    . . .
    { main operator is AU }
    begin
        $ST$ := empty-stack;
        for all $s \in \Sigma$ do $marked(s) := false$;
        L: for all $s \in \Sigma$ do
        if $\neg marked(s)$ then $au(\varphi, s, b)$
    end
        . . .
end
```

In the procedure label-graph(φ), ST is an auxiliary stack variable and the boolean procedure $stacked(s)$ indicates whether state s is currently on the stack ST. The recursive procedure $au(\varphi, s, b)$ performs the search for formula φ starting from state s. When au terminates, the boolean result parameter b will be set to true iff $s \models \varphi$. The code for procedure au is the following:

```
procedure $au(\varphi, s, b)$
begin
```

Assume that s is marked. If s is already labeled with φ, we set b to true and return. Otherwise, if s is on the stack, then we have found a cycle in the state graph on which $arg1(\varphi)$ holds but φ is never fulfilled. Thus we set b to false and return. Otherwise, we have already completed a depth-first search from s, and φ is false at s; so we must also set b to false and return in this case. Note that there is no need to distinguish between the last two cases, since the action is the same in each case.

```
if $marked(s)$ then
    begin
        if $labeled(s, \varphi)$ then
            begin $b := true$; return end;
        $b := false$; return
    end;
```

Mark state s as visited. Let $\varphi = \mathsf{AX}(f_1, f_2)$. If f_2 is true at s, φ is true at s; so label s with φ and return true. If f_1 is not true at s, the φ is not true at s; so return false.

```
$marked(s) := true$;
if $labeled(s, arg2(\varphi))$ then
    begin add-labeled($\varphi, f$); b:=true; return end
else if $\neg labeled(s, arg1(\varphi))$ then
    begin $b := false$; return end;
```

At this point of the algorithm we know if f_1 is true at s and that f_2 is not. Check to see if φ is true at all successor states of s. If there is some successor state $s1$ at which φ is false, the φ is false also at s; hence remove s from the stack and return false. If φ is true for all successor states, then φ is true at s; so remove s from the stack, label s with φ, and return true.

```
$push(s, ST)$; **for all** $s1 \in successors(s)$ **do**
    **begin**
        $au(\varphi, s1, b1)$
        **if** $\neg b1$ **then**
            **begin** $pop(ST)$; $b := false$; **return end**
    **end**;
$pop(ST)$; add-labeled$(s, \varphi)$; $b := true$; **return**
**end** of procedure $au$.
```

In order to handle an arbitrarily *CTL* formula φ, one has to successively apply the state labeling algorithm described above for the case $\mathsf{AU}(f_1, f_2)$ to the subformulae of φ, starting with simplest subformulae and working backward to φ:

```
**for** $f := length(\varphi)$ **step** -1 **until** 1 do
    label-graph$(f)$;
```

As proved in [27], since each pass through the loop takes time $O(|\Sigma|+|\mathcal{R}|)$, where Σ is the set of states and $\mathcal{R}$ is the transition relation of the KS, we conclude that the entire algorithm requires $O(length(\varphi) \times (|\Sigma| + |\mathcal{R}|))$.

We recall that a model-checker is a procedure that decides whether a given structure $\mathcal{K} = \langle \Sigma, \mathcal{R}, I \rangle$ is a model of a formula φ, i.e. whether $\mathcal{K}$ satisfies φ (written $\mathcal{K} \models \varphi$). Moreover, if $\mathcal{K} = \langle \Sigma, \mathcal{R}, I \rangle$ is a Kripke Structure, we say that $\mathcal{K} \models \varphi$ if and only if $\forall i \in I : i \models \varphi$.

Thus, given a Kripke Structure $\mathcal{K} = \langle \Sigma, \mathcal{R}, I \rangle$ and a *CTL* formula φ, we can say that a model-checker is an algorithm working in the following way:

1. we apply the procedure label-graph(φ) on $\mathcal{K}$;
2. if $\forall i \in I$ it holds $labeled(i, \varphi)$, then the answer of the model-checker is "$\mathcal{K}$ satisfies φ", otherwise we conclude that "$\mathcal{K}$ does not satisfy φ".

3.2 Syntax of the Query Language $\mathbb{W}$

In this section we describe the syntax of the language $\mathbb{W}$, a very simple graph-based language that we will use to characterize queries that have an intuitive temporal-logic interpretation. This language is very similar to G-Log [81](see also Chapter 2): on one hand, it is more simple because in our approach we do not consider the possibility to generate new relations by using rules, on the other hand, we include in the language the possibility to express the universal quantification in a very compact way (cf. Sect. 3.7.3).

Definition 3.2.1. *A* $\mathbb{W}$-graph *is a directed labeled graph* $\langle N, E, \ell \rangle$*, where* N *is a (finite) set of nodes,* $E \subseteq N \times (\mathcal{C} \times \mathcal{L}) \times N$ *is a set of labeled edges of the form* $\langle m, label, n \rangle$*,* ℓ *is a function* $\ell : N \longrightarrow \mathcal{C} \times (\mathcal{L} \cup \{\perp\})$*.* $\perp$ *means 'undefined' (or* dummy*), and*

– $\mathcal{C} = \{\ solid, dashed\ \}$ *denotes how the lines of nodes and edges are drawn.*
– $\mathcal{L}$ *is a set of labels.*

ℓ can be seen as the composition of the two single-valued functions $\ell_{\mathcal{C}}$ and $\ell_{\mathcal{L}}$. With abuse of notation, when the context is clear, we will use ℓ also for edges: if $e = \langle m, \langle c, k \rangle, n \rangle$, then $\ell_{\mathcal{C}}(e) = c$ and $\ell_{\mathcal{L}}(e) = k$. Two nodes may be connected by more than one edge, provided that edge labels be different.

Definition 3.2.2. *If* $G = \langle N, E, \ell \rangle$ *is a* $\mathbb{W}$*-graph, then*

– $G_s = \langle N_s, E_s, \ell|_{N_s} \rangle$ *is the solid subgraph of* G*, i.e.:*

$$\begin{array}{rcl} N_s & = & \{n \in N : \ell_{\mathcal{C}}(n) = solid\} \\ E_s & = & \{\langle m, \langle solid, \ell \rangle, n \rangle \in E : m, n \in N_s\} \end{array}$$

– *the* size *of* G *is* $|G| = |N| + |E|$
– *given two sets* $S, T \subseteq N$*,* T *is* accessible *from* S *if for each* $n \in T$ *there is a node* $m \in S$ *such that there is a path in* G *from* m *to* n*.*

Definition 3.2.3. *A* $\mathbb{W}$-instance *is a* $\mathbb{W}$*-graph* G *such that* $\ell_{\mathcal{C}}(e) = solid$ *for each edge* e *of* G *and* $\ell_{\mathcal{C}}(n) = solid$ *and* $\ell_{\mathcal{L}}(n) \neq \perp$ *for each node* n *of* G*.*

Definition 3.2.4. *A* $\mathbb{W}$-query *is a pointed* $\mathbb{W}$*-graph, namely a pair* $\langle G, \nu \rangle$ *with* ν *a node of* G*. A* $\mathbb{W}$-query $\langle G, \nu \rangle$ *is* accessible *if the set* N *of nodes of* G *is accessible from* $\{\nu\}$ *(the point).*

See Fig. 3.2 for some examples of $\mathbb{W}$-graphs. Dashed nodes and lines are introduced to allow a form of negation. The meaning of the first query is: collect all the teachers aged 37. The second query asks for all the teachers that have declared some age (note the use of a dummy node, i.e. a node labeled $\perp$). The third query, instead, requires to collect all the teachers that teach some course, but not Databases.

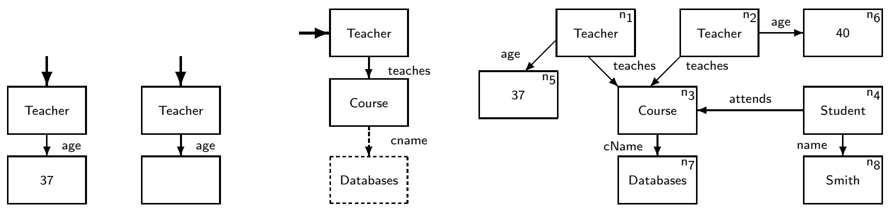


Figure 3.2. Three $\mathbb{W}$-queries and a $\mathbb{W}$-instance

Note that the $\mathbb{W}$-queries in Fig. 3.2 correspond to Very Simple Queries of G-Log (cf. Sect. 2.3.4).

3.3 $\mathbb{W}$-Instances as KTS

In this section we show how to build the *KTS* associated with a $\mathbb{W}$-instance.

Definition 3.3.1. *Let $G = \langle N, E, \ell \rangle$ be a $\mathbb{W}$-instance; we define the* KTS *$\mathcal{K}_G = \langle \Sigma_G, \mathcal{A}ct_G, \mathcal{R}_G, \mathcal{I}_G \rangle$ over Π_G as follows:*

– *The set of atomic propositions Π_G is the set of all the node labels of G:*

$$\Pi_G = \{p \,:\, (\exists n \in N)(p = \ell_{\mathcal{L}}(n)\}$$

– *The set of states is $\Sigma_G = N$.*
– *The set of actions $\mathcal{A}ct_G$ includes all the edge labels. In order to capture the notion of* before *we also add in $\mathcal{A}ct_G$ actions for the inverse relations and for the negation of all the relations introduced. Thus, if $m \xrightarrow{p} n$ belongs to E we add the actions $p, p^{-1}, \bar{p}, \bar{p}^{-1}$. We define two sets:*

$$\begin{array}{rcl} \mathcal{A}ct_G^+ & = & \{p, p^{-1} : (\exists m \in N)(\exists n \in N)(\langle m, p, n\rangle \in E)\} \\ \mathcal{A}ct_G & = & \{q, \bar{q} : q \in \mathcal{A}ct_G^+\} \end{array}$$

Moreover, we can assume, for each state s with no outgoing edges in E (a leaf) to add a self-loop edge labeled by the special action $\circlearrowleft$ that is not in $\mathcal{A}ct_G$.
– *The ternary transition relation $\mathcal{R}_G$ is defined as follows: let $\tilde{E}_G = E \cup \{\langle n, p^{-1}, m\rangle \,:\, \langle m, p, n\rangle \in E\}$. Then*

$$\mathcal{R}_G = \tilde{E}_G \cup \{\langle m, \bar{p}, n\rangle \,:\, m, n \in N, p \in \mathcal{A}ct_G^+, \langle m, p, n\rangle \notin \tilde{E}\}.$$

– *The interpretation function $\mathcal{I}_G$ can be defined as follows. In each state n the only formulae that hold are the unary atom $\ell_{\mathcal{L}}(n)$ and* true.[2]

$$\mathcal{I}_G(n) \quad = \quad \{\mathsf{true}, \ell_{\mathcal{L}}(n)\}$$

Observe that: $|\mathcal{R}_G| = |\mathcal{A}ct_G^+| \cdot |N|^2 \leq |E| \cdot |N|^2$ since for each pair of nodes, exactly one between q or $\bar{q}$ holds, for $q = p$ or $q = p^{-1}$, $q \in \Pi_G$.

Example 3.3.1. Consider the graph $G = \langle \{n_1, \ldots, n_8\}, E\rangle$, of Fig. 3.2. Then:

– $\Pi_G = \{$ Teacher, Course, Student, 37, 40, Databases, Smith$\}$
– $\Sigma_G = \{n_1, \ldots, n_8\}$
– $\mathcal{A}ct_G = \{$ teaches, teaches^{-1}, $\overline{\text{teaches}}$, $\overline{\text{teaches}^{-1}}$, attends, …
age, …, cName, …, name, …$\}$
$\mathcal{A}ct_G^+ = \{$teaches, …, name, teaches^{-1}, …, name$^{-1}\}$.

[2] The two unary atoms represent the basic local properties of a system state. Other properties can be added, if needed.

– $\tilde{E}_G$ is the union of E with the set

$$\begin{array}{l}\{ \quad \langle n_3, \text{teaches}^{-1}, n_1\rangle, \langle n_3, \text{teaches}^{-1}, n_2\rangle, \langle n_3, \text{attends}^{-1}, n_4\rangle, \\ \quad\; \langle n_5, \text{age}^{-1}, n_1\rangle, \langle n_6, \text{age}^{-1}, n_2\rangle, \langle n_7, \text{cName}^{-1}, n_3\rangle, \\ \quad\; \langle n_8, \text{name}^{-1}, n_4\rangle \quad \}\end{array}$$

Then $\mathcal{R}_G$ is the union of $\tilde{E}_G$ with the "complement" set and the set of self-loop edges labeled by $\circlearrowleft$:

$$\begin{array}{l}\{\langle m, \bar{p}, n\rangle \,:\, m, n \in \Sigma_G, p \in \mathcal{A}ct_G^{+} \wedge \langle m, p, n\rangle \notin \tilde{E}_G\} \cup \\ \{\langle m, \circlearrowleft, m\rangle \,:\, m \in \{n_5, n_6, n_7, n_8\}\}\end{array}$$

– The interpretation $\mathcal{I}$ is:

$$\begin{array}{rcl}\mathcal{I}(n_1) & = & \{\mathsf{true}, \text{Teacher}\} \\ \mathcal{I}(n_2) & = & \{\mathsf{true}, \text{Teacher}\} \\ \vdots & \vdots & \vdots \\ \mathcal{I}(n_7) & = & \{\mathsf{true}, \text{Databases}\} \\ \mathcal{I}(n_8) & = & \{\mathsf{true}, \text{Smith}\}\end{array}$$

3.4 CTL-Based Semantics of $\mathbb{W}$-Queries

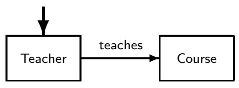
In this section we show how to extract *CTL* formulae from $\mathbb{W}$-queries. We associate a formula $\Psi_\nu(G)$ to a query (a $\mathbb{W}$-pointed graph) $\langle G, \nu\rangle$. Such a formula will give us the possibility to define a model-checking based methodology for querying a $\mathbb{W}$-instance. We anticipate this definition in order to make the meaning of formula encoding more clear.

Definition 3.4.1 (Querying). *Given a $\mathbb{W}$-instance I and a $\mathbb{W}$-query $\langle G, \nu\rangle$, let $\mathcal{K}_I$ be the KTS associated with I and $\Psi_\nu(G)$ the CTL formula associated with $\langle G, \nu\rangle$.* Querying *I with G amounts to solve the global model-checking problem for $\mathcal{K}_I$ and $\Psi_\nu(G)$, namely find all the states s of $\mathcal{K}_I$ such that $s \models \Psi_\nu(G)$.*

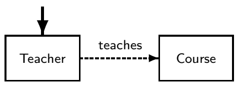
Remark 3.4.1. In order to correctly solve the global model-checking problem for a KTS $\mathcal{K}_I$ and a formula $\Psi_\nu(G)$, it is important to explicitly individuate the initial states of $\mathcal{K}_I$. In fact, the set of states of $\mathcal{K}_I$ must be accessible from the set of initial states (cf. Def. 3.2.2) because model-checking techniques are based on algorithms of graph visit.

3.4.1 Technique Overview

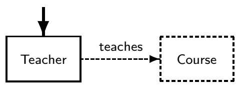
In order to explain the technique, we start our analysis by considering simple queries in which the pointed graph consists of two nodes (Fig. 3.3). Query R_1 has no dashed part: only positive information is required. Its meaning is to



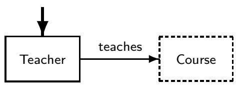
(R_1) Collect all the teachers of some course



(R_2) Collect all the teachers that do not teach all courses (i.e., s.t. there is a course that they do not teach)



(R_3) Collect all the teachers that teach no courses (i.e., for each course they do not teach it)



(R_4) Collect all the teachers that teach all courses (i.e., for each course they do not not teach —they teach— it).

Figure 3.3. Simple queries

look for nodes labeled Teacher that are connected with nodes labeled Course by an edge labeled teaches. The *CTL* formula must express the statement "In this state Teacher formula is true and there is one next state reachable by an edge labeled teaches, where the Course formula is true", i.e.

$$\mathsf{Teacher} \land \mathsf{EX}_{\mathsf{teaches}}(\mathsf{Course})$$

The *CTL* operator *neXt* (used either as EX or AX) captures the notion of following an edge in the graph. Thanks to this operator, we can easily define a directed path on the graph, nesting formulae that must be satisfied by one (or every) next state.

Query R_2 contains a dashed edge teaches that introduces a negative information. R_2 requires the existence of two nodes, and the *non*-existence of one edge labeled teaches between them. We can express this statement by

$$\mathsf{Teacher} \land \mathsf{EX}_{\overline{\mathsf{teaches}}}(\mathsf{Course})$$

The availability of the negation of the predicate symbol teaches, allows us to say that the relation teaches does not hold between two nodes is the same as requiring that, between the same pair of nodes, the relation $\overline{\mathsf{teaches}}$ holds.[3]

The meaning of query R_3, where there are dashed edges and nodes, is rather different. This formula is true if "there is a node labeled Teacher s.t., for all the nodes labeled Course, the relation teaches is not fulfilled". A *CTL* formula that states this property is the following:

$$\mathsf{Teacher} \land \mathsf{AX}_{\mathsf{teaches}}(\neg\mathsf{Course})$$

To give a semantics to query R_4, first replace the solid edge labeled teaches with the dashed edge labeled $\overline{\mathsf{teaches}}$. Then use the same interpretation as for query R_3:

[3] Here, an implicit closed word assumption is made.

$$\mathsf{Teacher} \wedge \mathsf{AX}_{\overline{\mathsf{teaches}}}(\neg\mathsf{Course})$$

Its intuitive meaning is: "it is true if Teacher is linked by edges labeled teaches to all the Course nodes of the graph. Note the extremely compact way for expressing universal quantification.

3.4.2 Admitted Queries

We will show how to encode $\mathbb{W}$-queries in *CTL* formulae. The equivalence result with G-log (see Sect. 3.7.3) and the NP-completeness of the subgraph bisimulation problem ([44]) prevents us to encode all possible queries in a framework that can be solved in polynomial time. We will encode four families of queries $Q = \langle G, \mu \rangle$:

- Q is an acyclic *accessible* query (Sect. 3.4.4).
- G is an acyclic *solid* graph (Sect. 3.4.4).
- G is an acyclic graph and after the application of a rewriting procedure, it becomes acyclic and accessible from $\{\mu\}$ (Sect. 3.4.4).
- G is in one of the four families above but some leaf nodes are replaced by simple solid cycles (Sect. 3.4.5).

3.4.3 Query Translation

As initial step, we associate a formula φ to each node and edge of a graph G. Then we will use this notion for the definition of the formula.

Definition 3.4.2. *Let $G = \langle N, E, \ell \rangle$ be a $\mathbb{W}$-graph.*

- *For all $n \in N$ we define the formula $\varphi(n)$ as:*

$$\varphi(n) \quad = \quad \begin{cases} \ell_{\mathcal{L}}(n) & \text{if } \ell_{\mathcal{L}}(n) \neq \perp \\ \mathsf{true} & \text{otherwise} \end{cases}$$

- *For all edges $e = \langle n_1, \langle c, p \rangle, n_2 \rangle \in E$, the formula $\varphi(e)$ is defined as:*

$$\varphi(e) = \begin{cases} p & \text{if } \ell_{\mathcal{C}}(e) = \ell_{\mathcal{C}}(n_2) \\ \overline{p} & \text{otherwise} \end{cases}$$

3.4.4 Acyclic Graphs

Let us assume that G is an acyclic graph.

Definition 3.4.3. *Let $G = \langle N, E, \ell \rangle$ be an acyclic $\mathbb{W}$-graph, and $\nu \in N$. Then the formula $\Psi_\nu(G)$, depending on both G and ν is defined recursively as follows (cf. Fig. 3.4):*

- *let $b_1, \ldots, b_h$ $(h \geq 0)$ be the successors of ν s.t. $\ell_{\mathcal{C}}(b_i) = solid$,*
- *let $c_1, \ldots, c_k$ $(k \geq 0)$ those s.t. $\ell_{\mathcal{C}}(c_i) = dashed$.*

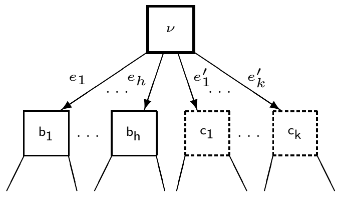


Figure 3.4. Graph for computing $\Psi_\nu(G)$.

– *for $i = 1, \dots, h$ and $j = 1, \dots, k$ let e_i be the edge which links ν to b_i and e'_j the one which links ν to c_j.*

1. *If $\ell_{\mathcal{C}}(\nu) =$ solid, then*

$$\Psi_\nu(G) \quad = \quad \varphi(\nu) \wedge \bigwedge_{i=1\dots h} \mathsf{EX}_{\varphi(e_i)}(\Psi_{b_i}(G)) \wedge \bigwedge_{j=1\dots k} \mathsf{AX}_{\varphi(e'_j)}(\Psi_{c_j}(G))$$

2. *else ($\ell_{\mathcal{C}}(\nu) =$ dashed)*

$$\Psi_\nu(G) = \neg\varphi(\nu) \vee \bigvee_{i=1\dots h} \mathsf{AX}_{\varphi(e_i)}(\Psi_{b_i}(G)) \vee \bigvee_{j=1\dots k} \mathsf{EX}_{\varphi(e'_j)}(\Psi_{c_j}(G))$$

Given a graph G, let $\bar{G}$ be the graph obtained from G by complementation of the colors of edges and nodes (solid becomes dashed and vice versa). It is immediate to prove, by induction on the depth of the subgraph of G that can be reached from a node ν, that $\Psi_\nu(G) = \neg\Psi_\nu(\bar{G})$. Moreover, it is immediate, by the recursive definition of the formula, and by the acyclicity of G, that the formula built is a *CTL* formula (cf. Def. 3.1.2).

Definition 3.4.4. *Let $G = \langle N, E, \ell \rangle$ be an acyclic $\mathbb{W}$-graph, $\nu \in N$, and $Q = \langle G, \nu \rangle$ be an accessible query. The formula associated to Q is $\Psi_\nu(G)$.*

Remark 3.4.2. Observe that:

1. Each node and edge of G is used to build the formula.
2. The size of the formula $\Psi_\nu(G)$ can grow exponentially w.r.t. $|G|$, since the construction of the formula involves the unfolding of a DAG. However, it is only a representation problem: subformulae can be repeated many times. It is an easy task to compute the formula avoiding the repetitions and keeping the memory allocation (and the execution time) linear w.r.t. $|G|$.

The condition on the accessibility of all nodes of G for ν can be weakened, at least for some cases. Consider, for instance, the two goals of Fig. 3.5. If works is the inverse relation of gives salary (i.e. gives salary^{-1}), then one expects the same results from query (a) and query (b). Thus, the idea is

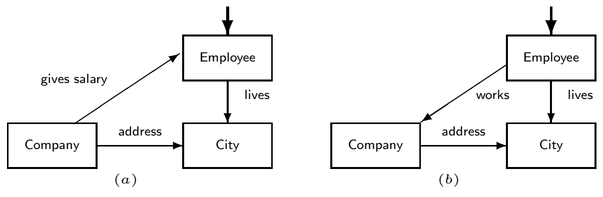


Figure 3.5. Two equivalent acyclic queries

to swap the direction of some edges, replacing the labeling relation with its inverse.[4] This procedure can be automatized:

Algorithm 3.4.1. *Let $G = \langle N, E, \ell \rangle$ be an acyclic solid $\mathbb{W}$-graph and $\nu \in N$.*

1. *Let $\hat{G} = \langle N, \hat{E} \rangle$ be the non-directed graph obtained from G defining:*
$$\hat{E} = \{\{m, n\} : \langle m, \ell, n \rangle \in E\}$$
2. *Identify each connected component of $\hat{G}$ by one of its nodes. Use ν for its connected component. Let $\mu_1, \ldots, \mu_h$ be the other nodes chosen.*
3. *Execute a breadth-first visit of $\hat{G}$ starting from ν, then from nodes $\mu_1, \ldots, \mu_h$.*
4. *Consider the list of nodes $\nu = n_0, n_1, \ldots, n_k$ ordered by the above visit.*
5. *Build the $\mathbb{W}$-graph $\boldsymbol{G} = \langle N, \boldsymbol{E}, \ell \rangle$ from G by swapping the edges that are not consistent with the above ordering:*
$$\begin{array}{rcl} \boldsymbol{E} & = & (E \setminus \{\langle n_a, \langle c, p \rangle, n_b \rangle \in E : b < a\}) \cup \\ & & \{\langle n_b, \langle c, p^{-1} \rangle, n_a \rangle : \langle n_a, \langle c, p \rangle, n_b \rangle \in E \wedge b < a\} \end{array}$$

The above algorithm always produces an acyclic graph, since the edges follow a strict order of the nodes. All the nodes of each connected component of $\hat{G}$ are accessible from the corresponding selected node $(\nu, \mu_1, \ldots, \mu_h)$ by construction (they have been reached by a visit). The Algorithm 3.4.1 can be implemented in time $O(|N| + |E|)$.

Thus, for each node $\nu, \mu_1, \ldots, \mu_h$ we can compute the formulae

$$\Psi_\nu(\boldsymbol{G}), \Psi_{\mu_1}(\boldsymbol{G}), \ldots, \Psi_{\mu_h}(\boldsymbol{G})$$

We recall here the semantics of the formula $\mathsf{EF}_a(\phi)$ we will use in the following definitions (see also Sect. 3.1):

Given a state s, $s \models \mathsf{EF}_a(\phi)$ iff there is a path $\langle \pi_0, \ell_0, \pi_1, \ell_1 \cdots \rangle$, s.t. $\pi_0 = s$ and $\exists j \in \mathbb{N}$ s.t. $\pi_j \models \phi$ and $(\forall i < j)\ \ell_i \in a$.

[4] Recall that in the KTS associated to a $\mathbb{W}$-instance, inverse relations for all the relations involved occur explicitly: the framework is already tuned to deal also with this case.

Definition 3.4.5. *Let $G = \langle N, E, \ell \rangle$ be an acyclic solid $\mathbb{W}$-graph and $\nu \in N$. Let $Q = \langle G, \nu \rangle$ a $\mathbb{W}$-query. The formula associated with Q is:*

$$\Psi_\nu(\boldsymbol{G}) \wedge \mathsf{EF}_{\mathcal{A}ct_G}(\Psi_{\mu_1}(\boldsymbol{G})) \wedge \cdots \wedge \mathsf{EF}_{\mathcal{A}ct_G}(\Psi_{\mu_h}(\boldsymbol{G}))$$

where $\mathcal{A}ct_G$ is as in Def. 3.3.1*.*

Observe that the Algorithm 3.4.1 terminates even if G admits cycles (save self-loops). However, in these cases, the semantics of the query is lost. Cyclic queries require different modal operators (see Subsection 3.4.5).

Let us study another family of acyclic queries that can be handled, via reduction to the accessible query case.

Definition 3.4.6. *Let $G = \langle N, E, \ell \rangle$ be an acyclic $\mathbb{W}$-graph, let $\nu \in N$, $\ell_C(\nu) = solid$, and $Q = \langle G, \nu \rangle$ be a $\mathbb{W}$-query.*

1. *Let $G_s = \langle N_s, E_s, \ell|_{N_s} \rangle$ its solid subgraph (cf. Def.* 3.2.2*).*
2. *Apply the Algorithm* 3.4.1 *to G_s. Let $\nu, \mu_1, \ldots, \mu_h$ be the root nodes chosen.*
3. *Swap in G the same edges that have been swapped by the Algorithm* 3.4.1 *in the step* 2*, obtaining the graph $\boldsymbol{G}$.*
4. *If $\boldsymbol{G}$ is acyclic, then compute the formulae*
$$\Psi_\nu(\boldsymbol{G}), \Psi_{\mu_1}(\boldsymbol{G}), \ldots, \Psi_{\mu_h}(\boldsymbol{G})$$
5. *If all the nodes of G have been visited during the last phase, then the formula associated with Q is:*
$$\Psi_\nu(\boldsymbol{G}) \wedge \mathsf{EF}_{\mathcal{A}ct_G}(\Psi_{\mu_1}(\boldsymbol{G})) \wedge \cdots \wedge \mathsf{EF}_{\mathcal{A}ct_G}(\Psi_{\mu_h}(\boldsymbol{G}))$$

Let us explain why we applicate the algorithm only to G_s. Consider, for example, the query (a) in Fig. 3.6. According to Def. 3.4.6 the formula associated with it is:

$$\mathsf{Teacher} \wedge \mathsf{AX}_{\neg\mathsf{teaches}}(\neg\mathsf{Course}) \wedge \mathsf{EF}_{\mathcal{A}ct_G}(\mathsf{Student} \wedge \mathsf{AX}_{\neg\mathsf{attends}}(\neg\mathsf{Course}))$$

which requires to find all those Teachers who teach all the Courses, if there is somewhere a Student who attends all the Courses.

The Algorithm 3.4.1 should replace the edge labeled attends in the query (a) with an edge labeled $\mathsf{attends}^{-1}$, as depicted in the query (b) of Fig. 3.6. The formula associated with this query is:

$$\mathsf{Teacher} \wedge \mathsf{AX}_{\neg\mathsf{teaches}}(\neg\mathsf{Course} \vee \mathsf{AX}_{\mathsf{attends}^{-1}}\mathsf{Student})$$

The two formulae have different models. The first is closer to the interpretation of graph-based formulae in other frameworks (e.g. G-Log).

3.4.5 Cyclic Queries

In this section we extend the technique assigning a temporal formula to queries admitting cycles. We make use of the Generally operator, used as $\mathsf{EG}_a(\phi)$, whose semantics is (see also Section 3.1):

> Given a state s, $s \models \mathsf{EG}_a(\phi)$ iff there is a path $\pi = \langle \pi_0, \ell_0, \pi_1, \ell_1 \ldots \rangle$, s.t. $\pi_0 = s$ and $\forall j \in \mathbb{N}$ ($\pi_j \models \phi$ and $\ell_j \in a$).

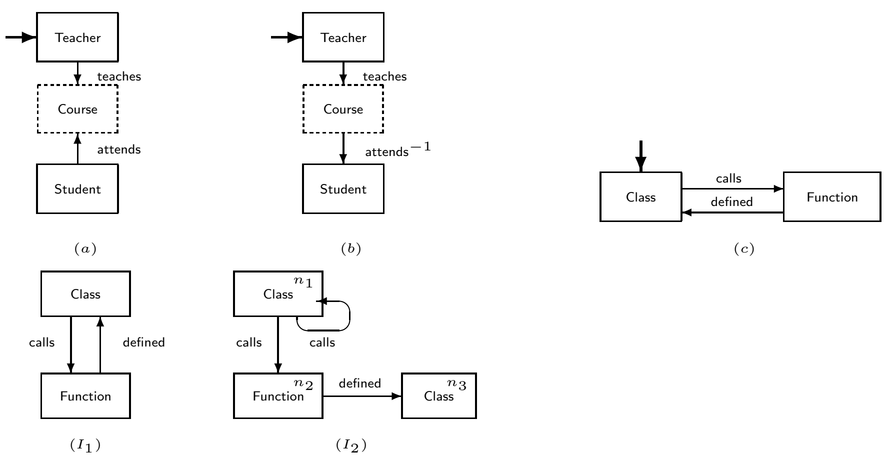


Figure 3.6. $\mathbb{W}$-queries and $\mathbb{W}$-instances

Consider the query (c) in Fig. 3.6. It requires to collect *all the classes which call a function defined inside themselves.* This property can be expressed by using the much richer temporal logic *CTL**, which extends *CTL* by allowing basic temporal operators where the path quantifier (A or E) is followed by an arbitrary linear-time formula, allowing boolean combinations and nesting, over F, G, X, and U [49]. Precisely, given a set $\mathcal{A}ct$ of actions and a set Π of atomic propositions, the set *Lit* of *literals* is defined as $Lit = \Pi \cup \{\neg q \mid q \in \Pi\} \cup \{\mathsf{true}, \mathsf{false}\}$. *State formulae* ϕ and *Path formulae* ψ are inductively defined by the following grammar, where $p \in Lit$ and $a \subseteq \mathcal{A}ct$:

$$\begin{array}{lcl} \text{state formulae:} & \phi & ::= & p \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathsf{A}\psi \mid \mathsf{E}\psi \\ \text{path formulae:} & \psi & ::= & \phi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathsf{G}_a\psi \mid \mathsf{F}_a\psi \mid \mathsf{X}_a\psi \mid \\ & & & \mathsf{U}_a(\psi, \psi) \end{array}$$

The *CTL** formula for the query (c) in Fig. 3.6 is:

$$\begin{array}{rcl} \Psi_{(c)} & = & \mathsf{Class} \wedge \mathsf{EX}_{\mathsf{calls}}(\mathsf{Function} \wedge \mathsf{EX}_{\mathsf{defined}}(\mathsf{Class})) \wedge \\ & & \mathsf{EG}_{\{\mathsf{calls},\mathsf{defined}\}}((\mathsf{Class} \vee \mathsf{Function}) \wedge \mathsf{Class} \rightarrow \mathsf{X}_{\mathsf{calls}}(\mathsf{Function}) \wedge \\ & & \qquad \mathsf{Function} \rightarrow \mathsf{X}_{\mathsf{defined}}(\mathsf{Class})) \end{array}$$

Observe that the modal operator X is used without any path quantifier. This is allowed in *CTL** but not in *CTL*. In particular, a path $\pi = \langle \pi_0, \ell_0, \pi_1, \ell_1 \ldots \rangle$ satisfies the formula $\mathsf{X}_a \psi$ when $\ell_0 \in a$ and ψ holds starting at the second state π_1 on the path.

The first part of the formula $\Psi_{(c)}$ above is aimed at identifying a path of length greater or equal that two, the Generally operator imposes to retrieve only cyclic paths where nodes labeled Class alternate with nodes labeled $\mathsf{Function}$.

With the *CTL* logic it is only possible to approximate the translation of this kind of cyclic queries. In fact, a *CTL* formula for the query (c) in Fig. 3.6 could be:

$$\begin{aligned}\Psi'_{(c)} \quad = \quad & \mathsf{Class} \wedge \mathsf{EX}_{\mathsf{calls}}(\mathsf{Function} \wedge \mathsf{EX}_{\mathsf{defined}}(\mathsf{Class})) \wedge \\ & \mathsf{EG}_{\{\mathsf{calls},\mathsf{defined}\}}(\mathsf{Class} \rightarrow \mathsf{EX}_{\mathsf{calls}}(\mathsf{Function} \wedge \mathsf{EX}_{\mathsf{defined}}(\mathsf{Class})))\end{aligned}$$

The node Class of instance (I_1) in Fig. 3.6 satisfies both the formulae $\Psi_{(c)}$ and $\Psi'_{(c)}$. Instead, the node n_1 of instance (I_2) in Fig. 3.6 satisfies $\Psi'_{(c)}$ but not $\Psi_{(c)}$. Thus, the *CTL* translation gives only an approximation of the *CTL** one.

Since the model-checking problem for *CTL** is PSPACE complete, we accept this loosing of precision, and we assign a *CTL* formula also to a generic (pointed) cycle.

Definition 3.4.7. *The formula $\Psi_{\nu_1}(G)$ associated to a red cyclic graph*

$$G = \langle \{\nu_1, \ldots, \nu_n\}, \{\langle \nu_1, \ell_1, \nu_2 \rangle, \langle \nu_2, \ell_2, \nu_3 \rangle, \ldots, \langle \nu_n, \ell_n, \nu_1 \rangle\}, \ell, \nu_1 \rangle,$$

is defined as:

$$\Psi_{\nu_1}(G) = \psi_C \wedge \mathsf{EG}_{\{\ell_1,\ldots,\ell_n\}}(\varphi(\nu_1) \rightarrow \psi_C)$$

where $\varphi(\nu_1)$ is as in Def. 3.4.2, *and ψ_C is the CTL formula $\Psi_{\nu_1}(C)$ associated to the DAG*

$$C = \langle \{\nu_1, \ldots, \nu_{n+1}\}, \{\langle \nu_1, \ell_1, \nu_2 \rangle, \langle \nu_2, \ell_2, \nu_3 \rangle, \ldots, \langle \nu_n, \ell_n, \nu_{n+1} \rangle\}\rangle,$$

rooted at ν_1 and where ν_{n+1} is a "copy" of ν_1, i.e. a new node s.t. $\ell(\nu_{n+1}) = \ell(\nu_1)$.

The formula ψ_C is used twice for the sake of simplicity. In its second usage it could be shortened, e.g. by removing the formula of the first node.

Note that a special case of generic cyclic queries are simple cycle queries (e.g. queries shaped as rings) composed by nodes with a unique label ℓ_n and edges all labeled ℓ_e. This kind of cyclic 𝕎-queries can be exactly translated into *CTL*.

Consider for example, the cyclic query (a) in Fig. 3.7, it requires to collect *all the functions which are recursive.* Its behavior could be described by:

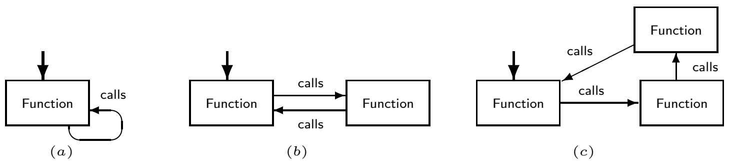


Figure 3.7. Three bisimilar simple cyclic queries

$$\mathsf{Function} \wedge \mathsf{EX}_{\mathsf{calls}}(\mathsf{Function} \wedge \mathsf{EX}_{\mathsf{calls}}(\mathsf{Function} \wedge \mathsf{EX}_{\mathsf{calls}}(\cdots)))$$

that can be mimicked exactly by

$$\mathsf{EG}_{\mathsf{calls}}\mathsf{Function} \tag{3.1}$$

It is easy to see that this formula (more compact than the formula in Def. 3.4.7) expresses the cyclic condition depicted in the $\mathbb{W}$-query (a) in Fig. 3.7. In fact, if we apply the procedure in Def. 3.4.7 to the $\mathbb{W}$-query (a) we obtain the formula

$$\mathsf{Function} \wedge \mathsf{EX}_{\mathsf{calls}}\mathsf{Function} \wedge \mathsf{EG}_{\mathsf{calls}}(\mathsf{Function} \rightarrow \mathsf{EX}_{\mathsf{calls}}\mathsf{Function})$$

which has the same models of $\mathsf{EG}_{\mathsf{calls}}\mathsf{Function}$.

It is easy also to see that queries (a) and (b) (and also (c)) can be considered equivalent, their models satisfy the formula $\mathsf{EG}_{\mathsf{calls}}\mathsf{Function}$. This fact has a graph counterpart: their pointed graphs are equivalent modulo bisimulation (for this topic, see Section 3.7.3). (a) is the minimum graph bisimulation equivalent (from [5] we know that there is always a unique minimum graph, called bisimulation contraction).

To sum up, consider the $\mathbb{W}$-query (a) in Fig. 3.8. In order to find its *CTL* encoding we have to apply the rewriting procedure to swap the in edge, thus obtaining the $\mathbb{W}$-query (b) and the formula:

$$\begin{aligned}\psi_\nu \quad = \quad & \mathsf{Class} \wedge \mathsf{EX}_{\overline{\mathsf{isa}}}(\mathsf{Class} \wedge \mathsf{EX}_{\mathsf{name}}\mathsf{Math}) \wedge \mathsf{EX}_{\mathsf{in}^{-1}}(\mathsf{Procedure} \wedge \\ & \mathsf{EX}_{\mathsf{calls}}(\mathsf{Fuction} \wedge \mathsf{EG}_{\mathsf{calls}}\mathsf{Function}))\end{aligned}$$

Note that the *CTL* formula associated to the simple cyclic query rooted at n is $\mathsf{EG}_{\mathsf{calls}}\mathsf{Function}$.

3.5 Complexity Issues

Let us state the main computational result of our approach:

Theorem 3.5.1. *Let $\langle G, \nu\rangle$ be a $\mathbb{W}$-query in one of the forms described in Section* 3.4.2. *Querying a $\mathbb{W}$-instance I with G is done in linear running time on $|\mathcal{K}_I|$ and $|\Psi_\nu(G)|$.*

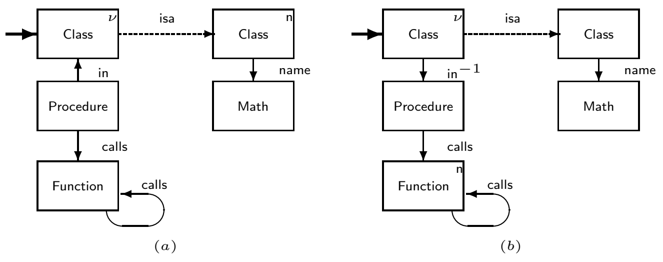


Figure 3.8. Two 𝕎-queries

The proof of the theorem follows from [27].

When computing $\mathcal{K}_I$, a quadratic time and space complexity is introduced as negative relations are computed and stored.

As far as the size of the formula is concerned, as discussed in Remark 3.4.2, even if $|\Psi_\nu(G)|$ can grow exponentially with $|G|$, it is natural to represent it using a linear amount of memory. This compact representation is allowed by the model-checker NuSMV.

As shown in Sect. 3.4.5 for cyclic queries, it seems to be impossible to map all the queries in *CTL* formulae. This fact can be justified algorithmically. The semantics of acyclic 𝕎-queries without negation can be proved to be equivalent to that of G-Log queries without negation (Theorem 3.7.2). If we extended this equivalence result to cyclic queries (even without negation), we would provide an implementation of a subset of G-log in which data retrieval is equivalent to the subgraph bisimulation problem, proved to be NP complete in [44]. This would be in contradiction with Theorem 3.5.1.

3.6 Implementation of the Method

We have effectively tested the data retrieval technique presented in the previous sections by using the model checker NuSMV [26], which is designed to check the satisfaction of specifications expressed in *CTL* (or *LTL*) on Kripke Structures. To this aim, KTS related to 𝕎-instances have been rewritten into Kripke Structures, where edges are not labeled (see Remark 3.1.1). The idea of the rewriting rule is expressed by the following:

Algorithm 3.6.1. *Given a KTS, replace every labeled edge*

$$m \xrightarrow{label} n$$

by the two edges

$$m \longrightarrow \mu, \mu \longrightarrow n,$$

where μ is a new node labeled label.

Inverse edges are not added by this encoding since the modal operator *before* is not considered in this work and, hence, not generated by the formula translation.

We have tested the three queries of Fig. 3.11 on the $\mathbb{W}$-graph of Fig. 3.10. We have coded the $\mathbb{W}$-instance into a KS. The space of states of the KS is determined by the declaration of the state variables (in the above example `s`). `s` is a scalar variable, which can take the symbolic value `n1`, ..., `n11` (the node identifiers in the instance) or `e1`, ..., `e12` (the identifiers of the nodes added in order to replace the labeled edges). The transition relation of the KS is expressed by assigning, for each value of the variable `s`, the list of nodes that can be reached from it using one edge. The variable ℓ is introduced to define the label of each state, identified by the value of the variable `s`.

```
MODULE main
VAR
s   : {n1, ..., n11, e1, ..., e12};
lab : {person, company, city,
      works, owns, address, lives};
ASSIGN
   init(s) := {n1, n5, n6, n11};
   next(s) := case
      s = n1 : e1; s = n2 : {e2, e3};
      s = n3 : {e3, e12}; s = n4 : n4; s = n5 : {e4, e5};
      s = n6 : {e6, e7}; s = n7 : e8; s = n8 : n8;
      s = n9 : n9; s = n10 : {e10, e11}; s = n11 : e9;
      s = e1 : n2; s = e2 : n3; s = e3 : n4; s = e4 : n3;
      s = e5 : n4; s = e6 : n4; s = e7 : n7;
      s = e8 : n8; s = e9 : n8; s = e10 : n7;
      s = e11 : n9; s = e12 : n4;
   esac;
DEFINE
   ℓ := case
      s = n1 : person; s = n2 : company;
      s = n3 : company; s = n4 : city;
      s = n5 : person; s = n6 : person;
      s = n7 : company; s = n8 : city; s = n9 : city;
      s = n10 : company; s = n11 : person;
      s = e1 : works; s = e2 : owns; s = e3 : address;
      s = e4 : works; s = e5 : lives; s = e6 : lives;
      s = e7 : works; s = e8 : address; s = e9 : lives;
      s = e10 : owns; s = e11 : address;
      s = e12 : address;
   esac;
```

Figure 3.9. NuSMV format of a $\mathbb{W}$-instance

The translation of the queries in the NuSMV format is:

```
SPEC
    ℓ = person & EX(ℓ = works & EX(ℓ = company))
SPEC
    ℓ = person & EX(ℓ = lives & EX(ℓ = city))
     & AX(ℓ = works -> ! EX(ℓ = company))
SPEC
    ℓ = company & EX(ℓ = owns & EX(ℓ = company
     & EX(ℓ = address & EX(ℓ = city))))
     & EX(ℓ = address & EX(ℓ = city))
```

(Q_1) requires all the Persons who work in a Company and (Q_2) all the Persons who live in a City and do not work in any Company. Query Q_3 is not a tree: it requires to collect all the nodes Company which own another Company. Both companies must have address in the (same) City (actually, the bisimulation based semantics can only force that they have address in *some* city. For requiring the *same* city, further information, e.g. the name of the city, is needed).

Note that in the *CTL* translation of query (Q_2) we do not use negated edges, in fact the required property can be formalized by using a universal quantification (e.g. if a Person works somewhere we require that the place is not a Company). However, also in the NuSMV implementation negated edges are required and have to be stored to correctly encode queries with negated edges between positive nodes. Consider, for example, query (R_2) of Fig. 3.3, in order to find all the pairs of nodes Teacher and Course that are not connected by an edge teaches, we have to add between each pair of nodes without a teaches relation, a *service* edge representing the fact that that relation does not hold. Note that we choose to label the new relation ¬teaches. In fact model-checkers use only paths to traverse a Transition System and thus to find pairs (or tuple) of nodes connected with a certain arc (i.e. a transition).

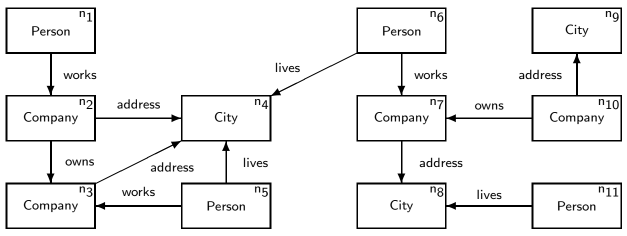


Figure 3.10. Sample Concrete Graph

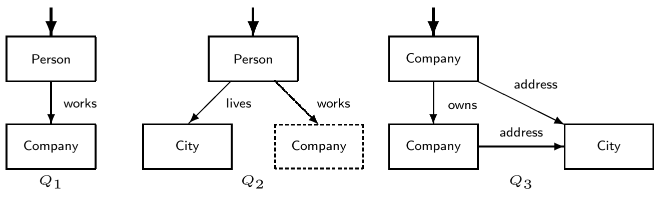


Figure 3.11. Sample Queries

NuSMV is a procedure that decides whether a given KS M is a model of a *CTL* formula Ψ, i.e. whether M satisfies Ψ, and returns as main output only `true` or `false`. In our application we are interested in finding out the set of states in M that satisfy Ψ. To this purpose we have extended the NuSMV code in order to solve the global model-checking problem, i.e. to print all the states of a given KS that satisfy Ψ.

Intuitively, given a Kripke Structure $\mathcal{K} = \langle \Sigma, \mathcal{R}, I \rangle$ and a *CTL* formula φ, we can say that the extension of the model-checker NuSMV is an algorithm working in the following way:

1. we apply the procedure *label-graph*(φ) on $\mathcal{K}$ (see Sect. 3.1.2);
2. for each $s \in \Sigma$ such that *labeled*(s, φ), the output of the model model-checker is “The state s satisfies the property φ”. If $\forall s \in \Sigma$ it holds $\neg$*labeled*(s, φ), we conclude that “There are no states that satisfy φ”.

For the examples in Figure 3.11, we have obtained the answers:

```
The states which satisfy the formula are:
   State0 :  l = person   s = n6
   State1 :  l = person   s = n5
   State2 :  l = person   s = n1
The states which satisfy the formula are:
   State0 :  l = person   s = n11
The states which satisfy the formula are:
   State0 :  l = company   s = n10
   State1 :  l = company   s = n2
Runtime Statistics:
   Machine name  :  xyz.sci.univr.it
   User time     :  0.030 seconds
   System time   :  0.050 seconds
```

Moreover, we have effectively tested on real-life size examples that our extension of the NuSMV system runs in linear time in both the sizes of the KS and Ψ.

3.7 Applications to Existing Languages

In this section we apply the model-checking based data retrieval approach to some existing query languages for semistructured data, such as UnQL, GraphLog, and G-Log.

3.7.1 UnQL

We will show that UnQL databases can be immediately mapped to $\mathbb{W}$-instances (see Section 2.9.1 for a brief description of the language). Then we will show how to encode UnQL queries into modal formulae.

Algorithm 3.7.1. *Let $G = \langle\langle N, E, \ell\rangle, \nu\rangle$ be an edge-labeled (UnQL) graph rooted at ν with name DB. Assume, w.l.o.g., that $DB \notin \ell(E)$. We define the $\mathbb{W}$-graph $G' = \langle N', E', \ell'\rangle$ rooted at ν as:*

1. *$E' = E$ is the set of edges;*
2. *$N' = N$ is the set of nodes;*
3. *$\ell'(\nu) = DB$;*
4. *$\ell'(m) = \perp$ if $m \in N \setminus \{\nu\}$.*

G' is rooted since G is rooted. Thus, we can use the root node (labeled DB) to identify the instance of a database.

Remark 3.7.1. UnQL databases can also be translated into Kripke Structures (see Remark 3.1.1). In fact, the label of each edge $\langle n, p, m\rangle$ can be associated to the arrival node m.

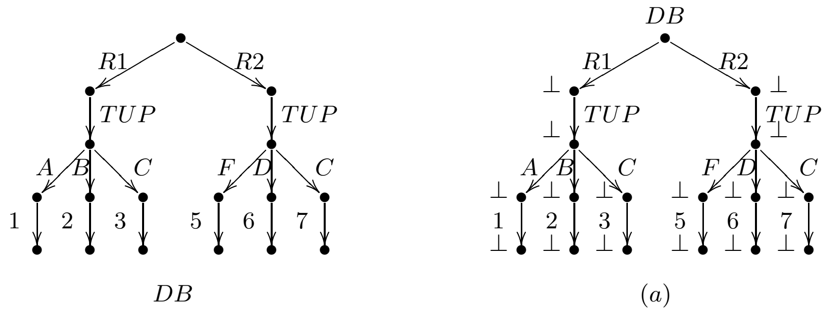


Figure 3.12. An UnQL graph, called DB, and its translation

For example, the UnQL graph DB in Fig. 3.12 is rewritten as the $\mathbb{W}$-graph (a) by labeling the set of nodes. The graph (a) can be seen as a Kripke Transition System $\mathcal{K} = \langle \Sigma, Act, \mathcal{R}, I\rangle$, where Σ is the set of bullet nodes,

$Act = \{R1, R1^{-1}, \overline{R1}, \ldots\}$, the transition relation $\mathcal{R}$ is the set of edges, and the interpretation function $I : \Sigma \to \wp(\{\bot, number, \mathsf{true}, \mathsf{false}\})$ is defined as follows:

$$I(s) \quad = \quad \begin{cases} \{\ell(s), \mathtt{true}\} & \text{if } s \text{ is not a leaf} \\ \{\ell(s), number, \mathtt{true}\} & \text{otherwise} \end{cases}$$

The atom *number* is used to model the fact that in some queries we need to know if a state of the Kripke structure contains a numeric value (i.e. its label is a number).

Let us consider the examples of UnQL query introduced in Section 2.9.1 on the DB database in Fig. 3.12 and their translations in temporal formulae. We will use the modal operator B (Before) (not included in the logic *CTL*), whose meaning is the following:

> Given a state s and a temporal formula ϕ, $s \models \mathsf{AB}_a\phi$ iff for all paths $\pi = \langle \ldots, x, \ell, s, \_, \pi_1, \ldots \rangle$, it holds that $x \models \psi$ and $\ell \in a$, whereas $s \models \mathsf{EB}_a\psi$ iff there is a path $\pi = \langle \ldots, x, \ell, s, \_, \pi_1, \ldots \rangle$ such that $x \models \psi$ and $\ell \in a$.

Note that B does not belong to *CTL* (cf. Definition 3.1.2), but inverse edges we introduce in Kripke Transition Systems allow to replace this operator with the *CTL* operator NeXt (X).

The expression

$$\textbf{select } t \textbf{ where } R1 \Rightarrow \backslash t \leftarrow DB \tag{3.2}$$

computes *the union of all trees t such that DB contains an edge $R1 \Rightarrow t$ emanating from the root.* The same concept can be expressed by the temporal formula

$$\varphi_{(1)} = \bot \wedge \mathsf{EB}_{R1}(DB)$$

which is true in each dummy (depicted as bullet) state which has as predecessor the root node DB and the edge between them is labeled $R1$, or equivalently, by the *CTL* formula

$$\psi_{(1)} = \bot \wedge \mathsf{EX}_{R1^{-1}}(DB).$$

The UnQL expression

$$\textbf{select } t \textbf{ where } \backslash l \Rightarrow \backslash t \leftarrow DB \tag{3.3}$$

retrieves *pointed nodes of any edge emanating from the root* and is translated by the formula

$$\varphi_{(2)} = \bot \wedge \mathsf{EB}_{Act}(DB)$$

which is true in all the bullet states accessible by means of a unique edge from the root named DB. The same UnQL query can be translated in *CTL* by means of the formula

$$\psi_{(2)} = \perp \wedge \mathsf{EX}_{Act}(DB)$$

An UnQL queries that look arbitrarily deep into the database to find *pointed nodes of all edges with a numeric label* is the following:

$$\textbf{select}\ \{l\}\ \textbf{where}\ \_* \Rightarrow \backslash l \Rightarrow \_ \leftarrow DB, isnumber(l) \tag{3.4}$$

The expression (3) can be easily translated into the *CTL* formula

$$\psi_{(3)} = \perp \wedge\, number$$

which is true in the leaf nodes.

UnQL can look arbitrarily deep into a database; this feature has an immediate translation into *CTL* thanks to the F operator. For example, given the $\mathbb{W}$-instances of Fig. 3.2, the *CTL* query

$$\text{Teacher} \wedge \mathsf{EF}_{\mathcal{A}ct_G}\text{Databases}$$

requires to find Teacher nodes having a path that eventually reaches a Databases node. It would be interesting to include this feature also into the query language $\mathbb{W}$; one possibility is to use dotted edges to represent paths of an arbitrary length.

The UnQL query

$$\begin{array}{rl} \textbf{select} & \{Tup \Rightarrow \{A \Rightarrow x, D \Rightarrow z\}\} \\ \textbf{where} & R1 \Rightarrow Tup \Rightarrow \{A \Rightarrow \backslash x, C \Rightarrow \backslash y\} \leftarrow DB, \\ & R2 \Rightarrow Tup \Rightarrow \{C \Rightarrow \backslash y, D \Rightarrow \backslash z\} \leftarrow DB \end{array} \tag{3.5}$$

computes *the join of $R1$ and $R2$ on their common attribute C and the projection onto A and D*. It is easy to check that its application to the database DB has an empty result.

UnQL queries expressing a join condition on a graph do not have a corresponding temporal formula, because in propositional temporal logics it is possible to identify a state only by using the set of atomic predicates (i.e. local properties) that are true in the state. Thus, nodes with the same properties cannot be distinguished.

An intuitive translation for the former query could be the following:

$$\begin{array}{rl} \varphi_{(4)} = & \perp \wedge \mathsf{EB}_{Act}(\perp \wedge \mathsf{EB}_{A}(\perp \wedge \mathsf{EX}_{C}(\perp \wedge \mathsf{EX}_{Act}(\perp))))\vee \\ & \perp \wedge \mathsf{EB}_{Act}(\perp \wedge \mathsf{EB}_{D}(\perp \wedge \mathsf{EX}_{C}(\perp \wedge \mathsf{EX}_{Act}(\perp)))) \end{array}$$

The formula φ holds in two states of the $\mathbb{W}$-graph (b) in Fig. 3.12 (the bullet nodes with unique ingoing edge labeled 1 and 6 respectively), and thus, the result of its application on the database DB is not correct.

Although we have not developed an algorithm to translate UnQL queries into CTL formulae, we can conclude that our approach allows to correctly deal with join-free queries of UnQL.

3.7.2 GraphLog

GraphLog is a query language based on a graph representation of both data and queries [33]. Queries represent graph patterns corresponding to paths in databases (see also Section 2.9.2).

For example, the graph I in Fig. 3.13 is a representation of a flights schedule database. Each flight number is related to the cities it connects (by the predicates *from* and *to*, respectively) and to the departure and arrival time (by the predicates *departure* and *arrival*).

GraphLog represents databases by means of *directed labeled multigraphs* which precisely correspond to $\mathbb{W}$-instances. GraphLog databases are not required to be rooted and therefore the corresponding $\mathbb{W}$-graphs are not necessarily rooted.

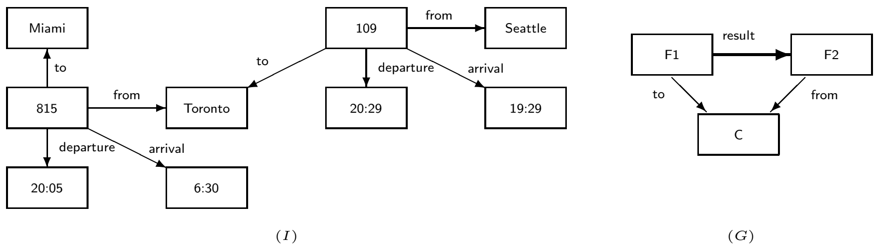


Figure 3.13. GraphLog representation of a database and a query

In order to apply model-checking techniques also to this language, we show how to encode GraphLog queries into *CTL* formulae. For example, graph (G) in Fig. 3.13 requires to find in a database two flights $F1$ and $F2$ departing from and arriving to the same city C, respectively. The edge 'result' will connect $F1$ and $F2$. The *CTL* formula associated with the non-costructive part of this query (i.e. the query without the 'result' edge) is:

$$\Psi(G) \quad = \quad \mathsf{Flight} \wedge \mathsf{EX}_{\mathsf{to}}(\mathsf{City} \wedge \mathsf{EX}_{\mathsf{from}^{-1}}\mathsf{Flight})$$

(which intuitively describes the node $F1$). We remark here on the fact that by inverting the from edge we avoid to represent the join condition that is present in the query. $\Psi(G)$ is clearly satisfied by the state labeled 109. Note that the Kripke Transition System associated to the instance (I) in Fig. 3.13 could be as follows: [5]

[5] In the interpretation function, for each state we add some local properties (i.e. the set of atomic properties true in each state is not only composed by the label of the state).

- $\Pi_G = \{$ Flight, City, ArrivalTime, DepartureTime, 109, 815, ...$\}$
- $\Sigma_G = \{n_1, \ldots, n_9\}$
- The interpretation $\mathcal{I}$ is:

$$\begin{array}{rcl}
\mathcal{I}(n_1) & = & \{\mathsf{true}, \mathrm{City}, \mathrm{Miami}\} \\
\mathcal{I}(n_2) & = & \{\mathsf{true}, \mathrm{Flight}, 815\} \\
\vdots & \vdots & \vdots \\
\mathcal{I}(n_8) & = & \{\mathsf{true}, \mathrm{City}, \mathrm{Seattle}\} \\
\mathcal{I}(n_9) & = & \{\mathsf{true}, \mathrm{ArrivalTime}, 19:29\}
\end{array}$$

The possibility to express closure literals causes some difficulties in the translation of query graphs into *CTL* formulae. Consider for example, the two query graphs in Fig. 3.14 representing the relationships 'parent' and 'relative' between people. Their natural translation into *CTL* should be:

$$\begin{array}{rcl}
\Psi(Q_1) & = & \mathsf{Person} \land ((\mathsf{EX}_{\mathsf{parent}}\mathsf{Person}) \rightarrow \\
& & (\mathsf{Person} \land \mathsf{EX}_{\mathsf{parent}}(\mathsf{Person} \land \mathsf{EX}_{\mathsf{relative}^{-1}}\mathsf{Person}))) \\
\Psi(Q_2) & = & \mathsf{Person} \land ((\mathsf{EX}_{\mathsf{relative}}(\mathsf{Person} \land \mathsf{EX}_{\mathsf{relative}}\mathsf{Person})) \rightarrow \\
& & (\mathsf{Person} \land \mathsf{EX}_{\mathsf{relative}}(\mathsf{Person} \land \mathsf{EX}_{\mathsf{relative}}(\mathsf{Person} \land \\
& & \mathsf{EX}_{\mathsf{relative}^{-1}}\mathsf{Person}))))
\end{array}$$

It is easy to see that the node n of the instance graph $\mathcal{I}$ in Fig. 3.14 satisfies the formula $\Psi(Q_1)$ but does not fulfill the natural interpretation of relationships between people. Once again, the problem is that in *CTL* it is not possible to constrain the last unary predicate Person in $\Psi(Q_1)$ to be satisfied by the same state of the first unary predicate Person.

In order to overcome this limitation and to be able to apply model-checking techniques to GraphLog, one should first compute the closure of labeled graphs representing databases. Intuitively, computing graph-theoretical closure entails inserting a new edge labeled a between two nodes, if they are connected via a path of any length composed of edges all labeled a in the original graph (see [42] for an application of graph closure to XML documents). Computing the closure is well-known to be a polynomial time problem w.r.t. the size of nodes of the graph.

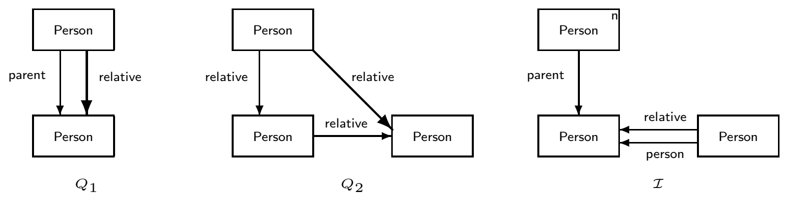


Figure 3.14. Two query graphs and an instance

Again, we can state that our approach based on model-checking techniques allows to correctly deal with join-free queries of GraphLog without closure operators.

The problem of expressing join conditions by means of *CTL* formulae remains open, because in propositional modal logics framework there is no way to force that two distinct states with the same set of local properties are really the same state. In fact, the propositional modal logic semantics forces a bisimulation equivalence between subgraphs fulfilling the same formula. This allows us to say that two nodes with the same properties are equivalent but not to require that they are the same node and thus, this is the reason why we cannot model correctly the *join* operation.

A possibility to solve this problem is to add to the set of local properties of each state a sort of identifier of the state: in this way, by supposing the set of all states to be finite (actually, this restriction is true in the database context) one could express join conditions. For example, the $\mathbb{W}$-query in Fig. 3.15 requires to find all the Students attending a Course together with at least one friend. If we know that the set of identifiers of Course nodes is $\{c_1, c_2\}$, we could translate the query with the *CTL* formula

$$\left(\begin{array}{l}(\mathsf{Student} \wedge \mathsf{EX}_{\mathsf{attends}}(\mathsf{Course} \wedge c_1) \wedge \\ \mathsf{EX}_{\mathsf{friend}}(\mathsf{Student} \wedge \mathsf{EX}_{\mathsf{attends}}(\mathsf{Course} \wedge c_1))) \vee \\ (\mathsf{Student} \wedge \mathsf{EX}_{\mathsf{attends}}(\mathsf{Course} \wedge c_2) \wedge \\ \mathsf{EX}_{\mathsf{friend}}(\mathsf{Student} \wedge \mathsf{EX}_{\mathsf{attends}}(\mathsf{Course} \wedge c_2)))\end{array}\right)$$

However, this translation causes an explosion of the formula size and it requires also to know in advance the set of identifiers of a certain state (in the example the Course state).

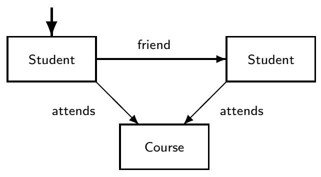


Figure 3.15. A $\mathbb{W}$-query with a join condition

3.7.3 G-Log

G-Log [81] as GraphLog, is a query language in which data, queries, and generic rules are expressed by graphs (see Section 2.1). As far as instances are concerned, syntax of G-Log instances is the same as that of $\mathbb{W}$. G-Log queries allow more flexibility than $\mathbb{W}$-ones. Nevertheless, often G-Log queries have only one "green" node with an edge pointing to a "red" node ν. These queries are the same as $\mathbb{W}$-queries, in which ν is the node pointed by the query. In G-Log two kinds of nodes are allowed, complex and atomic nodes, but this feature can be simulated by modifying the co-domain of the *label* function in $\mathbb{W}$. Instead, solid edges are forbidden to enter into dashed nodes in G-Log. This feature of $\mathbb{W}$ is one of the main points for programming nesting quantifications and for expressing in a compact way the universal quantification. Only existential requirements are instead allowed in G-Log by using a unique query and in order to obtain universal requirements one should compose a G-Log program (i.e. a combination of rules). If a $\mathbb{W}$ query fulfills this further requirement, is a G-Log query.

Semantics of query application is given in [81] via the notion of *graph embedding* and in this work (see also [37]) by using the notion of *subgraph bisimulation* (both NP complete, also without considering negation in query graphs). We prove here that all the G-Log queries that belong to the four families of graphs defined in Section 3.4.4 can be correctly solved using the model-checking approach. We have already discussed (see Section 3.5) that for complexity limits we cannot extend the method to all cyclic queries.

Theorem 3.7.2 (Correctness I). *Let I be a $\mathbb{W}$-instance and $\langle G, \nu \rangle$ be a $\mathbb{W}$-query without negation and s.t. $G|_{\{solid\}}$ is a red DAG rooted at ν. Then, a subgraph I' of I matches with G, and $m_1, \dots, m_z$ are in relation with ν if and only if all corresponding nodes $m'_1, \dots, m'_z$ in $\mathcal{K}_I$ are s.t. $m'_i \models \Psi_\nu(G)$.*

Proof. Assume that $I' \sim G|_{\{RS\}}$ and that b is a bisimulation. Since $G|_{\{RS\}}$ is rooted by the node ν, this node will be in relation b with some nodes $m_1, \dots, m_z$ of I'. For each $n \in N$ let *depth*(n) be the maximum length of a path from n to a leaf. We prove the property by induction on *depth*(ν).

If *depth*$(\nu) = 0$, the property is trivially true. There are z nodes in I that satisfy the requirements of the query. They are exactly the nodes in $\mathcal{K}_I$ in which the formulae are true. The same situation holds between the formula $\Psi_\nu(G)$ and a state of $\mathcal{K}_I$.

Let *depth*$(\nu) > 0$ and assume that the property holds for the subgraphs. We are in the situation of Figure 3.4 with $k = 0$, because $G|_{\{\text{dashed}\}} = \emptyset$ and thus,

$$\Psi_\nu(G) = \varphi(\nu) \wedge \bigwedge_{i=1\dots h} \mathsf{EX}_{\varphi(b_i)}(\Psi_\nu(G))$$

For one direction, assume I' matches with G, and $m_1, \dots, m_z$ are root nodes of I'. Of course they are bisimilar to ν. By definition of bisimulation, this means that $\ell_{\mathcal{L}}(\nu) = \ell_{\mathcal{L}}(m_i)$ for all $i = 1, \dots, z$. Thus, the formula $\varphi(\nu)$, namely $\ell_{\mathcal{L}}(\nu)$, is true in the states $m'_1, \dots, m'_z$ of the transition system $\mathcal{K}_I$. Moreover, by definition of bisimulation, it holds that for any edge $\langle \nu, e_i, b_i \rangle$ outgoing from ν and for each m_i there is in I (at least) one edge outgoing from m_i with the same label e_i and the node ξ_i reached is bisimilar to b_i (and vice versa, for each node m_i, ... The analysis of this part is analogous).

Bisimulation and the acyclicity of G implies acyclicity of I': we can apply the inductive hypothesis to the various pairs of subgraphs of G and of I' identified in this way ($\beta_1, \ldots, \beta_h$ are the subgraphs of $G|_{\{RS\}}$ rooted by $b_1, \ldots, b_k$ as in Figure 3.4). Thus, by inductive hypothesis, we have that there are nodes ξ_is such that $\xi_i \models \Psi_{b_i}(G)$ and that the node (state) ξ_i in $\mathcal{K}_I$ is in relation e_i with the node m'_i corresponding to the node m_i bisimilar to ν. Thus, each formula $\mathsf{EX}_{\varphi(e_i)}(\Psi_{b_i}(G))$ is fulfilled by m'_i, and also is the conjunction. The proof of the other direction is similar.

Theorem 3.7.3 (Correctness II). *Let $\langle G, \nu \rangle$ be a $\mathbb{W}$-queries consisting w.l.g. in a simple minimal cycle with a unique node ν and a unique edge e, and $I = \langle N, E, \ell \rangle$ be a $\mathbb{W}$-instance, and $\mathcal{K}_I = \langle \Sigma_G, \mathcal{A}ct_G, \mathcal{R}_G, \mathcal{I}_G \rangle$ the corresponding KTS. Then, a node $n \in N$ is s.t. $n \models \psi_\nu(G)$ if and only if there is a relation b s.t. n is the root of a subgraph I' of I, $G \overset{b}{\sim} I'$, and $n \overset{b}{\sim} \nu$.*

Proof. $\psi_\nu(G) = \mathsf{EG}_{\ell(e)}\ell(\nu)$. We suppose that $n_1 \in N$ is such that $n_1 \models \psi_\nu(G)$ and we construct the relation b. $n_1 \models \psi_\nu(G)$ iff $\ell(n_1) = \ell(\nu)$ (we let $b(n_1, \nu)$) and $\exists \langle n_1, e_1, n_2 \rangle \in E$ s.t. $\ell(n_2) = \ell(\nu)$ and $\ell(e_1) = \ell(e)$ (because of the semantics of the Generally operator) and this happens in the following two cases:

1. $n_2 = n_1$. It is easy to check that b is an isomorphism, and thus a bisimulation.
2. $n_2 \neq n_1$. In this case the relation b is computed following the previous steps; in fact, $n_2 \models \psi_\nu(G)$ and consequently, it has to be true that $\exists \langle n_2, e_2, n_3 \rangle \in E$ s.t. $\ell(n_3) = \ell(\nu)$ and $\ell(e_2) = \ell(e)$. Finally, we find a node p such that $\ell(p) = \ell(\nu)$ and $p = n_1$. At this point it is easy to check that the relation b is a bisimulation.

The other direction of the theorem is easily proved by considering a bisimulation between the query and a subgraph I' of the instance I rooted in a node n ($\Psi_\nu(G)$ is trivially satisfied by n).

3.8 Expressive Power of Temporal Logics

In this chapter we have shown an approach based on model-checking techniques to solve either graphical queries or some simple SQL-like queries on semistructured data.

In Figure 3.16 we sum up the main results of the proposed method:

– $\mathbb{W}$-queries with also simple cyclic conditions can be translated into propositional temporal logics. In particular, general acyclic accessible queries, acyclic solid graphs and acyclic graphs, which become accessible after the application of a rewriting procedure (cf. Section 3.4.4), can be translated into *CTL* and thus, can be solved in linear time on the size of the formula and the Kripke Transition System related to the instance. Note that the set

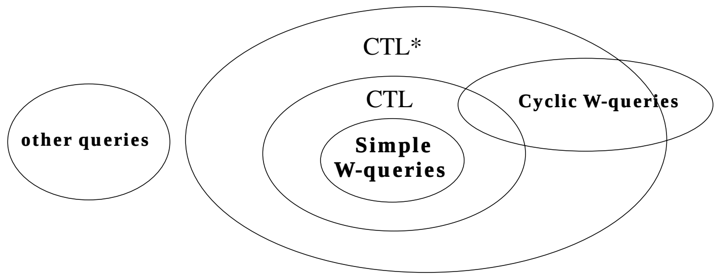


Figure 3.16. Complexity of queries w.r.t. propositional temporal logics

of these $\mathbb{W}$-queries is strictly contained in the set of *CTL* formulae. Consider for example the formula $\mathsf{AG}\varphi$ expressing a "safety property" φ that must hold in all paths and in every state of the model taken into account. We can state that it does not express any natural query on a database, because with queries we want to find out the set of nodes possessing a given property (e.g. φ) and not to verify that all nodes satisfy φ. Moreover, we have shown that often cyclic queries have not an exact translation into *CTL* and can only be approximated by using temporal formulae of this logic. On the contrary, we have shown that in general, cyclic queries can be translated into *CTL** and thus, the algorithm for solving them is PSPACE complete (cf. Section 3.4.5). We recall that only simple cyclic queries have a *CTL* translation.

– In Section 3.7 we have shown that G-Log queries can be considered in the same way as $\mathbb{W}$-queries, moreover, some other languages for semistructured data can be partially handled with the model-checking based approach. In particular, simple path expressions without join conditions and aggregator operators (e.g. the minimun, the maximun, the average) have a natural graphical representation into $\mathbb{W}$-queries and thus, have a *CTL* translation as well. In general, propositional temporal logics cannot express join conditions or computing aggregator functions on a set of states satisfying a given property.
– Join conditions are correctly expressed with first-order logic formulae and thus, we fell that only predicative modal logics [49] can deal with this kind of properties.
– We have not formally proved that all cyclic queries admit a *CTL** representation, and thus an open problem is to study whether the subset of cyclic queries that cannot be translated neither in *CTL* nor in *CTL** is empty.

Note that our technique can also be applied to XML-based languages such as Quilt [19] and XPath [96]. The input and output of these languages are documents usually modeled by trees. Once again, model-checking algorithms can correctly deal with properties expressing conditions to be satisfied on paths but without joins, grouping, or aggregation functions of SQL.

4. Temporal Aspects of Semistructured Data

In Chapter 3 we proposed a method based on model-checking techniques to efficiently solve (graphical) queries on semistructured data. However, it is known that model-checking has emerged in the past years as a promising and powerful approach to automatic verification of systems [27, 73, 28], and is especially used to verify *temporal properties*, which are usually expressed by means of temporal logics formulae, on reactive systems. Thus, as natural consequence of the temporal logic nature, in this chapter we want to adapt the technique introduced for accessing static information stored in graphical databases, to retrieve temporal aspects of semistructured data as well, such as, for example, the evolution of properties related to semistructured data.

Most applications of database technology are temporal in nature and rely on *temporal databases*, which record time-referenced data. In this setting, when history is taken into account, queries can ask about the evolution of data through time and constraints may restrict the way changes occur [24, 25]. It is clear that this kind of queries is useful whenever data are not static, they change over time and their evolution needs to be taken into account. As for the classical database field, also in the semistructured data context it is interesting to take into account evolution of data through time.

In particular, in this work we consider several dynamic aspects of semistructured data representation, based on different uses of the concept of "valid time". We define a graphical model allowing to represent and to retrieve temporal information about semistructured data according to the following interpretations of time:

- **Valid Time**: in this case data change chronologically and a sequence of modifications represents the evolution of facts in a modeled reality with respect to time. For example, the salary of an employee changes over time.
- **Interaction Time**: is the valid time relative to user browsing. When users browse through a document (for example a hypermedia representation) they choose a particular path in the graph representing semistructured information and in this way they define a personalized order between the visited objects. In this context we can create site views depending on each user's choices, that can be useful to personalize the data presentation.

The relevance of Valid Time is widely accepted, since, as in the classical database field ([90, 25, 24]), also in the context of semistructured data it is interesting to take into account the dynamic aspects of data, i.e. their evolutions through time and eventually through consecutive updates, in order to query past states and impose restrictions on how the data change in time [21].

The relevance of Interaction Time is evident because, while navigating the Web is simple, finding what the user needs in such a vast amount of information is seldom easy. Much time is wasted trying to find information and recovering from dead-end searches. It is widely recognized [8] that if users could access a customized version of the Web, their task would be made a lot simpler and less time would be wasted. User modeling research [63, 100, 9] has shown that each individual has a standard pattern of search behavior that can be used for user characterization. However, most currently available user profiling systems are still based on straightforward logging of the documents [71] as well as of the URLs and paths [65] visited by the user.

A first step in this direction consists of tracing users' navigation history, in order to recognize the habits of each user, e.g. hyperlink selections, time spent on a particular page or number of accesses to a specific information. A second step is to use and elaborate such information, in order to obtain a user model not only based on static declarations of interests and stereotypical descriptions but on actual interactions [39].

The main aim of this chapter is to present a general model for representing temporal aspects of semistructured data w.r.t. both interpretations of time. Besides accommodating the usual notion of valid time, adapted to semistructured information, this model allows also to store interactions of users with Web sites and therefore to provide a sound formal definition for user search patterns, based on the natural notion of *graph path*. Moreover, we introduce a SQL-like query language, which, in the same way of TQuel, extends SQL to take into account the time coordinate [86].

More in detail, the structure of the chapter is as follows. In Section 4.1 we introduce the main concepts related to temporal databases and in Section 4.2 we present a graphical data model suitable to represent static and dynamic aspects of semistructured data. In Section 4.3 we show possible changes to semistructured databases and in Section 4.4 we introduce a very simple SQL-like query language. To conclude, in Section 4.5 we show the applicability of our graphical temporal model in the pre-processing tasks of Web Usage Mining activities, in Section 4.6 we use the SQL-like query language to extract relevant information about users' interaction, while in Section 4.6.1 we show a model-checking approach to solve temporal queries.

4.1 An Introduction to Temporal Databases

In this section we give a brief introduction to the main concepts related to temporal databases [93, 25].

A *temporal database* is a repository of temporal information, while a *temporal query language* is any query language for temporal databases. The main issues to be addressed in this field are the following:

– *Choice of temporal domains*: points vs. intervals, linear vs. branching, dense vs. discrete, bounded vs. unbounded time.
  In the database context the term *instant* is used for a time point and the point-based view is predominant. Moreover, intervals are obtained as pairs of points. Instead, in the interval-based view it is common to have designators for interval endpoints, and thus moving between both views is very easy. There is not a single approach to represent time in a database, and different time domains may be considered. First the time domain may or may not be infinite in the past and future. Second, time may be considered as discrete, dense or continuous. The most standard temporal domains in this context are natural numbers $\langle \mathbb{N}, < \rangle$ or integer numbers $\langle \mathbb{Z}, < \rangle$, but it is commonly assumed that time is discrete and isomorphic to natural numbers [87]. These two temporal domains are "flat" and thus, in order to handle multiple time granularities (e.g. days vs. weeks) it is necessary to consider multiple interrelated temporal domains. An instant in a "higher-level" domain corresponds to a contiguous set of instants in another "lower-level" domain.
– *Data expressiveness* of a temporal data model used to represent temporal databases, that is the set of temporal databases we can express with the model and the space necessary to represent a given temporal relation.
– *Properties of query languages*: language design must consider the impact of the time-varying nature of data on all aspects of the language, including predicates on temporal values, temporal constructors, modification of temporal relations, integrity constraints. The main aspects to be considered for a temporal query language are query expressiveness, which data structures it is capable to deal with, query evaluation, and efficient implementation. Two well-known temporal query languages are TQuel [86] and TSQL2 [89, 88].

A database models and records information about a part of reality (typically called *modeled reality*). Aspects of this "reality" are represented in the database by using *database entities.* The facts (i.e. the logical statements that can be assigned a truth value) recorded by the database entities are of fundamental interest.

In general, time spans are associated with database entities, however, several different temporal aspects may be associated with them. In particular, the *valid time* of a fact is the time when the fact is true in the modeled reality. Valid time thus captures the time-varying states of the modeled reality [56]. Note that all facts have a valid time by definition, however, the valid time of a fact may not necessarily be recorded in the database for a lot of reasons (e.g. it may not be known, or recording it may be irrelevant for the applications supported by the database). The *transaction time* of a database is the time when the fact is current in the database. All database entities have a transaction time aspect but this aspect may or may not be stored in the database. Note that transaction time may be associated with any database entity, not only with facts. For example, transaction time may be associated with objects and values that are not facts because they cannot be true or false in isolation (e.g. the "Database" name of a course). The transaction time aspect of a database entity has a duration: from insertion to deletion, with multiple insertions and deletions being possible for the same entity. As a consequence, deleting an entity does not physically remove the entity from the database; rather, the entity remains in the database, but ceases to be part of the database's current state. Observe that transaction time of a database fact F is the valid time of the related fact "F is current in the database" [55].

The main aspects that are used to distinguish between valid time and transaction time can be summarized as follows [90]:

– *valid time* is characterized as the time when an event occurs in reality; *transaction time* is the time when the data concerning the event were stored in the database.
– A transaction-time value may be added to the database, yet once it has been added, it may not be changed. In fact, a time value that records when the data were stored cannot later be changed. Valid-time values, on the other hand, are always subject to change, since differences between the history (a sequence of events or time intervals) as it actually occurred and the representation of the history as stored in the database will often be detected after the fact. Thus, this distinction is between permitting only append operations or arbitrary modification operations.
– Valid time is generally characterized in the literature as being application-dependent, while transaction time is considered to be application-independent and thus, it can be automatically computed by the Database Management System. An example for understanding this difference is a retroactive salary raise, where the time at which the raise was recorded (for example, 12/10/2000) is considered application-independent, as it is not under the user's control, whereas the time at which the raise was to take effect (for example, 1/10/2000) is considered application-dependent.

Valid time and transaction time concepts allow also to characterize the features associated with a particular kind of DBMS supporting that time concept [90] in the following way.

Conventional databases model the real world, as it changes dynamically, by a snapshot at a particular point in time. A *state* or an *instance* of a database is its current content, which does not necessarily reflect the current status of the real world. Updating the state of a database is performed by using data manipulation operations such as insertion, deletion or replacement, taking effect as soon as it is committed. In this process, past states of the database, and those of the real world, are discarded and forgotten completely. We call this type of database a *static database*. In the relational model, a database is a collection of *relations*. Each relation consists of a set of *tuples* with the same set of *attributes*, and is usually represented as a 2-dimensional table [3]. As changes occur in the real world, changes are made on this table. For example, an instance of a relation "Faculty" at a certain moment may be:

Name	Rank
John	full professor
Tom	associate professor

However, there are many situations where this static database relying on snapshots is inadequate. For example, it cannot support historical queries and thus, answer to the question "What was Tom's rank 3 years ago?". It cannot also support retroactive changes or postactive changes, which are changes that are not true in the real world yet, but it is known that they are becoming true in the future.

One approach to solve the above problems is to store all past states, indexed by time, of the static database as it evolves. This approach requires a representation of *transaction time*, i.e. the time when the information was stored in the database. A relation under this approach can be illustrated conceptually in three dimensions. By fixing a time instant, it is possible to get a snapshot of the relation (a static relation) and make queries on it. The operation of extracting a snapshot is called *rollback*, and a database supporting it is called *static rollback database.*

Changes to this kind of database may only be made to the most recent static state. One limitation of supporting transaction time is that the history of database activities, rather than the history of the real world, is recorded. A tuple becomes valid as soon as it is entered into the database as in a static database. There is no way to store retroactive/postactive changes, nor to correct errors in past tuples. Errors can sometimes be overridden (if they are in the current state) by they cannot be forgotten.

A common approach to implement a static rollback database is to append the start and end points of the transaction time to each tuple, indicating the points in time when the tuple was in the database. For example, a "Faculty" relation in this approach looks as follows:

Name	Rank	Transaction time	
		Start time	End time
John	associate professor	1/1/1990	1/11/1998
John	full professor	1/11/1998	now
Tom	associate professor	1/10/1999	now

While static rollback databases record a sequence of static states, *historical databases* record a single *historical state* for each relation, storing the history as it is best known. When errors are discovered, they are corrected by modifying the database. Previous states are not retained, so it is not possible to view the database as it was in the past and there is no record stored of the errors that have been corrected. Historical databases must represent *valid time*, the time when the stored information model reality. Historical databases may also be represented in three dimensions and to implement them it is possible to append endpoints of the valid time to each tuple, indicating the points in time when the tuple accurately modeled reality. An example of the "Faculty" relation reported above is:

Name	Rank	Valid time	
		Start time	End time
John	associate professor	1/10/1989	1/10/1998
John	full professor	1/10/1998	now
Tom	associate professor	3/10/1999	now

Note that the information that John was promoted full professor on 1/10/1998 was stored in the static rollback database only one month later (the start transaction time is 1/11/1998).

Benefits of both approaches can be combined by supporting both transaction time and valid time. While a static rollback database views tuples valid at some time as of that time, and a historical database always views tuples valid at some moment as of *now*, a temporal DBMS makes it possible to view tuples valid at some moment seen as of some other moment, completely capturing the history of retroactive/postactive changes. The term *temporal database* indicates the need for both valid and transaction time in handling temporal information. A temporal relation may be thought of as a sequence of historical states, each of which is a complete historical relation. The rollback operation on a temporal relation selects a particular historical state, on which an historical query may be performed. Each transaction causes a new historical state to be created, hence, temporal relations are append-only. Now the "Faculty" database looks like:

Name	Rank	Valid time		Transaction time	
		Start time	End time	Start time	End time
John	associate	1/10/1989	now	1/01/1990	1/11/1998
John	associate	1/10/1989	1/10/1998	1/11/1998	now
John	full	1/10/1998	now	1/11/1998	now
Tom	associate	3/10/1999	now	1/10/1999	now
Tom	full	3/10/1999	now	5/10/1999	now

Note that the relation shows that John started working on 1/10/1989 but the information was entered into the database on 1/1/1990 retroactively. Tom was entered into the database on 1/10/1999 as joining the faculty as an associate professor on 3/10/1999. The fact that he was actually a full professor was noted on 5/10/1999.

Another last notion of time, called *user-defined time* [59] can be used when additional temporal information, not handled by transaction or valid time, is stored in the database. For example, for the "Faculty" relation one may be interested in the "promotion date", that is the date shown on the promotion letter. In this case the valid date is the date when the promotion letter was signed, i.e. the date when the promotion was validated, and the transaction date is the date when the information concerning the promotion was stored into the database. Note that this notion of time is application-dependent.

These concepts of classical temporal databases can be applied also to semistructured databases, which are usually represented by means of labeled graphs instead of flat tables (see [52] for a survey). Thus, the first step for studying temporal aspects of semistructured data is to extend graph based data models in order to keep trace of historical information.

4.2 A Graphical Temporal Data Model for Semistructured Data

In this section we introduce a data model based on labeled graphs, useful to represent static and dynamic aspects related to semistructured data. We do not focus on a particular existing static model because our choice is general enough to apply to several approaches presented in the literature [33, 81, 66].

In order to represent dynamic aspects of semistructured databases and to allow querying about their evolution through time, we add the information about the *temporal element* of objects and relations to their graph. This kind of information represents the *validity* of objects and relations, i.e. the time when the objects or relationships are true in the modeled reality [30, 57]. We need to consider temporal element instead of simple intervals, in order to support "reincarnation" since the death of an object is not necessarily terminal. In this way, we are able to derive, for example, if a particular information is currently true, or if it was true in a previous state, if it has been deleted or updated.

This approach allows us to store the minimal amount of information, in fact we avoid duplicating a lot of nodes and edges (cf. Sect. 5.3.1 of Chapter 5).

Definition 4.2.1. *A* semistructured temporal graph *is a directed labeled rooted graph $\langle N, E, r, \ell \rangle$, where N is a (finite) set of nodes, E is a set of labeled edges of the form $\langle m, label, n \rangle$, with $m, n \in N$, $label \in (\mathcal{T}_e \times (\mathcal{L}_e \cup \{\perp\}) \times (\mathcal{V} \cup \{\perp\}))$, $r \in N$ is the root of the graph, and $\ell : N \longrightarrow (\mathcal{T}_n \cup \{\perp\}) \times (\mathcal{L}_n \cup \{\perp\}) \times (\mathcal{S} \cup \{\perp\}) \times (\mathcal{V} \cup \{\perp\})$. $\perp$ means 'undefined', and:*

- *$\mathcal{T}_n = \{$ complex, atomic $\}$ is a set of types for nodes;*
- *$\mathcal{T}_e = \{$ relational, temporal $\}$ is a set of types for edges;*

- $\mathcal{L}_n$ *and* $\mathcal{L}_e$ *are sets of labels to be used respectively as names for complex or atomic objects (nodes), and relations (edges). We require that* $\mathcal{L}_n \cap \mathcal{L}_e = \emptyset$*;*
- $\mathcal{S}$ *is a set of strings to be used as atomic values;*
- $\mathcal{V}$ *is a set of temporal elements such as* $[t_1, t_2) \cup [t_3, t_4) \cup \ldots \cup [t_{n-1}, t_n)$*, with* $n > 1$*, to be used as time intervals for nodes and edges. We use a temporal element to keep trace of different time intervals when an object exists in the database.*

ℓ *can be seen as composed by four single-valued functions* $\ell_{\mathcal{T}_n}, \ell_{\mathcal{L}_n}, \ell_{\mathcal{S}}, \ell_{\mathcal{V}}$*. With abuse of notation, when the context is clear, we will use* ℓ *for nodes and edges: if* $e = \langle m, \langle t, k, v\rangle, n\rangle$*, then* $\ell_{\mathcal{T}_e}(e) = t$*,* $\ell_{\mathcal{L}_e}(e) = k$*, and* $\ell_{\mathcal{V}}(e) = v$*. It is also required that:*

- $(\forall x \in N)(\ell_{\mathcal{T}_n}(x) = complex \rightarrow \ell_{\mathcal{S}}(x) = \perp)$ *(i.e. values are associated to* atomic objects *only);*
- $(\forall x \in N)(\ell_{\mathcal{T}_n}(x) = atomic \rightarrow \ell_{\mathcal{V}}(x) = \perp)$ *(i.e. temporal elements are not associated to atomic objects because in the database the temporal element of a value coincides with the temporal element of its unique ingoing edge);*
- $(\forall e = \langle m, label, n\rangle \in E)(\ell_{\mathcal{T}_n}(m) = complex)$ *(i.e. atomic nodes are leaves);*
- $(\forall e = \langle m, \langle relational, label\rangle, n\rangle \in E)(\ell_{\mathcal{T}_n}(n) = atomic \rightarrow label =$ *"Has Pro_ perty") (i.e. each atomic node is connected to its parent by an edge labeled "Has Property").*
- $(\forall x \in E)(\ell_{\mathcal{T}_e}(x) = temporal \rightarrow \ell_{\mathcal{V}}(x) = \perp)$ *(i.e. temporal elements are associated to* relational edges *only);*
- $(\forall x \in E)(\ell_{\mathcal{T}_e}(x) = temporal \rightarrow \ell_{\mathcal{L}_e}(x) =$ *"Temporal") (i.e. the label "Temporal" is associated to each* temporal edge*).*

Note that two nodes may be connected by more than one edge, provided that edge labels be different.

In this work we assume that each node of a semistructured temporal graph is unambiguously characterized by an *identifier*. Moreover, the label of the (unique) root represents the name of the database.
For the sake of readability, in the examples we usually omit the type label of nodes and edges.

Temporal edges are represented by jagged lines. As we introduced in Definition 4.2.1 we distinguish two kinds of nodes: complex objects, graphically represented by rectangles, and atomic objects, depicted as ovals. Complex objects are related to other complex objects and possibly "possess" a number of attributes (atomic objects), whereas atomic objects represent objects with an atomic value (i.e. a string, an integer, but also a text, an image, a sound) and do not exist independently of their parent complex object. For this reason we choose to replicate atomic objects with the same atomic value but connected to different complex objects, although they represent the same printable information. This is also the reason why we do not associate an independent temporal element to atomic objects. For each complex object it is possible to declare a *key*, that is a subset of its properties (attributes) that identifies in a unambiguous way the object itself.

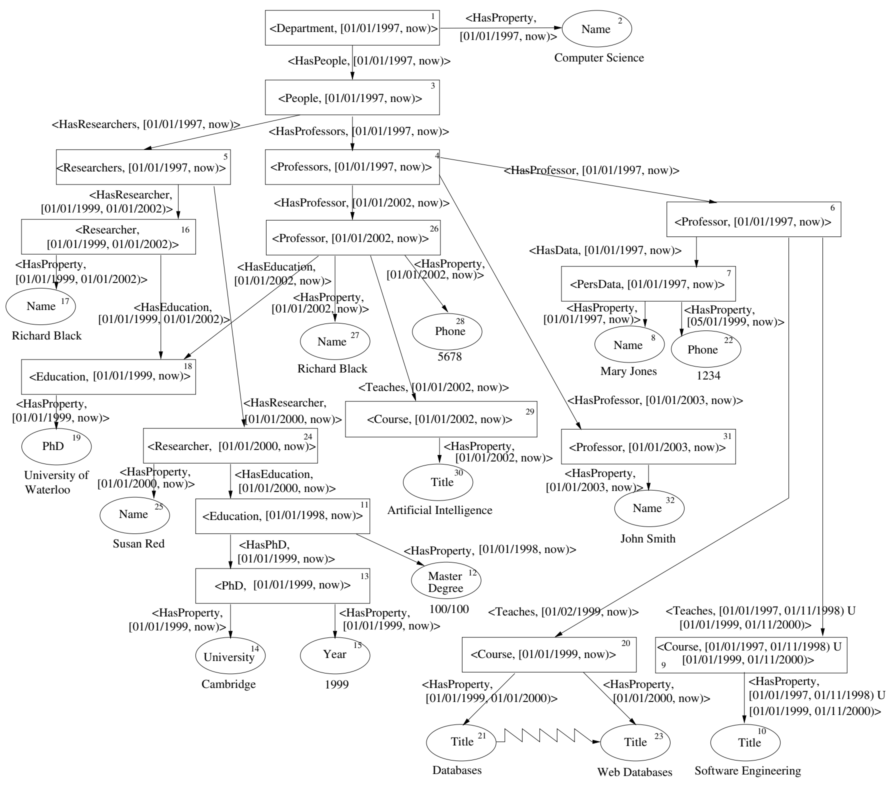

<Department, [01/01/1997, now)> 1
<HasProperty, [01/01/1997, now)>
Name 2
Computer Science
<HasPeople, [01/01/1997, now)>
<People, [01/01/1997, now)> 3
<HasResearchers, [01/01/1997, now)>
<HasProfessors, [01/01/1997, now)>
<Researchers, [01/01/1997, now)> 5
<Professors, [01/01/1997, now)> 4
<HasProfessor, [01/01/1997, now)>
<Professor, [01/01/1997, now)> 6
<HasResearcher, [01/01/1999, 01/01/2002)>
<Researcher, [01/01/1999, 01/01/2002)> 16
<HasProfessor, [01/01/2002, now)>
<Professor, [01/01/2002, now)> 26
<HasData, [01/01/1997, now)>
<PersData, [01/01/1997, now)> 7
<HasProperty, [01/01/1999, 01/01/2002)>
Name 17
Richard Black
<HasEducation, [01/01/1999, 01/01/2002)>
<HasEducation, [01/01/2002, now)>
<HasProperty, [01/01/2002, now)>
<HasProperty, [01/01/2002, now)>
Name 27
Richard Black
Phone 28
5678
<HasProperty, [01/01/1997, now)>
<HasProperty, [05/01/1999, now)>
Name 8
Mary Jones
Phone 22
1234
<Education, [01/01/1999, now)> 18
<HasProperty, [01/01/1999, now)>
PhD 19
University of Waterloo
<HasResearcher, [01/01/2000, now)>
<Researcher, [01/01/2000, now)> 24
<Teaches, [01/01/2002, now)>
<Course, [01/01/2002, now)> 29
<HasProperty, [01/01/2002, now)>
Title 30
Artificial Intelligence
<HasProfessor, [01/01/2003, now)>
<Professor, [01/01/2003, now)> 31
<HasProperty, [01/01/2003, now)>
Name 32
John Smith
<HasProperty, [01/01/2000, now)>
Name 25
Susan Red
<HasEducation, [01/01/2000, now)>
<Education, [01/01/1998, now)> 11
<HasPhD, [01/01/1999, now)>
<HasProperty, [01/01/1998, now)>
Master Degree 12
100/100
<PhD, [01/01/1999, now)> 13
<HasProperty, [01/01/1999, now)>
<HasProperty, [01/01/1999, now)>
University 14
Cambridge
Year 15
1999
<Teaches, [01/02/1999, now)>
<Teaches, [01/01/1997, 01/11/1998) U [01/01/1999, 01/11/2000)>
<Course, [01/01/1999, now)> 20
<Course, [01/01/1997, 01/11/1998) U [01/01/1999, 01/11/2000)> 9
<HasProperty, [01/01/1999, 01/01/2000)>
<HasProperty, [01/01/2000, now)>
<HasProperty, [01/01/1997, 01/11/1998) U [01/01/1999, 01/11/2000)>
Title 21
Databases
Title 23
Web Databases
Title 10
Software Engineering

Figure 4.1. A semistructured temporal graph reporting faculty data

For example, in Figure 4.1 we show a semistructured temporal graph, called Department (note that Department is the label of the root node), representing information about some teachers and related courses.

As explained above, we associate a temporal element $I = [t_1, t_2) \cup \ldots \cup [t_{n-1}, t_n)$, with $n > 1$, to each relational edge and complex object; for each time interval $[t_i, t_{i+1})$ in I, with $i \in \{1, \ldots, n-1\}$, the instant t_i represents the *start time* (i.e. the "birth" of the object), while t_{i+1}, called *end time*, represents the time instant when that piece of information (node or edge) ceased to be true in the reality (i.e. the "death" of the object). In the following sections, when an entity is currently true, i.e. it is valid, t_n assumes the constant value "*now*". This value can be seen as ∞ and it does not cause difficulties when it is used as end time, although the time interval is open to the right.

In the examples we propose later in this work, time granularity depends on the application: it may be one day, or one year, etc. However, our approach will encode instants of time in a discrete fashion.

4.3 Operations on Temporal Data

In this section we describe a set of possible operations that could affect a semistructured database and how its semistructured temporal graph is consequently modified.

Let $G = \langle N, E, r, \ell \rangle$ be a semistructured temporal graph. We consider the following change operations:

1. **Insert the root node**
   $obj_r = \textit{insert-root-node}(\textit{complex}, l, [t_i, t_j))$ inserts the root node with labels $\ell_{\mathcal{T}_n}(obj_r) = \textit{complex}$, $\ell_{\mathcal{L}_n}(obj_r) = l$, and gives as result the object identifier obj_r of the root node itself.
2. **Insert a complex object**
   $obj_k = \textit{insert-complex-node}(\textit{complex}, l, P, [t_i, t_j))$ inserts a node with labels $\ell_{\mathcal{T}_n}(obj_k) = \textit{complex}$, $\ell_{\mathcal{L}_n}(obj_k) = l$, set of attributes P, and gives as result the object identifier obj_k of the node itself. Note that P is a set of couples $\langle Label, Value \rangle$. The valid time of the inserted object is $[t_i, t_j)$. If t_j is the constant "now" then the object is current, otherwise it is historic. This operation inserts also the set of atomic objects P (which represent the properties of the object obj_k) by calling the operation *insert-atomic-node* for all the pairs $\langle Label, Value \rangle \in P$, as described below.
   In Figure 4.2 we show the insertion of the current complex object "Course" and the creation of its atomic object with label "Title" and value "Databases". The valid time of the inserted objects is [01/01/1999, *now*). Note that the valid time of the atomic object is reported on the edge labeled "Has Property" between "Course" and "Title". The insert operation is:
   insert-complex-node(*complex*, Course, {⟨Title, Databases⟩}, [01/01/1999, *now*))
   and returns as result the object identifier of the created node obj_3.
   After this operation the course node is not reachable from the root and thus cannot be retrieved by using TSS-QL. However the node itself can be used by other operations and in particular can be connected with a path to the root of the graph.
3. **Insert a relationship**
   $\textit{insert-relationship}(obj_k, label, obj_t, [t_i, t_j))$ adds an edge labeled $label$ between obj_k and obj_t. This operation does not have constraints and does not make sense between a complex and an atomic object because properties, represented by means of atomic objects, are part of complex objects.

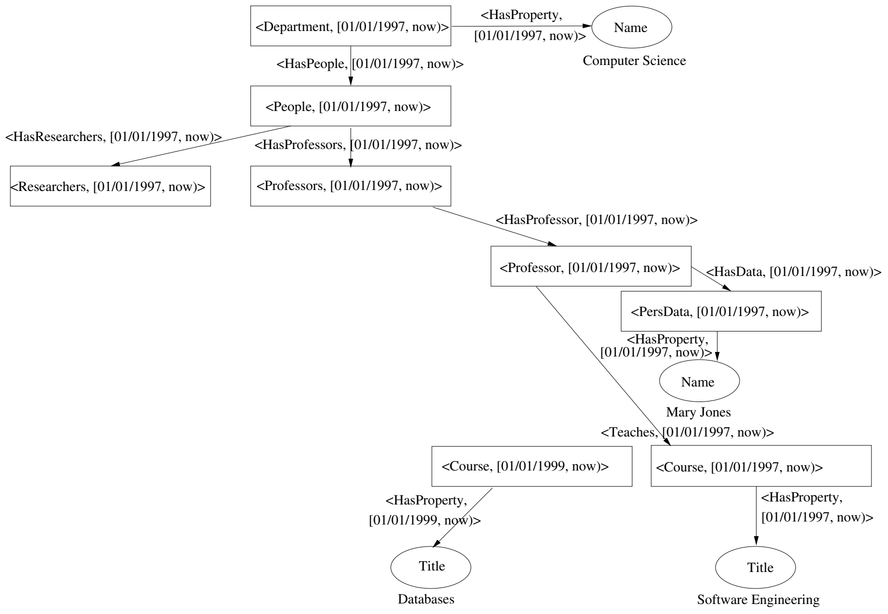

<Department, [01/01/1997, now)>
<HasProperty, [01/01/1997, now)>
Name
Computer Science
<HasPeople, [01/01/1997, now)>
<People, [01/01/1997, now)>
<HasResearchers, [01/01/1997, now)>
<HasProfessors, [01/01/1997, now)>
<Researchers, [01/01/1997, now)>
<Professors, [01/01/1997, now)>
<HasProfessor, [01/01/1997, now)>
<Professor, [01/01/1997, now)>
<HasData, [01/01/1997, now)>
<PersData, [01/01/1997, now)>
<HasProperty, [01/01/1997, now)>
Name
Mary Jones
<Teaches, [01/01/1997, now)>
<Course, [01/01/1999, now)>
<Course, [01/01/1997, now)>
<HasProperty, [01/01/1999, now)>
<HasProperty, [01/01/1997, now)>
Title
Databases
Title
Software Engineering

Figure 4.2. A new course has been added

We suppose the professor "Mary Jones" starts teaching the course "Databases" at time 01/02/1999. In Figure 4.3 we show the new relationship "Teaches" with valid time $[01/02/1999, now)$, between the complex objects "Professor" and "Course", that we obtain with the operation:

$$\textit{insert-relationship}(obj_1, Teaches, obj_3, [10/02/1999, now))$$

4. **Insert an atomic object**
$obj_t = \textit{insert-atomic-node}(obj_p, atomic, l, s, [t_i, t_j))$ creates a child node of obj_p with labels $\ell_{\mathcal{T}_n}(obj_t) = atomic$, $\ell_{\mathcal{L}_n}(obj_t) = l$, $\ell_{\mathcal{S}}(obj_t) = s$ and $\ell_{\mathcal{V}}(obj_t) = \perp$. The operation returns as result the object identifier obj_t. Moreover, this operation adds also an edge from the parent complex object obj_p to the atomic object itself, with label "Has Property" and temporal element $[t_i, t_j)$. Note that in the valid time context the insertion of an atomic object is used to insert new values but also to record an update to a value. Indeed, any change which involves a property should still keep information about its previous value by adding a temporal edge (see the evolution of the "Title" property for the "Databases" Course in Fig. 4.1). Note that an atomic object is always connected to a complex object, because it represents a property of the complex object, and thus the temporal element of its ingoing edge starts after the start time of its parent, i.e. the following constraint must hold: $t_1 \leq t_i$ and $t_j \leq t_2$, where $\ell_{\mathcal{V}}(obj_p) = [t_1, t_2)$.

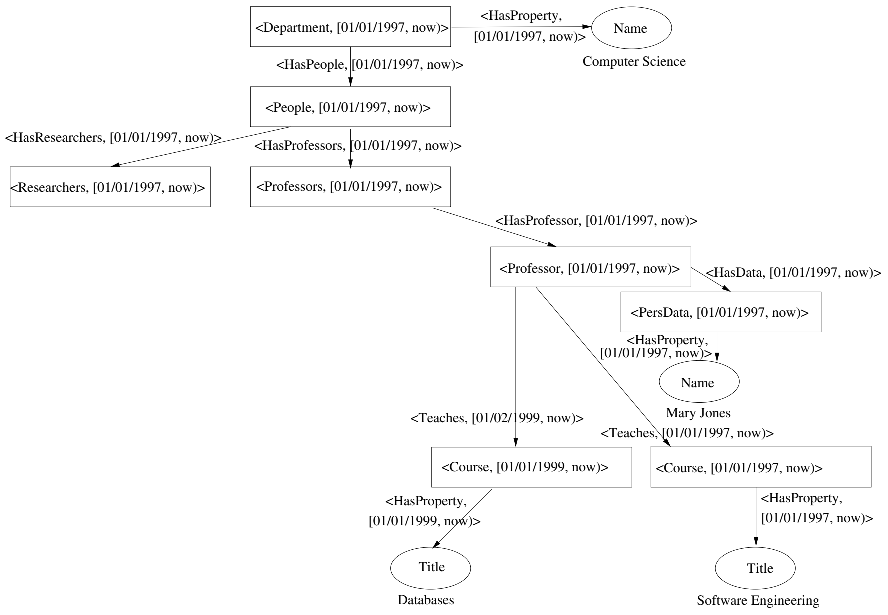

<Department, [01/01/1997, now)>
<HasProperty, [01/01/1997, now)>
Name
Computer Science
<HasPeople, [01/01/1997, now)>
<People, [01/01/1997, now)>
<HasResearchers, [01/01/1997, now)>
<HasProfessors, [01/01/1997, now)>
<Researchers, [01/01/1997, now)>
<Professors, [01/01/1997, now)>
<HasProfessor, [01/01/1997, now)>
<Professor, [01/01/1997, now)>
<HasData, [01/01/1997, now)>
<PersData, [01/01/1997, now)>
<HasProperty, [01/01/1997, now)>
Name
Mary Jones
<Teaches, [01/02/1999, now)>
<Teaches, [01/01/1997, now)>
<Course, [01/01/1999, now)>
<Course, [01/01/1997, now)>
<HasProperty, [01/01/1999, now)>
<HasProperty, [01/01/1997, now)>
Title
Databases
Title
Software Engineering

Figure 4.3. A relationship has been inserted

In Figure 4.4 we show the insertion of the current atomic object "Phone", with value "1234" and temporal element $[05/01/1999, now)$, which is reported on the edge labeled "Has Property" between "PersData" and "Phone". Supposing that PersData has as object identifers obj_5, then the insert operation is:

$$\textit{insert-atomic-node}(obj_5, atomic, Phone, 1234, [05/01/1999, now))$$

and returns as result obj_6.

5. **Remove an object**
   remove-node(obj_k) removes the node with identifier obj_k (if it exists). Note that if the object is complex we also remove its properties (i.e. its atomic objects) and also all the relationships with other complex objects. After this operation some portions of the original graph can be unreachable from the root node.
6. **Remove a relationship** (between complex objects)
   remove-relationship$(obj_k, label, obj_t)$ removes an existing edge.
   Note that it is possible to remove a relationship between valid objects. In fact, if we remove an edge $\langle obj_k, label, obj_t\rangle$ we do not delete the object obj_t, although it is not reachable, because it is a valid object, i.e. it is currently true in the reality. Also after this operation some portions of the original graph can be unreachable from the root node.

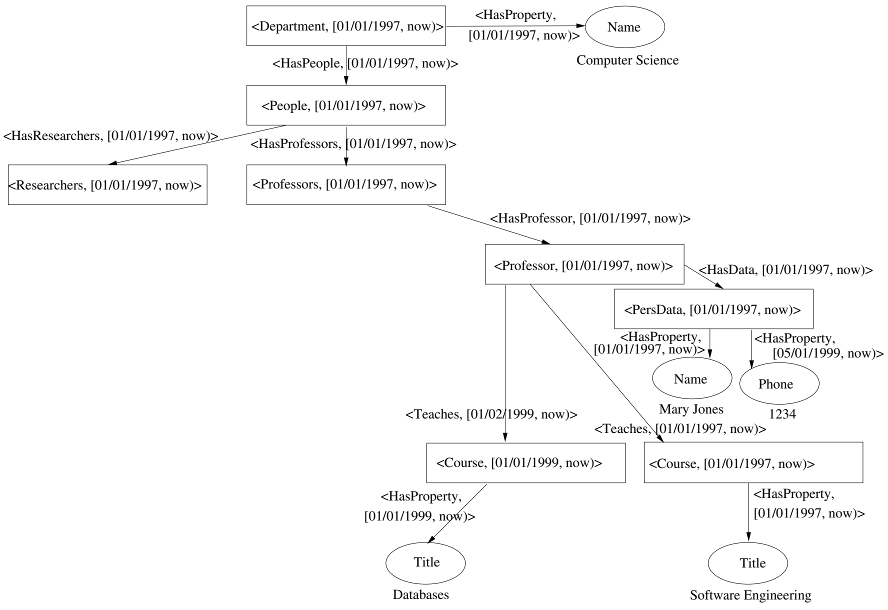

<Department, [01/01/1997, now)>
<HasProperty, [01/01/1997, now)>
Name
Computer Science
<HasPeople, [01/01/1997, now)>
<People, [01/01/1997, now)>
<HasResearchers, [01/01/1997, now)>
<HasProfessors, [01/01/1997, now)>
<Researchers, [01/01/1997, now)>
<Professors, [01/01/1997, now)>
<HasProfessor, [01/01/1997, now)>
<Professor, [01/01/1997, now)>
<HasData, [01/01/1997, now)>
<PersData, [01/01/1997, now)>
<HasProperty, [01/01/1997, now)>
<HasProperty, [05/01/1999, now)>
Name
Mary Jones
Phone
1234
<Teaches, [01/02/1999, now)>
<Teaches, [01/01/1997, now)>
<Course, [01/01/1999, now)>
<Course, [01/01/1997, now)>
<HasProperty, [01/01/1999, now)>
<HasProperty, [01/01/1997, now)>
Title
Databases
Title
Software Engineering

Figure 4.4. A new atomic object has been added

7. **Modify the temporal element of an object**
   modify-object-interval$(obj_k, OldTemporalElement, NewTemporalEle-ment)$ checks the valitidy of the constraint imposing that the temporal element of atomic objects must be included in the temporal element of their parent. If this constraint is satisfied the operation modifies the temporal element of the node.
   In Figure 4.5 we modify the temporal element of the complex object "Course" and also of its atomic object "Title" (with value "Software Engineering"). To modify the temporal element of an atomic object, we change the temporal element of its "Has Property" edge. The operation for the "Course" object is:

   modify-object-interval$(obj_2, [01/01/1997, \text{now}), [01/01/1997, 01/11/2000))$

8. **Modify the temporal element of a relation**
   modify-relation-interval$(obj_k, label, obj_t, OldTemporalElement, New-TemporalElement)$ consists of modifying the temporal element of a relation.
   We suppose the professor "Mary Jones" stops teaching "Software Engineering" at time 01/11/2000. In Figure 4.6 we modify the temporal element of the relationship "Teaches", between "Professor" ("Mary Jones") and "Course" ("Software Engineering"), from the temporal ele-

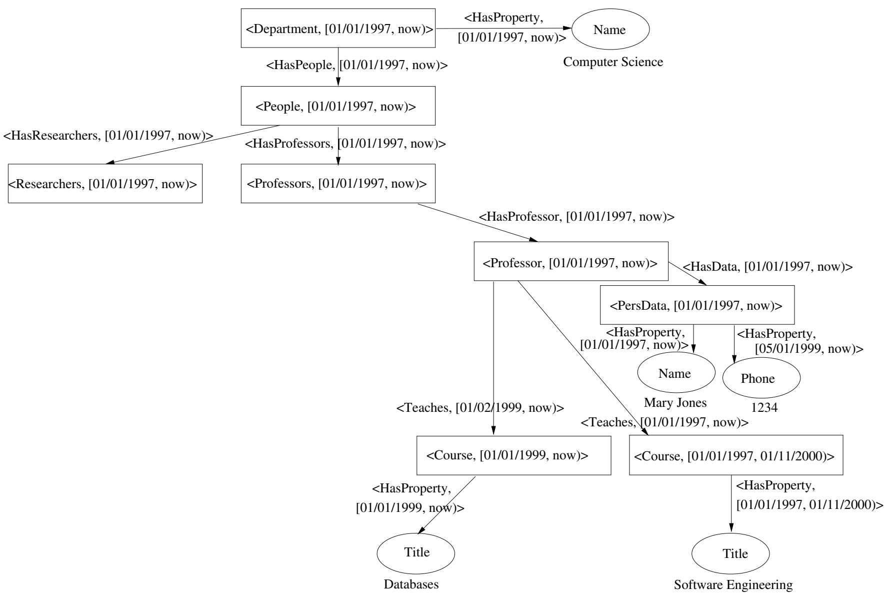

<Department, [01/01/1997, now)>
<HasProperty, [01/01/1997, now)>
Name
Computer Science
<HasPeople, [01/01/1997, now)>
<People, [01/01/1997, now)>
<HasResearchers, [01/01/1997, now)>
<HasProfessors, [01/01/1997, now)>
<Researchers, [01/01/1997, now)>
<Professors, [01/01/1997, now)>
<HasProfessor, [01/01/1997, now)>
<Professor, [01/01/1997, now)>
<HasData, [01/01/1997, now)>
<PersData, [01/01/1997, now)>
<HasProperty, [01/01/1997, now)>
<HasProperty, [05/01/1999, now)>
Name
Mary Jones
Phone
1234
<Teaches, [01/02/1999, now)>
<Teaches, [01/01/1997, now)>
<Course, [01/01/1999, now)>
<Course, [01/01/1997, 01/11/2000)>
<HasProperty, [01/01/1999, now)>
<HasProperty, [01/01/1997, 01/11/2000)>
Title
Databases
Title
Software Engineering

Figure 4.5. A temporal element of an object has been modified

ment $[01/01/1997, now)$ to the new one $[01/01/1997, 01/11/2000)$. The operation is:
modify-relation-interval$(obj_1, Teaches, obj_2,$ [01/01/1997, now), [01/01/1997,01/11/2000))

9. **Update a value**
update$(obj_k$, *old-value*, *new-value*) is used to correct a value: we change in the graph the label $\ell_{\mathcal{L}_n}(obj_k)$ from *old-value* to *new-value*.
In Figure 4.7 we show the database after the update of the value "Databases" which is modified into "Web Databases". Supposing the object identifier of this property is obj_7, the operation is used for this update is:
$$update(obj_7, Databases, WebDatabases)$$
The new value is valid from 01/01/2000, thus the operation *modify-object-interval* is also needed.

4.4 TSS-QL: Temporal Semistructured Query Language

In this section we present the SQL-like query language *TSS-QL* [77], syntactically specified in Section 4.4.1, aimed at querying semistructured data modeled by means of semistructured temporal graphs (see Section 4.2). The semantics will be informally specified only by examples.

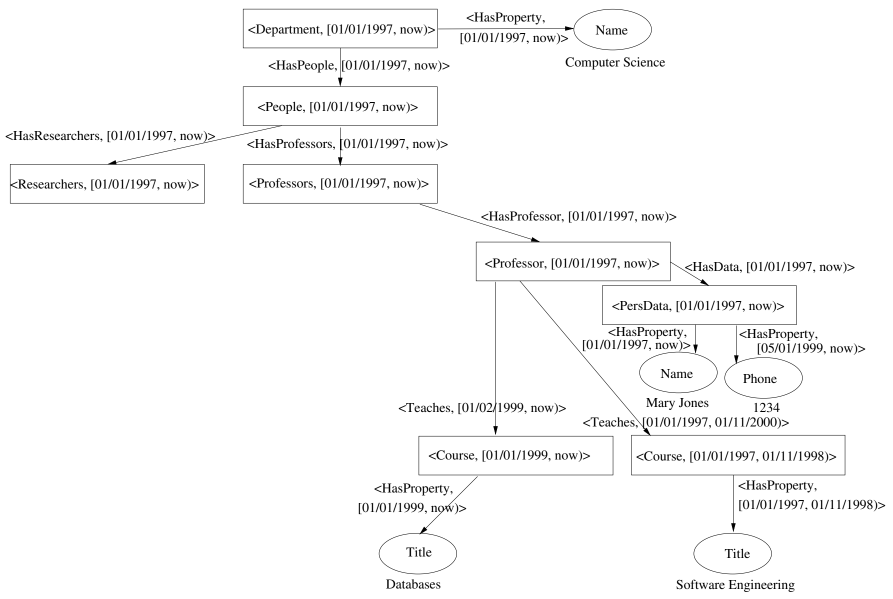

<Department, [01/01/1997, now)>
<HasProperty, [01/01/1997, now)>
Name
Computer Science
<HasPeople, [01/01/1997, now)>
<People, [01/01/1997, now)>
<HasResearchers, [01/01/1997, now)>
<HasProfessors, [01/01/1997, now)>
<Researchers, [01/01/1997, now)>
<Professors, [01/01/1997, now)>
<HasProfessor, [01/01/1997, now)>
<Professor, [01/01/1997, now)>
<HasData, [01/01/1997, now)>
<PersData, [01/01/1997, now)>
<HasProperty, [01/01/1997, now)>
<HasProperty, [05/01/1999, now)>
Name
Mary Jones
Phone
1234
<Teaches, [01/02/1999, now)>
<Teaches, [01/01/1997, 01/11/2000)>
<Course, [01/01/1999, now)>
<Course, [01/01/1997, 01/11/1998)>
<HasProperty, [01/01/1999, now)>
<HasProperty, [01/01/1997, 01/11/1998)>
Title
Databases
Title
Software Engineering

Figure 4.6. The temporal element of a relationship has been modified

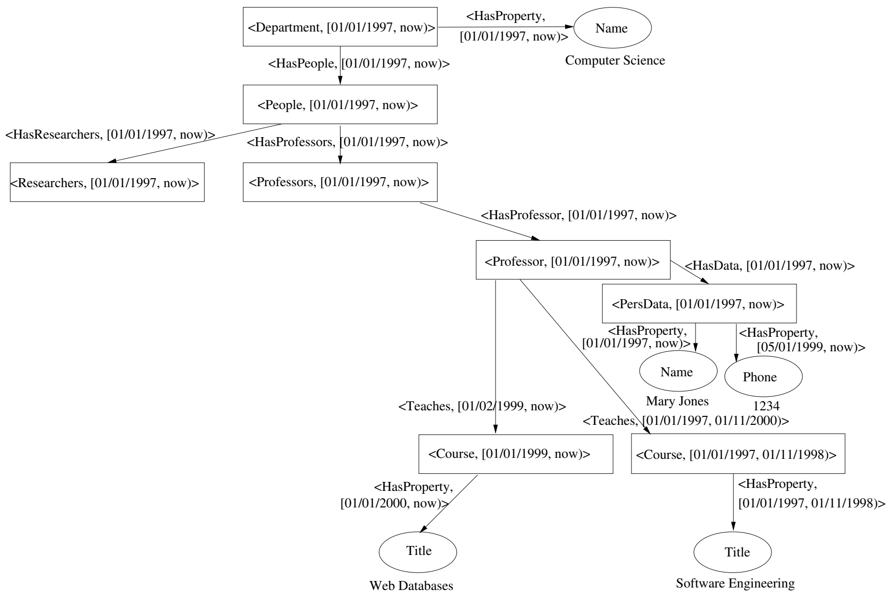

<Department, [01/01/1997, now)>
<HasProperty, [01/01/1997, now)>
Name
Computer Science
<HasPeople, [01/01/1997, now)>
<People, [01/01/1997, now)>
<HasResearchers, [01/01/1997, now)>
<HasProfessors, [01/01/1997, now)>
<Researchers, [01/01/1997, now)>
<Professors, [01/01/1997, now)>
<HasProfessor, [01/01/1997, now)>
<Professor, [01/01/1997, now)>
<HasData, [01/01/1997, now)>
<PersData, [01/01/1997, now)>
<HasProperty, [01/01/1997, now)>
<HasProperty, [05/01/1999, now)>
Name
Mary Jones
Phone
1234
<Teaches, [01/02/1999, now)>
<Teaches, [01/01/1997, 01/11/2000)>
<Course, [01/01/1999, now)>
<Course, [01/01/1997, 01/11/1998)>
<HasProperty, [01/01/2000, now)>
<HasProperty, [01/01/1997, 01/11/1998)>
Title
Web Databases
Title
Software Engineering

Figure 4.7. A value has been updated

We use the concept of path expression, which identifies a path on the information graph. The path expression has to start and finish with a node label and may contain *wildcards*, which allow the user to access information also when their structure is not known.

The **SELECT** clause specifies the form of the result of a query by means of a list of path expressions. In particular the result is a set of paths in the instance graph.

A path expression may contain different wildcard characters, whose meaning is the following:

"*" represents any sequence $\langle e_0, o_1, e_1, \ldots, o_n, e_n \rangle$ with $n \geq 0$. Note that the length of the path is the number of nodes in the path, and thus, if $n = 0$ the sequence is composed by one edge.

"?" represents a sequence $\langle e_0 \rangle$ or $\langle e_0, o_1, e_1 \rangle$. In the former case the wildcard is used to represent, between two nodes, a relationship whose name we do not know, whereas, in the latter case, it is used to represent an unknown path composed by edge-node-edge.

The **FROM** clause contains a list of database names (actually, a list of root node labels of semistructured temporal graphs) and specifies the name of the databases (i.e. their temporal graphs) to be considered as information sources.

The **WHERE** clause specifies some conditions (possibly linked by connectives such as AND, OR) on objects or sub-objects matching the **SELECT** clause. It is thus possible to define restrictions on the values of atomic objects or on the temporal elements of nodes and edges. Specifically, temporal constructs are introduced for referencing the temporal elements of nodes and edges ($\rightarrow$ **INTERVAL**), for testing a particular condition on temporal element w.r.t a time instant (*begin*, *end*, *begin before*, ...), and for the comparison of different temporal elements (*precede*, *overlap*, *equal*). Relations between temporal elements are the classical ones recognized and defined in [62].

In order to allow the user to specify, during the query composition process, the exact structure of the data to retrieve, we include the **EXIST** operator. In a TSS-QL query this operator is followed by the path specifying the structure in the graph related to a database of the information of interest.

4.4.1 Grammar of TSS-QL

In this section we introduce the grammar for the language TSS-QL by proposing different SQL-like fragments; in this way, we gradually increase query expressiveness. We plan to add new temporal operators to the language and the possibility to express updates as well.

The first fragment, i.e. the "Path Fragment", is called **P-Fragment** and shown in Figure 4.8. We have called this portion of TSS-QL the "Path Fragment" because of the nature of the information we can extract (paths in

```
<query>::= SELECT [<label> AS] <select list>
           [ FROM <from list> ]
           [ WHERE [<label> AS] <condition> ]
           [ EXCEPT <query> ]

<select list>::= <select path>, <select list>|
                 <select path>

<select path>::= <node path>|<attribute t-list path>

<node path>::= <node label>|<edge path>.<node label>

<node label>::= <object label>|<attribute label>

<edge path>::= <object label>.<edge label>| <object label>.<wildcard>|
               <object label>.<edge label>.<edge path>|
               <object label>.<wildcard>.<edge path>

<attribute t-list path> ::= <attribute t-list>| <edge path>.<attribute t-list>

<attribute t-list> ::= <attribute label>.Temporal.<attribute label>|
                       <attribute label> # <attribute label>|
                       <attribute label> # <attribute t-list>

<wildcard>::= ∗ | ?

<from list>::= <DB label>, <from list>| <DB label>

<condition>::= <predicate>
               <condition> OR <condition>|
               <condition> AND <condition>|

<predicate>::=EXISTS <node path>|
              <attribute path> <op> <value>
              <node path> IN <query>

<attribute path>::=<attribute label>|
                   <object path>.<edge label>.<attribute path>|
                   <object path>.<wildcard>.<attribute path>|

<op>::= = | =! =
```

The intuitive meaning of the remaining nonterminals is the following:

<label>	a string
<DB label>	a database name
<object label>	a complex node name
<attribute label>	an atomic node name
<edge label>	an edge label
<value>	a constant string

Figure 4.8. The **P-Fragment** of TSS-QL

the semistructured graph including simple conditions on atemporal as well as temporal information. In particular, equality of temporal elements can be checked, and temporal edges can be followed in order to investigate on objects history). The semantics of this fragment is specified by translating its queries into formulae of the logic *CTL* (see Section 4.6.1).

With the "Join-Temporal Fragment", called **JT-Fragment**, the expressive power of TSS-QL is augmented in two directions: a) it introduces the possibility of explicitly including complex temporal operators in queries, by means of complex conditions on the temporal elements of objects and relationships, for example, conditions to determine whether two temporal elements are disjoint, partially overlap, are one included in the other, etc. (see [62] for the semantics of these conditions), b) it allows comparisons between values belonging to different paths. For this last aspect, with the JT-Fragment we can express properties that force, for example, the equality between values in two different paths of the semistructured temporal graph representing the information of interest. This possibility reminds the "join operator" of SQL in the classical relational context.

Note that, however, the "Path Fragment" is already rather expressive. Indeed, although explicit comparisons between node values cannot be performed, many of the joins that would be required if the data were represented in the classical relational way are directly expressed by path traversal.

The **JT-Fragment** is reported in Figures 4.9 and 4.10. Note that the temporal operators, here, include all kinds of comparison.

With the **CA-Fragment**, which is reported in Figures 4.20 and 4.21, we add the possibility to use SQL standard aggregate operators, such as, for example, *MIN*, *MAX*, and *COUNT*.

4.4.2 Some Examples of TSS-QL Queries

In this section we give a flavour of the semantics of TSS-QL by showing some intuitive examples of queries on a database called Faculty; Fig. 4.1 reports a part of its semistructured temporal graph. Note that here time granularity is "one day". For each query we report the SQL-like form based on the grammar described in Section 4.4.1, and a possible correspondent graphical representation that reflects the main features of G-Log (cf. Chapter 2). We investigate on the possibility to translate TSS-QL queries into graphical queries for applying all the results we have shown in this work for graph-based languages.

In this work we do not give a precise grammar for the graphical version of our language, but it is quite intuitive to associate a meaning to the graphical counterpart of each specific query. A work which addresses the problem of comparing and translating graphs and textual queries is [40].

```
<query>::= SELECT [<label> AS] <select list>
           [ FROM <from list> ]
           [ WHERE [<label> AS] <condition> ]
           [ EXCEPT <query> ]

<select list>::= <select path>, <select list>|<select path>

<select path>::= <node path>|
                 <attribute t-list path>
                 <interval path>→INTERVAL

<node path>::= <node label>|<edge path>.<node label>

<node label>::= <object label>|<attribute label>

<edge path>::= <object label>.<edge label>| <object label>.<wildcard>|
               <object label>.<edge label>.<edge path>|
               <object label>.<wildcard>.<edge path>

<interval path>::= <object label>|<edge label>|<edge path>|
                   <edge path>.<node label>

<object path>::= <object label>|<edge path>.<object label>

<attribute t-list path> ::= <attribute t-list>| <edge path>.<attribute t-list>

<attribute t-list> ::= <attribute label>.Temporal.<attribute label>|
                       <attribute label> # <attribute label>|
                       <attribute label> # <attribute t-list>

<wildcard>::= * | ?

<from list>::= <DB label>, <from list>| <DB label>
```

Figure 4.9. The SELECT-FROM clauses of the **JT-Fragment**

More in detail, we represent complex objects with thin line rectangles and atomic objects with thin line ovals. Each object has a label composed by its name and the time interval (note that we represent a generic interval with _). The part of the graph depicted with thin lines represents the conditions specified in the **WHERE** clause (i.e. the structure of required information). A bold square links the result of the query (i.e. the object specified in the **SELECT** clause).

Query 1. Find the names of the professors who were working (i.e. teaching at least one course) during the whole year 1999. (supposing that the "Name" property is a sub-object of a node called "PersData").

```
<condition>::= <predicate>
               <condition> OR <condition>|
               <condition> AND <condition>|

<predicate>::=EXISTS <node path>|
              <attribute path> <op> <value>
              <node path> IN <query>
              <attribute path> <op> <attribute path>|
              <interval path>→ INTERVAL <temporal condition>

<attribute path>::=<attribute label>|
                   <object path>.<edge label>.<attribute path>|
                   <object path>.<wildcard>.<attribute path>|

<temporal path expression>::=
               <edge label>.<object label> → INTERVAL |
               <edge label> → INTERVAL

<temporal condition> ::= <temporal op> <value>|
     <interval op><select path>.<temporal path expression>|
     <interval op><interval values>|
     <temporal condition> AND<temporal condition>|
     <temporal condition> OR<temporal condition>

<op>::= = | >|<|>= | =<| =! =

<temporal op>::= begin | end | begin before | end before |
                 begin after | end after

<interval op>::= precede | overlap | equal
```

The intuitive meaning of the remaining nonterminals is the following:

<label>	a string
<DB label>	a database name
<object label>	a complex node name
<attribute label>	an atomic node name
<edge label>	an edge label
<value>	a constant string
<interval values>	a constant interval

Figure 4.10. The WHERE clause of the **JT-Fragment**

SELECT Professor.HasProperty.PersData.HasPropery.Name
FROM Department
WHERE **EXISTS** Professor.Teaches.Course
AND Professor.Teaches → **INTERVAL**
(begin “01/01/1999”
OR begin before “01/01/1999”)
AND end after “01/01/1999”

This query requires to find in the Department database the name of those Professor objects connected to at least a Course object by means of an edge labeled Teaches, such that its temporal element is composed also by an interval which contains the year "1999". The **EXISTS** clause is needed because this is a semistructured database, thus in principle no constraint imposes that the Teaches edge should end with the node of type Course.

In the TSS-QL query these requirements are specified through paths, which are graphically depicted in Fig. 4.11. Note that the time intervals of the reported objects are generic, and thus replaced with the symbol _, while the interval of the edge Teaches is specified as described in the **WHERE** clause.

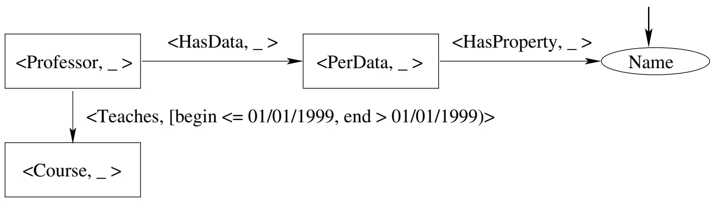


Figure 4.11. Example of the visual query *Find the names of professors who were working in 1999*

The result of this query applied to the database depicted of Fig. 4.1 is Professor Mary Jones. Note that the query finds the names of those professors whose data are structured as the query dictates, i.e., the professors with a *PersData* sub-object, as specified in the **SELECT** clause. As we said before, semistructured data are irregular: some data are missing, as in the case of professor John Smith who does not have a *Course* sub-object, thus does not even satisfy the **EXISTS** clause. It may also happen that a professor does not have a *PersData* sub-object (see Richard Black). Similar concepts be represented by using different types (personal data of professor Richard Black are structured in different way from personal data of professor Mary Jones). In this context flexible path expressions, which allow querying without knowing the structure in advance, are needed, thus path expressions in TSS-QL are built from labels and wildcards as seen in Section 4.4. The same query, composed by using wildcards, is the following:

```
SELECT  Professor.*.Name
  FROM  Department
 WHERE  EXISTS Professor.Teaches.Course
   AND  Professor.Teaches → INTERVAL
        (begin "01/01/1999"
        OR begin before "01/01/1999")
        AND end after "01/01/1999"
```

The graphical version of this query is shown in Fig. 4.12. Note that this query allows us to find the names of professors who were working in 1999, independently of the structure of the personal data.

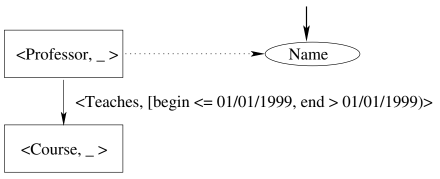


Figure 4.12. Example of the visual query *Find the names of professors who were working in 1999*

Query 2. Find the names of professors who were teaching the "Databases" course during the whole year 1999.

```
SELECT  Professor.HasData.PerData.HasProperty.Name
  FROM  Department
 WHERE  Professor.Teaches.Course.HasProperty.Name = "Databases"
   AND  Professor.Teaches → INTERVAL
        (begin "01/01/1999"
        OR begin before "01/01/1999")
        AND end after "01/01/1999"
```

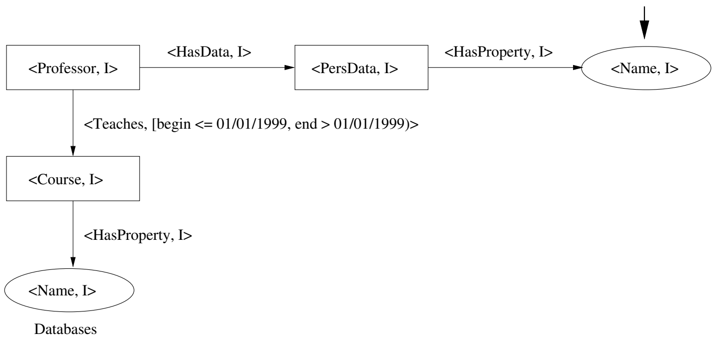


Figure 4.13. Example of visual query *Find the names of professors who were teaching the "Databases" course in 1999*

This query is similar to the previous one, but a restriction on the value of the course name is imposed. In this case we do not have the **EXISTS** condition in the **WHERE** clause because we want to find a precise Course object. The graphical version is in Fig. 4.13, where the name of the Course object is specified below the atomic object Name.

Query 3. Find the current name of the courses which was called "Databases" at some past time.

```
SELECT  Course.HasProperty.Title
  FROM  Faculty
 WHERE  Course.HasProperty.Title → INTERVAL end "now"
   AND  Course IN
SELECT  Course
  FROM  Faculty
 WHERE  Course.HasProperty.Title = "Databases"
   AND  Course.HasProperty.Title → INTERVAL end before "now"
```

In the **WHERE** clause we specify that the interval of the Title property of the course of interest has to end with the constant "now", in this way we assure that only the current name is retrieved.

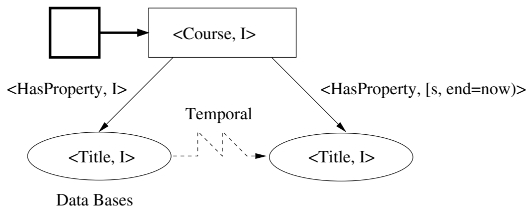


Figure 4.14. Example of query *Find the current name of the Courses which was at some time called "Databases"*

Note that in order to find *all the names* ever assumed by the course called "Databases" we have to write the following query:

```
SELECT  Course.HasProperty.Title
  FROM  Faculty
 WHERE  Course IN
SELECT  Course.HasProperty.Title
  FROM  Faculty
 WHERE  Course.HasProperty.Title = "Databases"
```

4.5 A Graphical Model for User Navigation History

In this section we move from the classical representation of Valid Time to that of Interaction Time. When users browse through a document (for example a hypermedia representation) they choose a particular path in the graph representing semistructured information and in this way they define their order among the visited objects. Following the research lines of Web Usage Mining field, we can create a view depending on each user's choices, that can be useful to personalize data presentation.

A first step in this direction consists of tracing users' navigation history, in order to recognize the habits of each user, e.g. hyperlink selections, time spent on a particular page or number of accesses to a specific information. A second step is using and processing such information, in order to obtain a user model not only based on static declarations of interests and stereotypical descriptions but on actual interactions.

Results on such pre-processing activities about historical analysis can be successively used for improving and customizing site content presentation: in fact the graph-based temporal representation is used not only to keep track of user's preferences, but also to pose all sorts of different queries by means of TSS-QL [77]. We have just seen that the language has a graphical version that allows us to reduce the problem of solving a query on a graph-based model to the problem of performing a *matching* of the "query graph" with the "site graph" as we explained in Chapter 2. Again, as a consequence of the time complexity of this problem, our idea is to find the fragment of our language that can be translated into the logic *CTL* in order to apply model-checking algorithms to solve (part of) TSS-QL queries.

4.5.1 Analyzing User History Navigation

In this section we formalize our approach to keep trace of users' navigation history.

Given a Web site modeled by a semistructured temporal graph G as defined in Definition 4.2.1, in a specific time instant t a user U can interact with the site whose information is represented by the snapshot of G at time t, called $S_t(G)$; in fact $S_t(G)$ stores all the objects and relations of G which are current at the time instant t.

Definition 4.5.1. *Let $G = \langle N, E, r, \ell \rangle$ be a semistructured temporal graph, a* path *in G is a non-empty sequence $p = \langle n_0, n_1, \ldots \rangle$ of nodes s.t.: for all i, $n_i \in N$ and $\langle n_i, n_{i+1} \rangle \in E$.*

Now we define how to keep trace of the activities a user takes with his/her mouse and keyboard while visiting a Web site during a single session.

In the sequel, we shall assume a unique identifier to be associated to each user; in order to justify this assumption, we observe that several technologies are available to this effect, including reading the IP address of the user's machine or the content of a special purpose *cookie ID* managed by the Web server.

A *user session* is the period of time when a user started and ended accessing a Web site. We can assume that a user session is terminated when a user is inactive for more than 30 minutes.

For simplicity, we can also assume that a site does not change during a session, in this way in a specific time instant t a user U can interact with the site whose information is represented only by the snapshot of G at time t, i.e. $S_t(G)$.

Definition 4.5.2. *The* interaction *of a user U in a session S (at time t) with a Web site represented by a semistructured temporal graph G is a pair $\langle p, GiveInfo_U \rangle$ where:*

1. *$p = \langle n_1, n_2, \ldots, n_m \rangle$ is a path in the snapshot $S_t(G)$;*
2. *$GiveInfo_U$ is a labeling function which assigns to each node n in the path p the name of the current session and the interval time the user spends visiting n (called "thinking time interval"). For all i, $GiveInfo_U(n_i) = \langle S, t_{U_i} \rangle$ where S is the current session and t_{U_i} is the "thinking time temporal element" (actually it is composed by a unique interval) of user U on node n_i.*

Remark 4.5.1. Note that in order to avoid duplication of information, we can associate, without loss of generality, the labels related to the name of the session and the name of the user only to the first visited node (i.e. the root of the site graphical representation). We recall here that a Web site is a semistructured temporal graph and thus, it is a rooted graph.

$GiveInfo_U$ can be seen as composed by two single-valued functions, namely $GiveInfoS_U$ and $GiveInfoT_U$, which return respectively the current session and the thinking time interval of each node n in a path p.

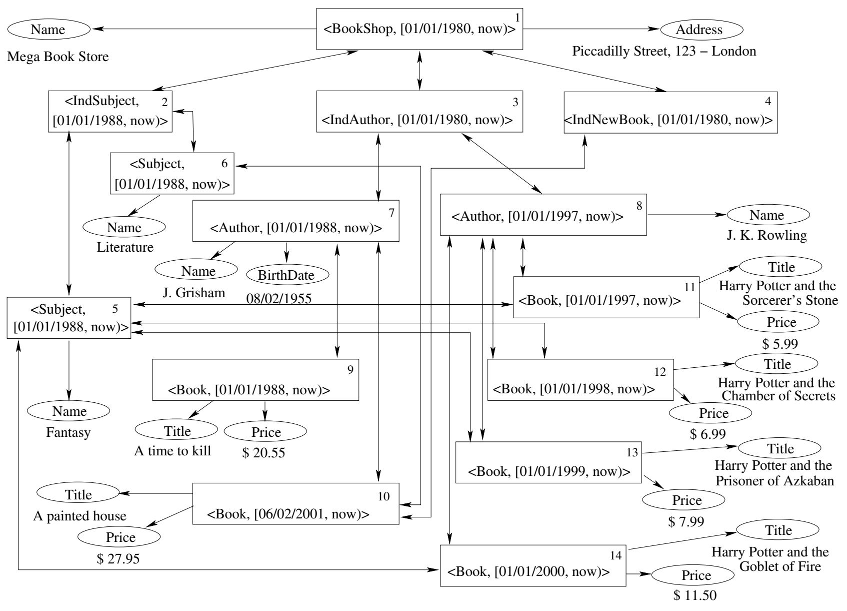
Definition 4.5.3. *The* global interaction *of a user U with a Web site represented by a semistructured temporal graph G is a set of interactions (as introduced in Definition* 4.5.2*).*

In Figure 4.15 we show a graph-based representation of a bookshop site. In this example we suppose that each complex object (drawn as a rectangle) represents a Web page, while atomic objects (drawn as ovals) are the information contained in the site pages. Edges between complex objects represent navigational links; for readability reasons we do not report their labels, which are composed by the name "Link" and the time interval. We omit also the edge labels between complex and atomic objects composed by the name "Has Property" and the time interval.

In Figure 4.16 we represent three interactions of a user who visited the Web Site represented in Figure 4.15. For example, in session A the user starts from the BookShop Page, spending 5 minutes on that page, and then he/she clicks on the Index Page of Subjects, after 2 minutes the user clicks on the page of the "Fantasy" subject, after 3 minutes he/she visits the page of the book titled "Harry Potter and the Prisoner of Azkaban" and so on. For readability reasons in Figure 4.16 and Figure 4.17 we represent only complex objects because they correspond to visited pages (attributes are only information contained in these pages).

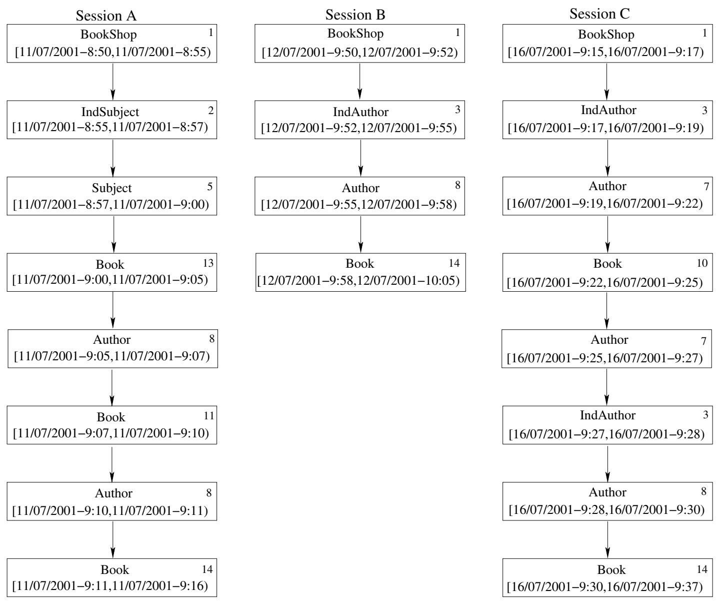
In order to build a history-based user profile, we now analyze the information stored in the "global interaction" of each user.

We introduce a definition to obtain a graph from a global interaction A of a user with a Web site. Intuitively, we construct a labeled graph G_A which



<BookShop, [01/01/1980, now)> 1
Name
Mega Book Store
Address
Piccadilly Street, 123 – London
<IndSubject, [01/01/1988, now)> 2
<IndAuthor, [01/01/1980, now)> 3
<IndNewBook, [01/01/1980, now)> 4
<Subject, [01/01/1988, now)> 6
Name
Literature
<Author, [01/01/1988, now)> 7
Name
J. Grisham
BirthDate
08/02/1955
<Author, [01/01/1997, now)> 8
Name
J. K. Rowling
<Subject, [01/01/1988, now)> 5
Name
Fantasy
<Book, [01/01/1997, now)> 11
Title
Harry Potter and the Sorcerer's Stone
Price
$ 5.99
<Book, [01/01/1988, now)> 9
Title
A time to kill
Price
$ 20.55
<Book, [01/01/1998, now)> 12
Title
Harry Potter and the Chamber of Secrets
Price
$ 6.99
<Book, [01/01/1999, now)> 13
Title
Harry Potter and the Prisoner of Azkaban
Price
$ 7.99
<Book, [06/02/2001, now)> 10
Title
A painted house
Price
$ 27.95
<Book, [01/01/2000, now)> 14
Title
Harry Potter and the Goblet of Fire
Price
$ 11.50

Figure 4.15. A semistructured temporal graph for a bookshop site



Session A
BookShop 1 [11/07/2001–8:50,11/07/2001–8:55)
IndSubject 2 [11/07/2001–8:55,11/07/2001–8:57)
Subject 5 [11/07/2001–8:57,11/07/2001–9:00)
Book 13 [11/07/2001–9:00,11/07/2001–9:05)
Author 8 [11/07/2001–9:05,11/07/2001–9:07)
Book 11 [11/07/2001–9:07,11/07/2001–9:10)
Author 8 [11/07/2001–9:10,11/07/2001–9:11)
Book 14 [11/07/2001–9:11,11/07/2001–9:16)
Session B
BookShop 1 [12/07/2001–9:50,12/07/2001–9:52)
IndAuthor 3 [12/07/2001–9:52,12/07/2001–9:55)
Author 8 [12/07/2001–9:55,12/07/2001–9:58)
Book 14 [12/07/2001–9:58,12/07/2001–10:05)
Session C
BookShop 1 [16/07/2001–9:15,16/07/2001–9:17)
IndAuthor 3 [16/07/2001–9:17,16/07/2001–9:19)
Author 7 [16/07/2001–9:19,16/07/2001–9:22)
Book 10 [16/07/2001–9:22,16/07/2001–9:25)
Author 7 [16/07/2001–9:25,16/07/2001–9:27)
IndAuthor 3 [16/07/2001–9:27,16/07/2001–9:28)
Author 8 [16/07/2001–9:28,16/07/2001–9:30)
Book 14 [16/07/2001–9:30,16/07/2001–9:37)

Figure 4.16. Global interaction composed by three interactions of a user in sessions called A, B, and C

has as nodes the set of visited nodes (i.e. all the nodes in A) and as edges the set of all node pairs $\langle a, b \rangle$ such that in at least a path of A the edge $\langle a, b \rangle$ exists. For each node n of G_A a labeling function *GiveTotalInfo* gives the total thinking time, that is the temporal element composed by the thinking times $GiveInfo_U(n)$ in all the paths where n is present.

Definition 4.5.4. *The* merge G_A *of a global interaction* $A = \{i_1, \ldots, i_k\}$ *over a semistructured temporal graph* G *is obtained as: for all* $j \in \{1, \ldots, k\}$, *let* i_j *be the interaction* $i_j = \langle \langle n_1, \ldots, n_{h_j} \rangle, \mathit{GiveInfo}^j \rangle$.
We obtain a semistructured temporal graph $G_A = \langle N, E, \mathit{GiveTotalInfo} \rangle$ *where:*

– $N = Complex \cup Atomic$ *where* $Complex = \{n_1, \ldots, n_{h_1}\} \cup \ldots \cup \{n_1, \ldots n_{h_k}\}$ *(note that* $\cup$ *is the set union and thus* N *does not have repeated elements) and* $Atomic = \{m \mid \ell_{\mathcal{T}}(m) = atomic \wedge \exists n \in Complex \wedge \langle n, \textit{has-property}, m \rangle$ *is an edge of* $G\}$.
– $\langle a, b \rangle \in E$ *with* $a, b \in N$ *iff there is a path in* A *containing an edge* $\langle a, b \rangle$.
– *For each node* $n \in N$ *the labeling function GiveTotalInfo gives the temporal element related to* n, *i.e. the union of all the thinking time intervals of* n.

Note that nodes and edges preserve their original labels composed by name and type.

Observe that the graph we obtain by applying the procedure in Definition 4.5.4 may be a non-connected graph; for example, it could be the case that the user clicks on different nodes in each session.

The "merge" graph represents the activities (i.e. clicks on the pages of a Web site) that a user takes in a set of sessions. In particular, its set of nodes is composed by all complex objects in the global interaction (i.e. the visited pages) and their attributes (i.e. atomic objects). For readability reasons in Figure 4.17 we represent only complex objects because they correspond to visited pages (attributes are only information contained in these pages).

From this graph it is possible to derive the time spent by the user on different pages. This information may be used as an indication of the user preferences. Unfortunately, since this graph contains cycles, we can derive neither the exact sequence of clicks, i.e. the precise order between different clicks, nor the information about the sessions, because we do not represent it. This last piece of information could give us an idea of the favorite order when navigating a site of interest.

The merge graph we obtain by combining the three interaction paths depicted in Figure 4.16 is shown in Figure 4.17. Note that from the graph it is possible to derive for each node the total thinking time (for example, the favorite page for this user seems to be the page of "Harry Potter and the Goblet of Fire", which has the highest thinking time). However, it is not possible to derive the navigation history of each session because we do not keep trace of session names.

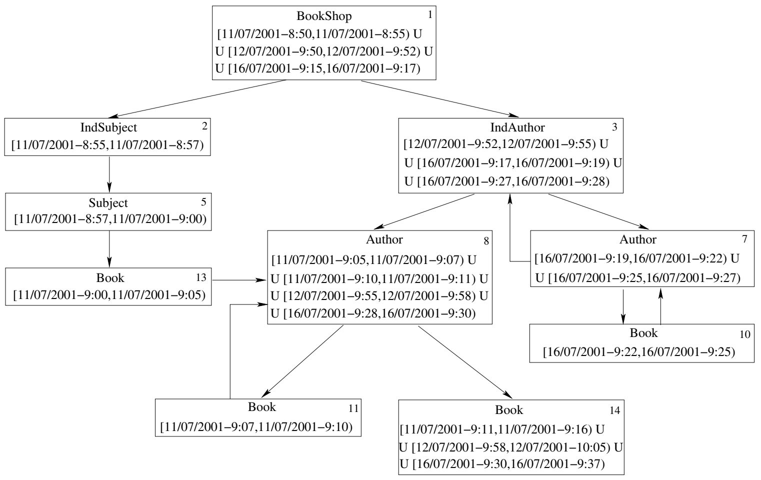

BookShop 1
[11/07/2001–8:50,11/07/2001–8:55) U
U [12/07/2001–9:50,12/07/2001–9:52) U
U [16/07/2001–9:15,16/07/2001–9:17)
IndSubject 2
[11/07/2001–8:55,11/07/2001–8:57)
IndAuthor 3
[12/07/2001–9:52,12/07/2001–9:55) U
U [16/07/2001–9:17,16/07/2001–9:19) U
U [16/07/2001–9:27,16/07/2001–9:28)
Subject 5
[11/07/2001–8:57,11/07/2001–9:00)
Author 8
[11/07/2001–9:05,11/07/2001–9:07) U
U [11/07/2001–9:10,11/07/2001–9:11) U
U [12/07/2001–9:55,12/07/2001–9:58) U
U [16/07/2001–9:28,16/07/2001–9:30)
Author 7
[16/07/2001–9:19,16/07/2001–9:22) U
U [16/07/2001–9:25,16/07/2001–9:27)
Book 13
[11/07/2001–9:00,11/07/2001–9:05)
Book 10
[16/07/2001–9:22,16/07/2001–9:25)
Book 11
[11/07/2001–9:07,11/07/2001–9:10)
Book 14
[11/07/2001–9:11,11/07/2001–9:16) U
U [12/07/2001–9:58,12/07/2001–10:05) U
U [16/07/2001–9:30,16/07/2001–9:37)

Figure 4.17. The "Merge" graph of interactions A, B and C

While in Figure 4.16 we report the global interaction of a particular user, in Figure 4.18 we show the global interaction of a group of users. This representation is useful to analyze the behavior of a set of users (or of a unique user working in more parallel sessions) in order to classify their habits and preferences.

Also for a group of users we can derive a "merge graph" containing the set of visited pages and the total time spent on each page. Note that in this case we can have parallel sessions of different users on the same page, i.e. thinking time intervals may overlap, and thus, in order to have the real total time spent on a page, we do not merge intervals which have a non empty intersection.

4.6 Using the Query Language TSS-QL to Obtain Relevance Information

In order to achieve our goal regarding the customization of WWW sites, we have to elaborate the information on actual interactions of users. Our usage of the temporal data model for user profiling is based on the observation that the time spent by the user on the individual components of a site can be safely assumed as an implicit relevance measure. In fact the information about each individual component of a Web site can be obtained by means of extended TSS-QL queries (see [77, 39] for further details). More in detail, it is possible to query the original Web site graph, the interaction paths and the Merge graph, because they are all semistructured temporal graphs.

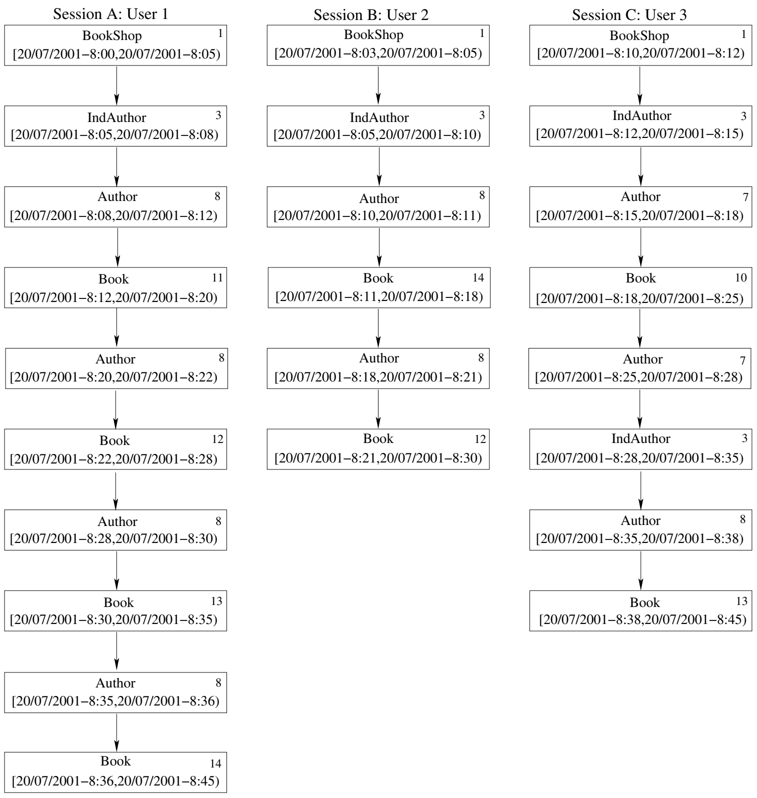

Session A: User 1
BookShop 1 [20/07/2001–8:00,20/07/2001–8:05)
IndAuthor 3 [20/07/2001–8:05,20/07/2001–8:08)
Author 8 [20/07/2001–8:08,20/07/2001–8:12)
Book 11 [20/07/2001–8:12,20/07/2001–8:20)
Author 8 [20/07/2001–8:20,20/07/2001–8:22)
Book 12 [20/07/2001–8:22,20/07/2001–8:28)
Author 8 [20/07/2001–8:28,20/07/2001–8:30)
Book 13 [20/07/2001–8:30,20/07/2001–8:35)
Author 8 [20/07/2001–8:35,20/07/2001–8:36)
Book 14 [20/07/2001–8:36,20/07/2001–8:45)
Session B: User 2
BookShop 1 [20/07/2001–8:03,20/07/2001–8:05)
IndAuthor 3 [20/07/2001–8:05,20/07/2001–8:10)
Author 8 [20/07/2001–8:10,20/07/2001–8:11)
Book 14 [20/07/2001–8:11,20/07/2001–8:18)
Author 8 [20/07/2001–8:18,20/07/2001–8:21)
Book 12 [20/07/2001–8:21,20/07/2001–8:30)
Session C: User 3
BookShop 1 [20/07/2001–8:10,20/07/2001–8:12)
IndAuthor 3 [20/07/2001–8:12,20/07/2001–8:15)
Author 7 [20/07/2001–8:15,20/07/2001–8:18)
Book 10 [20/07/2001–8:18,20/07/2001–8:25)
Author 7 [20/07/2001–8:25,20/07/2001–8:28)
IndAuthor 3 [20/07/2001–8:28,20/07/2001–8:35)
Author 8 [20/07/2001–8:35,20/07/2001–8:38)
Book 13 [20/07/2001–8:38,20/07/2001–8:45)

Figure 4.18. Global interaction of a group of users

Again, we increase the expressive power of TSS-QL by introducing with the **CA-Fragment**, which is shown in Figures 4.20 and 4.21, the possibility to express complex queries with aggregation functions (e.g. MIN, MAX, COUNT).

Now we report some examples of queries useful to extract relevant information on the behavior of users:

1. *Find the book pages reached from John Grisham's page with a click*
   SELECT Author.Book
   FROM Global Interaction
   WHERE Author.Name = "John Grisham"
2. *Find the book pages reached from John Grisham's page with some clicks*
   SELECT Author.*.Book
   FROM Global Interaction
   WHERE Author.Name = "John Grisham"
3. *Find the pages reached from John Grisham's page with a click*
   SELECT Author.Dummy

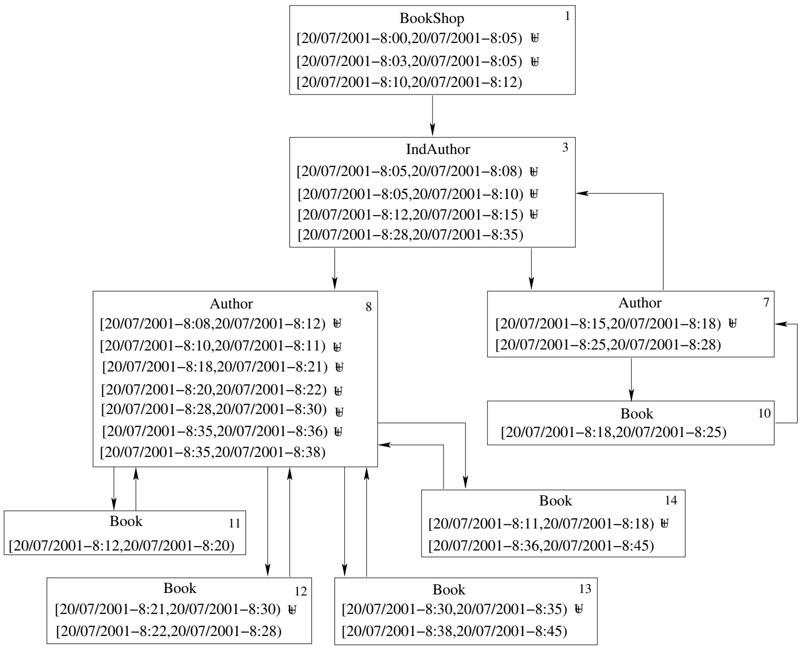


Figure 4.19. The “Merge” graph of a group of users

FROM Global Interaction
WHERE Author.Name = “John Grisham”

4. *Find the users who visited John Grisham’s page as last page*
   SELECT BookShop→ UserID
   FROM Global Interaction
   WHERE Author.Name = “John Grisham”
   EXCEPT
   SELECT Author→ UserID
   FROM Global Interaction
   WHERE Author.Name = “John Grisham” **AND**
   EXISTS Author.Dummy

Note that the word “Dummy” is a placeholder for any node label and the field “UserID” is used to extract the label related to the name of the user from the first visited node (cf. Remark 4.5.1).

4.6.1 Semistructured Temporal Graph as a KTS

In this section we show how to build the *KTS* associated with a semistructured temporal graph (see [46]), in order to apply the technique based on model-checking algorithms to efficiently extract relevant information about user navigation history.

```
<query>::= SELECT [<label> AS] <select list>
           [ FROM <from list> ]
           [ WHERE [<label> AS] <condition> ]
           [ GROUP BY <attribute path expression> ]
           [ EXCEPT <query> ]

<select list>::= <select path>, <select list>|<select path>

<select path>::= <node path>|
                 <attribute t-list>
                 <interval path>→INTERVAL
                 <aggregation path expression>

<node path>::= <node label>|<edge path>.<node label>

<node label>::= <object label>|<attribute label>

<edge path>::= <object label>.<edge label>| <object label>.<wildcard>|
               <object label>.<edge label>.<edge path>|
               <object label>.<wildcard>.<edge path>

<interval path>::= <object label>|<edge label>|<edge path>|
                   <edge path>.<node label>

<object path>::= <object label>|<edge path>.<object label>

<attribute t-list path> ::= <attribute t-list>|
                            <edge path>.<attribute t-list>

<attribute t-list> ::= <attribute label>.Temporal.<attribute label>|
                       <attribute label> # <attribute label>|
                       <attribute label> # <attribute t-list>

<wildcard>::= ∗ | ?

<from list>::= <DB label>, <from list>| <DB label>

<aggregation path expression> ::= <aggregation operator>
                                  (<simple path expression>)

<simple path expression> ::= <node label>|
        <object label>.<edge label>.<simple path expression>|
        <object label>.<wildcard>.<simple path expression>|
        <label>

<attribute path expression> ::= <attribute label>|
          <object label>.<edge label>.<select path>

<aggregation operator>::= COUNT | MAX | MIN | AVG

<from list>::= <DB label>, <from list>|
               <DB label>
```

Figure 4.20. The SELECT-FROM clauses of the **CA-Fragment**

```
<condition>::= <predicate>
               <condition> OR <condition>|
               <condition> AND <condition>|

<predicate>::=EXISTS <node path>|
              <attribute path> <op> <value>
              <node path > IN <query>
              <attribute path> <op> <attribute path>|
              <interval path> → INTERVAL <temporal condition>
              <label> = ANY <query >

<attribute path>::=<attribute label>|
                   <object path>.<edge label>.<attribute path>|
                   <object path>.<wildcard>.<attribute path>|

<temporal path expression>::= <edge label>.<object label> → INTERVAL |
                              <edge label> → INTERVAL

<temporal condition> ::= <temporal op> <value>|
    <interval op><select path>.<temporal path expression>|
    <interval op><interval values>|
    <temporal condition> AND<temporal condition>|
    <temporal condition> OR<temporal condition>

<op>::= = | >|<|>= | =<| =! =

<temporal op>::= begin | end | begin before | end before |
                 begin after | end after

<interval op>::= precede | overlap | equal
```

The intuitive meaning of the remaining nonterminals is the following:

<label>	a string
<DB label>	a database name
<object label>	a complex node name
<attribute label>	an atomic node name
<edge label>	an edge label
<value>	a constant string
<interval values>	a constant interval

Figure 4.21. The WHERE clause of the **CA-Fragment**

Definition 4.6.1. *Let $G = \langle N, E, r, \ell \rangle$ be a semistructured temporal graph; we define the* KTS $\mathcal{K}_G = \langle \Sigma_G, \mathcal{A}ct_G, \mathcal{R}_G, \mathcal{I}_G \rangle$ *over* Π_G *as follows:*

- *The set of atomic propositions Π_G is the set of all the node labels of G.*
- *The set of states is Σ_G is the set of nodes N.*
- *The set of actions $\mathcal{A}ct_G$ includes all the edge labels. In order to capture the notion of* before *we also add to $\mathcal{A}ct_G$ actions for the inverse relations and for the negation of all the relations introduced. Moreover, we can assume, for each state s corresponding to a complex object with no outgoing edge in*

E to another complex object, to add a self-loop edge labeled by the special action $\circlearrowleft$ that is not in $\mathcal{A}ct_G$. We recall that only complex objects represent site pages.

– *The transition relation $\mathcal{R}_G$ contains all the pairs of nodes which are connected by means of a labeled edge (actually, edge labels are the actions defined in the previous point).*
– *For the interpretation function $\mathcal{I}_G$ we can note that in each state the only formulae that hold are the atoms related to the labels and* true*.*

The queries we have shown in Section 4.4 can be respectively translated in the following *CTL* formulae.

1. *Find the book pages reached from John Grisham's page with a click*
   $\mathsf{Book} \wedge \mathsf{EX}_{\mathsf{Link}^{-1}}(\mathsf{Author} \wedge \mathsf{EX}_{\mathsf{HasProperty}}(\mathsf{John\ Grisham}))$
   This *CTL* formula is true in all Book states which are reachable by means of a click from an Author state which possesses a property John Grisham.
2. *Find the book pages reached from John Grisham's page with some clicks*
   $\mathsf{Book} \wedge \mathsf{EF}_{\mathsf{Link}^{-1}}(\mathsf{Author} \wedge \mathsf{EX}_{\mathsf{HasProperty}}(\mathsf{John\ Grisham}))$
   This formula is very similar to the previous one, but the path from the Book state to the Author state has an arbitrarily length.
3. *Find the pages reached from John Grisham's page with a click*
   $\mathsf{true} \wedge \mathsf{EX}_{\mathsf{Link}^{-1}}(\mathsf{Author} \wedge \mathsf{EX}_{\mathsf{HasProperty}}(\mathsf{John\ Grisham}))$
   This *CTL* formula is true in any state which is reachable by means of a click from an Author state which possesses the property John Grisham. Note that the atom true holds in each state.
4. *Find the users who visited John Grisham's page as last page*
   $\mathsf{EG}_{\circlearrowleft}(\mathsf{Author} \wedge \mathsf{EX}_{\mathsf{HasProperty}}(\mathsf{John\ Grisham}))$
   This *CTL* formula is true in all Author leaf states labeled "John Grisham" and thus, with this formula we do not actually infer the exact name of the users satisfying this requirement (i.e. we do not deal with the UserID property of the Author object).

In Figure 4.23 we report the translation in the NuSMV format of Session *B* depicted in Fig. 4.16 (for readability reasons we do not include all atomic objects), whereas the translation of the four queries we have just proposed is in Figure 4.22:

All the previous queries are contained in the portion of TSS-QL that can be solved in polynomial time on the size of the model and the *CTL* formula, by using a model-checker (see Section 4.6.2 for more details). We have called this portion of TSS-QL the "Path-Fragment" (i.e. P-Fragment in Section 4.4.1) because of the nature of information we can extract.

Now we introduce some other queries expressible with the portion of TSS-QL which uses aggregation operators, called the "Complex Aggregation Fragment" (i.e. CA-Fragment). Note that this fragment does not admit a translation into *CTL* because the model-checker cannot elaborate any retrieved

```
SPEC
    ℓ = Book & EX(ℓ = Author &
    EX(ℓ = HasProperty & EX(ℓ = JohnGrisham)))
SPEC
    ℓ = Book & EF(ℓ = Author &
    EX(ℓ = HasProperty & EX(ℓ = JohnGrisham)))
SPEC
    true & EX(ℓ = Author &
    EX(ℓ = HasProperty & EX(ℓ = JohnGrisham)))
SPEC
    EG(ℓ = Author &
    EX(ℓ = HasProperty & EX(ℓ = JohnGrisham)))
```

Figure 4.22. NuSMV format of CTL queries

```
MODULE main
VAR
s   : {n1, n3, n8, n14, n15, n16};
lab : {BookShop, IndAuthor, Author, Book, HasProperty,
      J.K.Rowling};
ASSIGN
    init(s) := n1;
    next(s) := case
        s = n1  : n3;
        s = n3  : {n8, n1};
        s = n8  : {n14, n15, n3};
        s = n14 : {n14, n8};
        s = n15 : {n16, n8};
        s = n16 : n15;
    esac;
DEFINE
    ℓ := case
        s = n1  : BookShip;
        s = n3  : IndAuthor;
        s = n8  : Author;
        s = n14 : Book;
        s = n15 : HasProperty;
        s = n16 : J.K.Rowling;
    esac;
```

Figure 4.23. NuSMV format of a user session

information. In order to solve with the model-checking based approach these queries, we should develop a system that can analyze the data previously extracted by a model-checker.

1. *Find the favorite book page of a user*
   SELECT Book.Name, **MAX**(clicks)
   FROM Global Interaction
   WHERE clicks = **ANY** (**SELECT COUNT**(Book)
   FROM Global Interaction
   GROUP BY Book.Name)
2. *Find the number of times that a user clicks from the Index of Author to the J. K. Rowling's page*
   SELECT COUNT(IndAuthor.Author)
   FROM Global Interaction
   WHERE Author.Name = "J. K. Rowling"
3. *Find the date when a user visited John Grisham's page*
   SELECT Author→INTERVAL
   FROM Merge
   WHERE Author.Name = "John Grisham"
   The keyword "INTERVAL" is used to extract the time interval of John Grisham's page.

4.6.2 Complexity Results on TSS-QL Fragments

In this section we summarize the results on semantics presented in this paper, and discuss the main complexity results about TSS-QL fragments.

The **P-Fragment** can be correctly translated into *CTL*, as depicted in Figure 4.24, because it includes only simple path queries without join conditions and operators applied on temporal elements (we remind that queries with paths including temporal edges are contained in this fragment). Intuitively, in order to translate semistructured temporal graphs into Kripke Transition Systems, we have considered as local properties of each state the set of labels of the corresponding node in the semistructured graph. Moreover, the translation of a query into a *CTL* formula is obtained by using the modal operators X or F for expressing respectively paths of length one or paths of an arbitrarily length. The F operator of *CTL* allows direct calculation of transitive closures that is used to gave semantics of queries containing wildcard. Thus, if we remain within the P-Fragment, by using the model-checker we can also compute queries containing wildcards.

With the **JT-Fragment** we add 1) comparison between values and 2) general temporal operators to the language.

1. Propositional temporal logics cannot express constraints that force two states with the same set of local properties to be a unique state. In fact, the modal logic semantics forces a bisimulation equivalence between subgraphs fulfilling the same formula. This allows us to say that two nodes with the same properties are equivalent but not to require that they be the same node. For this reason comparisons of values belonging

to different paths (i.e. "join-conditions") do not have always an exact translation into *CTL*.

2. The problem of general temporal conditions, i.e. conditions on the temporal element of an object or relationship in a semistructured temporal graph, is that model-checking provides an algorithm to find *all* the states which satisfy a formula and these states are characterized by the set of their local properties. Thus, a model-checker cannot extract and analyze *only a subset* of these properties, for example only the temporal element. In fact, with our approach we can correctly check temporal conditions with the equality operator, i.e. the model-checker can only tests if a given temporal element is a property of a state.

In the same way, the **CA-Fragment** includes some queries without a translation into *CTL*. In this case the reason is that a model-checker cannot elaborate any previously retrieved information and thus, cannot deal with typical aggregation functions of SQL, such as MIN, MAX, and COUNT.

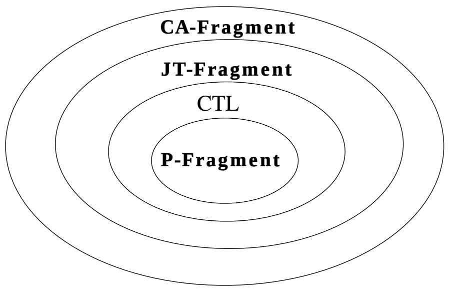


Figure 4.24. Complexity of TSS-QL Fragments with respect to *CTL*

5. Related Works

In this thesis we have focused our attention on three languages for semistructured data: G-Log, $\mathbb{W}$, and TS-QL. We have deeply studied the relationships between them by considering their expressive power and by applying techniques coming from the model-checking field to efficiently solve the data retrieval problem.

5.1 Semantics Aspects of Query Languages

Several articles have pointed out the need for data models and query languages to integrate heterogeneous data sources and a lot of proposals have emerged in the past years. We can cite LOREL [66], UnQL [12], GraphLog [33], G-Log [81], WG-Log [32], WebSQL [67], WebOQL [7] and, StruQL [50].

Among them, a graphical language that shows many similarities with G-Log is GraphLog [33], a declarative language with graph-based data model and rules. GraphLog is a deterministic language intended for the relational model, it is not Turing-complete, and allows no sequences of rules. Moreover, GraphLog query graphs are acyclic and the queries, which require patterns to be present in (or absent from) the database graph, are supposed to extend databases only with new edges (i.e. queries can only define new relations). Conversely, G-Log was originally developed as a language and data model for *complex objects with identity* [1], and in its full form it is Turing complete (see [83]). The structure and the meaning of queries in the two languages are rather similar, but cycles are allowed in G-Log, and G-Log rules enable the user to extend the databases both with entities and relations.

Similarities can also be found between our approach to give semantics for graphical languages and previous works on UnQL [12], where the notion of bisimulation is used for investigating query decomposition. However, differences between G-Log and UnQL are quite deep. For instance, when assigning semantics to the language basic blocks, we allow information to be located in graph nodes, while UnQL locates information on edges; more importantly, G-Log queries are written directly in the graph formalism, while UnQL describes data instances graphically, and the query language of UnQL is SQL-like. Moreover, G-Log allows to express cyclic information and queries, and achieves its high expressive power by allowing a fully user-controlled non-determinism.

Anyway, keeping in mind these differences, the results of the work shown in this thesis about the bisimulation-based semantics, can be also applied to GraphLog and UnQL, and in general to any graphical language whose main aim is to query and transform a graph-based data model by using graph-based rules.

5.2 Efficient Query Retrieval

In the semistructured database field we find also many attempts to give a graph-based representation of first-order logic or to translate graphical queries into logical formulae (see for example [81, 33, 10, 61]). More recently, other proposals have studied the relationships between databases and modal logics (see [20, 16, 75, 76, 43]), but none of them attack the problem of efficiently solve queries with model-checking algorithms.

A work which introduces temporal-logic queries for model understanding and model checking is [20]. The author proposes an approach for inferring properties of models as a special case of evaluating temporal-logic queries by using a model checker. In [20] a query is a *CTL* formula with a special symbol '?', called placeholder, which appears exactly once. Given a model M, the semantics of a query is a proposition p such that replacing '?' with p in the query, it results a formula that holds in M and is as strong as possible. Moreover, the author defines the subset of *CTL* queries that are guaranteed to be evaluated in time linear in the size of the model and in the length of the query. This approach could be used in the semistructured data field to understand properties of semistructured instances and to iteratively gain knowledge about them. In fact, it is well known that semistructured data [2] has some structure but it is often irregular and thus, an efficient approach based on model checking techniques could be applied in order to derive a possible representation, or an approximation of the structure, of a data source for driving users in the query formulation process.

Consider for example the $\mathbb{W}$-instance of Fig. 3.2. The evaluation of the query $\mathsf{AG}_{\mathcal{A}ct}?$ on that $\mathbb{W}$-graph, gives as result $p = \text{Teacher} \vee \text{Course} \vee \text{Databases} \vee \text{Student} \vee \text{Smith} \vee 40 \vee 37$, i.e. an indication of all the labels included in the model of Fig. 3.2. Whereas the evaluation of the query $\text{Teacher} \wedge \mathsf{EX}_{\mathcal{A}ct}?$ on the same model gives as result all the successors of Teacher states, i.e. $p = \text{Course} \vee 40 \vee 37$.

In [16] the authors define a query language for semistructured data that is based on the ambient logic [17], which is a modal logic originally proposed to describe the structural and computational properties of distributed and mobile computations. The ambient logic is designed to describe properties of labeled trees and thus, it is suitable to be used like a query language for semistructured data. The proposed query language, called Tree Query Language (TQL), is characterized by the fact that a matching expression is a logic expression combining matching and logical operators. It would be interesting to compare TQL with the temporal logic *CTL* in term of expressive power and to show the possibility to apply polynomial model-checking algorithms also to this SQL-like query language, which is essentially based on a modal logic.

In [75] the authors suggest to query data using 'query automata', that are particular cases of two-ways deterministic tree automata. They show the equivalence with formula expressible in second-order monadic logic. In [76] they identifies subclasses of formulae/automata that allow efficient algorithms. A future work is to determine the exact relationships with our approach.

An attempt to apply model-checking in the WWW field is introduced in [43], where model-checking is used for the analysis of the World Wide Web, i.e. to help with the detection of errors in the structure and connectivity of Web pages. In this context, the model (a Kripke Structure) of the Web is not known in advance, but can only be gradually explored by using model-checking techniques. In particular, in [43] a constructive μ-calculus is defined in order to effectively apply algorithms for the analysis of the Web, where some operations of the ordinary μ-calculus [49], such as the computation of sets of predecessor Web pages and the computation of the greatest fixpoint, are not possible because of the graph structure of the Web itself.

5.3 Temporal Models and Query Languages for Semistructured Data

The possibility to use an efficient algorithm for the *CTL* temporal logic in order to solve static queries on semistructured databases has increased our interest in studying real temporal properties of semistructured data as well.

In the past years temporal database management was a promising field of research that is now covered in textbooks (see for example [99, 3, 48]) and several data models, which usually extended relational data models [29, 74, 86], were proposed in order to consider the time-varying nature of data. However, relational data models [29, 74, 86] are flat and thus unsuitable for semistructured data. In this thesis we have introduced a general temporal data model based on labeled graphs, which is built upon concepts from temporal databases and semistructured data models, to store time-varying semistructured information.

Other works on temporal aspects for semistructured data are [15, 6, 47, 21, 22].

In [15] the authors describe a data warehousing system for managing selected useful historical web information. To support temporal queries and data analysis they define a temporal web data model which is able to capture web documents together with their updating history into temporal web tables. The graph-based model proposed in [15] is designed to support the representation of Web sites and is not general enough to be applied to semistructured databases: each node in the graph denotes a web page, and each directed link denotes a hyperlink between two web pages. Moreover, in order to keep trace of historical information the authors attach a temporal attribute to each node to denote the valid time interval of its web document instances. The valid time is an interval enclosed by two time points, called *last-modified time* and *downloading time*. This interval indicates the period within which the web document captured by the node remains unchanged. The last-modified time indicates the latest time a web object is changed while the downloading time represents the date and time at which a web page departs the HTTP server from which the web document is accessed. In our opinion, this time interval does not represent the valid time of a given web object, because it is not connected to the real content of the web page. Moreover, the time interval does not represent the transaction time of a web object in a site and seems to be the transaction time of the system developed to create the Warehouse.

Another work which focuses on the definition and the implementation of a logical data model for representing histories of XML documents is [6]. The proposed model is an extension of the XPath [95] data model with a label on edges regarding the valid time concept. In particular, the model is used for XML documents and thus, edges represent only the containment relation between Web pages or portion of pages. Our idea is to study temporal properties on a more general temporal model suitable to represent generic semistructured data (not only Web sites).

In [47] the authors propose a semistructured model where annotations on edge labels allow to record information about updates. In particular, they consider annotations about transaction time, valid time and kind of update. They also define temporal operators supporting coalescing, collapsing and slicing operations. In the proposed model, like in the OEM one [80], the edge labels do not actually represent relationships between objects, they are used to make nodes self-describing in the sense that a node is characterized by the sequences of labels on paths through the graph that lead to the node itself. Once again, we choose to consider a more general and expressive model based on labeled graphs where both edges and nodes have properties (expressed by means of labels). In this way we can also consider semantic relationships between nodes connected by paths; consequently information about transaction time have to been added to both nodes and edges.

Another two works which address the problem of representing and querying changes in semistructured data are [21, 22]. The context taken into consideration in these works is similar to ours and thus we deeply describe the main differences and the advantages of our model.

5.3.1 Comparison with the DOEM Model

In [21, 22] the authors propose a model based on the Object Exchange Model (OEM) (see [80]), a simple graph-based data model, with objects as nodes and object-subobject relationships represented as labeled arcs. Change operations (i.e. node insertion, update of node values, addition and removal of labeled arcs) are represented in DOEM (for Delta-OEM) by using *annotations* on the nodes and arcs of an OEM graph. Intuitively, annotations are the representation of the history of nodes and edges. To implement DOEM and Chorel they use a method that encodes DOEM databases as OEM databases and translates Chorel queries into equivalent Lorel queries over the OEM encoding. With this implementation they allow annotations on a DOEM graph D to be attached to the nodes and edges of a OEM database O_D. In this way, for each object o of D there is a complex object o' in O_D with some subobjects, which are required in order to represent the history of the node o, its old values or relationships with other nodes. This method causes a growth in the number of nodes of the final graph over which it is possible to apply the query.

The main aim of our approach is to overcome this problem: on one hand, we adopt a model which reduces the amount of annotations needed for information representation; on the other hand, we plan to effectively apply model-checking techniques as a way to solve queries which are able to take into account static and dynamic aspects of semistructured data. When the properties belong to some classes of formulae the problem of finding if a model possesses a given property is decidable and algorithms running in *linear time* on both the *sizes* of the *model* and the *formula* can be employed [27]. Consequently it is important to keep the size of semistructured databases as small as possible.

For example, in Fig. 5.1 we show the DOEM graph corresponding to a portion of the instance in Fig. 4.1 of Chapter 4: note that only edges are labeled with the name of destination objects. Actually, in both OEM and DOEM edges represent containment relations between objects, and do not allow to indicate different semantic relationships between objects, for this reason our model has a higher expressive power. Annotations are represented in DOEM by rectangles containing the name of the change operation and the date when the modification occurred.

In Fig. 5.2 we report the OEM database corresponding to DOEM graph in Fig. 5.1. Note that in this OEM graph annotations have been encoded into complex objects storing the history of change operations by means of connected sub-objects. To highlight the difference between our temporal data model and DOEM we consider an update operation to an atomic object, because it clearly distinguishes the two approaches by considering the number of added nodes. For example, to perform the update on the Title of the Course previously called Data Bases, into Web Data Bases in the semistructured graph of Fig. 4.1, we only add an atomic object with the new name, whereas in

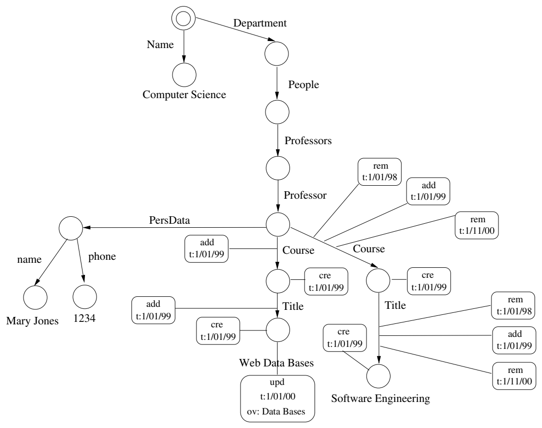


Figure 5.1. A DOEM semistructured database

DOEM the number of added nodes is shown in the dashed region of Fig. 5.2. Note that we add the minimum piece of information to keep trace of the update.

Our graphical model allows to represent and retrieve temporal information about semistructured data according to at least two interpretations of the time concept that are Valid Time and Interaction Time. The former notion is widely used in classic temporal database field, whereas the latter is related to the time a user spends while visiting Web sites.

Monitoring and analyzing how the Web is used is nowadays an active area of research in both the academic and commercial worlds. Web Usage Mining or Analysis field can be defined as being interested in patterns of behavior for Web users.

In particular, personalizing the Web experience for a user is a crucial task of many Web-based applications such as e-commerce applications. In fact, making dynamic and personalized recommendations to a Web user, based on their profile and not only on general usage behavior, is very attractive in many applications. Web Usage Mining is an excellent approach for achieving this goal, as shown in [92].

Some existing systems, such as WebWatcher [58], Letizia [63], WebPersonalizer [72], have all concentrated on providing Web Site personalization based on usage information. WebWatcher [58] "observes" a user as they browse the Web and identifies links that are potentially interesting to the user. Letizia [63] is a client side agent that searches for pages similar to ones that the user has already visited. The WebPersonalizer [72] page recommendations are based on clusters of pages found from the server log for a site. The system recommends pages from clusters that most closely match the current session.

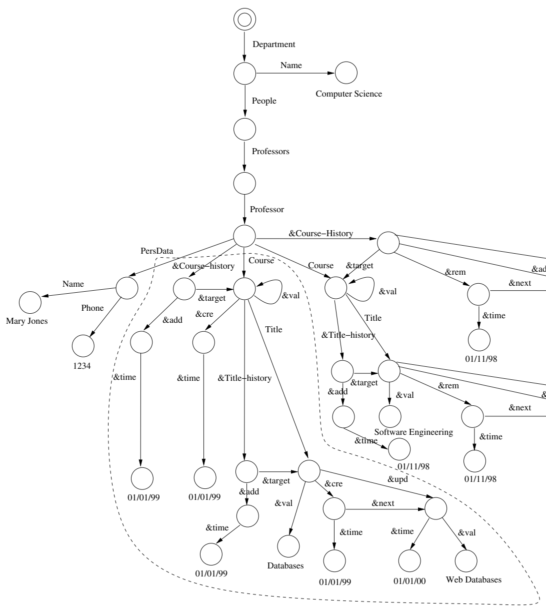


Figure 5.2. A OEM semistructured database

Web Usage Mining techniques introduced in [23, 35, 98, 91] have been used for discovering frequent item sets, association rules, clusters of similar pages and users, sequential patterns, and to perform path analysis.

As we said before, analysis of how users are visiting a site is crucial for optimizing the structure of the Web site and thus, there are a lot of reasons in pre-processing data for mining before running the mining algorithms in order to address some important issues (see for example [34]). These include developing a model of access log data and developing techniques to clean/filter the raw data in order to eliminate outliners and/or irrelevant items. According to [36] a data model can be constructed by considering the following information:

– **Content**: the real data stored in the Web pages.
– **Structure**: data which describe the organization of the content.
– **Usage**: data that describe the usage pattern of Web pages, such as, for example, page references, time and date of accesses.
– **User Profile**: data that provide personal information about users of the Web site.

Moreover, before any mining is done on Web usage data, sequences of page references must be grouped into logical units representing user sessions. A *user session* is the set of page references made by a user during a single visit to a site and is the smallest transaction to consider for grouping interesting page references as proposed in [23, 35].

In the context of pre-processing tasks, our proposal is to use a database approach based on our graphical data model to store user interactions and to apply either a SQL-like query language or the model-checking based approach to retrieve information of interest about user navigation history. As a consequence, the possibility to apply the same model in different contexts allows us to assert that a generic temporal model (i.e. a model that is not strictly application-dependent) may be useful to represent more interpretations of the "time concept".

6. Conclusion

In this work we have proposed a method to provide suitable graph-based semantics to languages for semistructured data, supporting both data structure variability and topological similarities between queries and document structures. A suite of operational semantics based on the notion of bisimulation has been introduced both at the concrete level (instances) and at the abstract level (schemata).

In the context of semistructured and WWW data efficiency is a primary requirement and thus, complexity results on subgraph-bisimulation problem have stimulated our interest in investigating alternative approaches to solve (graphical) queries on databases that do not present a rigid structure. The main idea of the work is to effectively solve the data retrieval problem for semistructured data by using techniques and algorithms coming from the model-checking field. In particular, we have considered three languages for semistructured data: G-Log, $\mathbb{W}$, and TS-QL. We have first studied G-Log, because it is a general language that offers the expressive power of logic, the modeling power of object-oriented DBs, and the representation power of graphs for drawing instances, schemata and graphs. $\mathbb{W}$ is more or less the subset of G-Log that has a natural translation into the modal logic setting. Finally, in order to express temporal queries we have introduced a SQL-like language that allows to express complex conditions on the time-varying nature of semistructured data. However, we have shown that it is quite natural to extend graphical languages (e.g. the G-Log language) to express temporal properties with labeled graphs.

In the recent years a lot of attention has been given by the research community to the study of methods for representing and querying semistructured data. In our opinion, the approach we proposed in this work presents several benefits:

- first of all, we benefit from the cross-fertilization of methods originally designed in quite different research areas (Databases, Logics, and Model-Checking) and with our proposal we try to establish a relationship between them.
- Moreover, from a formal point of view, the work is also interesting for classifying polynomial subclasses of the subgraph isomorphism/bisimulation problems.

- The approach based on model-checking algorithms is general enough to be applied not only to our graphical language, but to the main existing languages for semistructured data as well, and thus, can be further developed to effectively solve the data retrieval problem.
- The nature of the *CTL* logic has allowed us to extend our main idea to the temporal database setting. First of all, we have shown that a general graph-based model can be used to represent both static and dynamic aspects of semistructured data. Moreover, we have applied temporal formulae to express real temporal queries and model-checking to solve them.
- To conclude, we have concentrated our attention not only on the classical notion of the "time concept", as it is deeply known in the database field, but we have further investigated on the possibility to consider more than one dynamic aspects of semistructured data.

The subjects of this work represent and introductory study from which many extensions in different directions may start. In particular, we plan to investigate on the following aspects:

- The *graph-formula* translation based on model-checking techniques could be extended to a wider family of graphs including join conditions. It might also be interesting to find the exact relationships between the set of *CTL** formulae and the set of formulae needed for cyclic queries.
- We have implemented the method on the model-checker NuSMV. However, for an effective implementation on WWW data some form of heuristic (e.g. check part of the domain name) must be applied in order to cut the search space for accessible data (that can be all the web). This issue is studied in [43] in a slight different context.
- Another future direction is to mix this technique with constraint-based data retrieval (where temporal formulae can be used directly).
- In the temporal database field, we plan to provide a formal semantics for our SQL-like query language and to extend it in order to include the possibility to express updates. Moreover, we will investigate on the exact translation between the SQL-like version of TS-QL and its graphical counterpart, by considering the expressive power and limits of the two approaches.
- We plan also to investigate on the problem of using the temporal sequence of all navigation actions taken by a user, as an input to a site personalization tool. Another aim of our database approach is to find techniques, coming from the Web Mining field, to use previous history to reduce the amount of browsing actions needed to achieve a user's search goal.

References

1. A. Abiteboul and P. Kanellakis. Object Identity as a Query Language Primitive. *Journal of the ACM*, 45(5):798–842, 1998.
2. S. Abiteboul. Querying Semi-Structured Data. In *Proceedings of the International Conference on Database Theory*, volume 1186 of *Lecture Notes in Computer Science*, pages 262–275, 1997.
3. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley Publishing Company, 1995.
4. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.
5. P. Aczel. *Non-well-founded sets.*, volume 14 of *Lecture Notes, Center for the Study of Language and Information.* Stanford, 1988.
6. T. Amagasa, M. Yoshikawa, and S. Uemura. A Data Model for Temporal XML Documents. In *Database and Expert Systems Applications, 11th International Conference, DEXA 2000*, volume 1873 of *Lecture Notes in Computer Science*, pages 334–344. Springer-Verlag, Berlin, 2000.
7. G. Arocena and A. Mendelzon. WebOQL: Restructuring documents, databases, and webs. In *Proceedings of the 14th International Conference on Data Engineering*, pages 336–350. IEEE Computer Society Press, 1998.
8. R. Barrett, P.P. Maglio, and D.C. Kellem. How to Personalize the Web. In *CHI 97 Electronic Publications: Papers*, San Jose, Canada, 1997.
9. J. Borges and M. Levene. Data Mining of Users Navigation Patterns. In *Web Usage Analysis and User Profiling*, volume 1836 of *Lecture Notes in Computer Science*, pages 92–111, 1999.
10. D. Bryce and R. Hull. SNAP: A Graphical-based schema manager. In *Proceedings of the Second International Conference on Data Engineering*, pages 151–164. IEEE Computer Society, 1986.
11. P. Buneman. Semistructured data. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Priciples of Database Systems (PODS)*, pages 117–121. ACM Press, 1997.
12. P. Buneman, S. B. Davidson, G. G. Hillebrand, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 505–516. ACM Press, 1996.
13. P. Buneman, S. B. Davidson, G. G. Hillebrand, and D. Suciu. Adding structure to unstructured data. In *Database Theory - ICDT'97, Sixth International Conference Proceedings*, volume 1186 of *Lecture Notes in Computer Science*, pages 336–350, 1997.
14. P. J. Cameron. First-Order Logic. In L. W. Beineke and R. J. Wilson, editors, *Graph Connections. Relationships Between Graph Theory and other Areas of Mathematics.* Clarendon Press, Oxford, 1997.

15. Y. Cao, E. P. Lim, and W. K. Ng. On Warehousing Historical Web Information. In *Proceedings of Conceptual Modeling - ER 2000, 19th International Conference on Conceptual Modeling*, volume 1920 of *Lecture Notes in Computer Science*, pages 253–266. Springer-Verlag, Berlin, 2000.
16. L. Cardelli and G. Ghelli. A Query Language Based on the Ambient Logic. In *Proceedings of the 10th European Symposium on Programming (ESOP 2001)*, volume 2028 of *Lecture Notes in Computer Science*, pages 1–22, 2001.
17. L. Cardelli and A. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.
18. S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: a Graphical Language for Querying and Restructuring XML Documents. *Computer Network*, 31(11–16):1171–1187, 1999.
19. D. Chamberlin, J. Rubie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Proceedings of the World Wide Web and Databases, Third International Workshop WebDB 2000, Selected Papers*, Lecture Notes in Computer Science, pages 1–25. Springer-Verlag, Berlin, 2001.
20. W. Chan. Temporal-logic Queries. In *Proceedings of 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 450–463, 2000.
21. S. S. Chawathe, S. Abiteboul, and J. Widom. Representing and Querying Changes in Semistructured Data. In *Proceedings of the Fourteenth International Conference on Data Engineering*, pages 4–13. IEEE Computer Society, 1998.
22. S. S. Chawathe, S. Abiteboul, and J. Widom. Managing historical semistructured data. *Theory and Practice of Object Systems*, 5(3):143–162, 1999.
23. M. S. Chen, J. S. Park, and P. S. Yu. Data mining for path traversal patterns in a web environment. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 385–392, 1996.
24. J. Chomicki. Temporal Integrity Constraints in Relational Databases. *IEEE Data Engineering Bulletin*, pages 33–37, 1994. Special Issue on Database Constraint Management.
25. J. Chomicki. Temporal Query Languages: a Survey. In *Proceedings of the International Conference on Temporal Logic*, pages 506–534. Springer–Verlag, 1994.
26. A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.
27. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent System using Temporal Logic Specification. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
28. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. Mit Press, 1999.
29. J. Clifford and A. Croker. The historical relational data model (HRDM) and algebra based on lifespans. In *Proceedings of the International Conference on Data Engineering*, pages 528–537. IEEE Computer Society, 1987.
30. J. Clifford and A. U. Tansel. On an algebra for historical relational databases: Two views. In *Proceedings of the 1985 ACM Sigmod International Conference on Management of Data*, pages 247–265. ACM Press, 1985.
31. S. Comai. *Graphical Query Languages for Semi-structured Information*. PhD thesis, Politecnico di Milano, 1999.
32. S. Comai, E. Damiani, R. Posenato, and L. Tanca. A Schema-based Approach to Modeling and Querying WWW Data. In *Proceedings of the Third International Conference on Flexible Query Answering Systems FQAS'98*, volume 1495 of *Lecture Notes in Computer Science*, pages 110–125, 1998.

33. M. P. Consens and A. O. Mendelzon. Graphlog: a Visual Formalism for Real Life Recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 404–416. ACM Press, 1990.
34. R. Cooley, B. Mobasher, and J. Srivastava. Web Mining: Information and Pattern Discovery on the World Wide Web. In *Proceedings of the 9th IEEE International Conference on Tools with Artificial Intelligence*, 1997.
35. R. Cooley, B. Mobasher, and J. Srivastava. Data preparation for mining world wide web browsing patterns. *Knowledge and Information System*, 1(1):5–32, 1999.
36. R. Cooley, P. N. Tan, and J. Srivastava. Discovery of Interesting Usage Patterns from Web Data. To appear in Springer-Verlag LNCS/LNAI series, 2000.
37. A. Cortesi, A. Dovier, E. Quintarelli, and L. Tanca. Operational and Abstract Semantics of a Query Language for Semi–Structured Information. To appear in Theoretical Computer Science, 2002.
38. P. Cousot and R. Cousot. Abstract Interpretation: a unified framework for static analysis of programs by costruction of approximation of fixpoints. In *Fourth ACM Principles of Programming Logic*, pages 83–94, 1977.
39. E. Damiani, B. Oliboni, E. Quintarelli, and L. Tanca. Modeling users' navigation history. In *Workshop on Intelligent Techniques for Web Personalisation. In Seventeenth International Joint Conference on Artificial Intelligence*, pages 7–13, 2001.
40. E. Damiani, B. Oliboni, L. Tanca, and D. Veronese. Using WG-Log Schemata to Represent Semistructured Data. In *Proceedings of the Eighth Working Conference on Database Semantics (DS-8)*, pages 331–349, 1999.
41. E. Damiani and L. Tanca. Semantic Approches to Structuring and Querying Web Sites. In *Procedings of 7th IFIP Working Conference on Database Semantics (DS-97)*, 1997.
42. E. Damiani and L. Tanca. Blind Queries to XML Data. In *Proceedings of 11th International Conference (DEXA 2000)*, volume 1873 of *Lecture Notes in Computer Science*, pages 345–356, 2000.
43. L. de Alfaro. Model Checking the World Wide Web. In *Proceedings of Computer Aided Verification, 13th International Conference*, volume 2102 of *Lecture Notes in Computer Science*, pages 337–349, 2001.
44. A. Dovier and C. Piazza. The Subgraph Bisimulation Problem and its Complexity. Technical Report 27/00, Università di Udine. Dipartimento di Matematica e Informatica, November 2000.
45. A. Dovier, C. Piazza, and A. Policriti. A Fast Bisimulation Algorithm. In *13th International Conference on Computer Aided Verification, CAV 2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, 2001.
46. A. Dovier and E. Quintarelli. Model-Checking Based Data Retrieval. In *Proceedings of 8th Biennial Workshop on Data Bases and Programming Languages (DBPL'01)*, pages 29–45, 2001.
47. C. E. Dyreson, M. H. Böhlen, and C. S. Jensen. Capturing and Querying Multiple Aspects of Semistructured Data. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases*, pages 290–301. Morgan Kaufmann, 1999.
48. R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, 1994.
49. E. A. Emerson. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, Mass., 1990.

50. M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a web-site management system. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(3):4–11, 1997.
51. G. Filè, R. Giacobazzi, and F. Ranzato. A Unifying View on Abstract Domain Design. In *ACM Computing Surveys, 28(2)*, pages 333–336, 1996.
52. D. Florescu, A. Levy, and A. Mendelzon. Database Techniques for the World Wide Web: A Survey. *SIGMOD Record*, 27(3):59–74, 1998.
53. M. R. Garey and D. S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.
54. R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of the Twenty-Third International Conference on Very Large Data Bases*, pages 436–445. Morgan Kaufmann, 1997.
55. C. S. Jensen. *Temporal Database Management.* PhD thesis, Aalborg University, 2000.
56. C. S. Jensen and C. E. Dyreson. A Consensus Glossary of Temporal Database Concepts - February 1998 Version. In *Temporal Databases: Research and Practice*, volume 1399 of *Lecture Notes in Computer Science*, pages 36–405. 1998.
57. C. S. Jensen, C. E. Dyreson, M. H. Bohlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. J. Hayes, S. Jajodia, W. Kafer, N. Kline, N. A. Lorentzos, Y. G. Mitsopoulos, A. Montanari, D. A. Nonen, E. Peressi, B. Pernici, J. F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. U. Tansel, P. Tiberio, and G. Wiederhold. The consensus glossary of temporal database concepts - february 1998 version. In *Temporal Databases: Research and Practice. (the book grow out of a Dagstuhl Seminar, June 23-27, 1997)*, volume 1399 of *Lecture Notes in Computer Science*, pages 367–405. Springer, 1998.
58. T. Joachims, D. Freitag, and T. M. Mitchell. Web Watcher: A Tour Guide for the World Wide Web. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97*, volume 1, pages 770–777. Morgan Kaufmann, 1997.
59. S. Jones and P. J. Mason. Handling the Time Dimension in a Data Base. In *Proceedings of the International Conference on Data Bases*, pages 65–83, 1980.
60. P. C. Kanellakis and S. A. Smolka. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. *Information and Computation*, 86(1):43–68, 1990.
61. H. Kim, H. Korth, and A. Silberschatz. Picasso: A graphical query language. *Software-Practice and Experience*, 18(1):169–203, 1988.
62. P. B. Ladkin. *The Logic of Time Representation.* PhD thesis, University of California at Berkeley, 1987.
63. H. Lieberman. Letizia: an Agent That Assists Web Browsing. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95*, volume 1, pages 924–929. Morgan Kaufmann, 1995.
64. A. Lisitsa and V. Sazanov. Bounded Hyperset Theory and Web–like Data Bases. In *Computational Logic and Proof Theory, 5th Kurt Gödel Colloquium*, volume 1289 of *Lecture Notes in Computer Science*, pages 172–185, 1997.
65. P. P. Maglio and R. Barrett. How to Build Modeling Agents to Support Web Searchers. In *User Modeling: Proceedings of the Sixth User Modeling International Conference*, pages 5–16, 1997.
66. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: a database management system for semistructured data. *SIGMOD Record*, 23(3):54–66, 1997.

67. A. Mendelzon, G. Mihaila, and T. Milo. Querying the World Wide Web. In *Proceedings of the Fourth Conference on Parallel and Distributed Information Systems*, pages 80–91. IEEE Computer Society, 1996.
68. R. Milner. An algebraic definition of simulation between programs. In *Second International Joint Conference on Artificial Intelligence*, pages 481–489. William Kaufmann, 1971.
69. R. Milner. A Calculus of Communicating System. In *Lecture Notes in Computer Science*, volume 92. Springer-Verlag, Berlin, 1980.
70. R. Milner. Operational and Algebraic Semantics of Concurrent Processes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 19, pages 1203–1241. Elsevier Science Publishers B.V., 1990.
71. D. Mladenic and M. Grobelnik. Efficient text categorization. In *Text Mining Workshop on ECML-98*, 1998.
72. B. Mobasher, R. Cooley, and J. Srivastava. Creating adaptive web sites through usage-based clustering of urls. In *Proceedings of Knowledge and Data Engineering Workshop*, 1999.
73. M. Müller-Olm, D. Schmidt, and B. Steffen. Model-Checking. A Tutorial Introduction. In *Proceedings of the International Static Analysis Symposium* (*SAS '99*), volume 1694 of *Lecture Notes in Computer Science*, pages 330–354. Springer-Verlag, Berlin, 1999.
74. S. B. Navathe and R. Ahmed. A temporal relational model and a query language. *Information Sciences*, 49(1–3):147–175, 1989.
75. F. Neven and T. Schwentick. Query Automata. In *Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 205–214. ACM Press, 1999.
76. F. Neven and T. Schwentick. Expressive and Efficient Pattern Languages for Tree-Structured Data. In *Proceedings of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 145–156. ACM Press, 2000.
77. B. Oliboni, E. Quintarelli, and L. Tanca. Temporal aspects of semistructured data. In *Proceedings of The Eighth International Symposium on Temporal Representation and Reasoning (TIME-01)*, pages 119–127. IEEE Computer Society, 2001.
78. B. Oliboni and L. Tanca. Querying XML specified WWW sites: links and recursion in XML-GL. In *Proceedings of Sixth International Conference on Rules and Objects in Databases*, volume 1861 of *Lecture Notes in Computer Science*, pages 1167–1181, 2000.
79. R. Paige and R. E. Tarjan. Three partition refinement algorithms. In *SIAM Journal on Computing*, volume 16, pages 973–989, 1987.
80. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260. IEEE Computer Society, 1995.
81. J. Paredaens, P. Peelman, and L. Tanca. G–Log: A Declarative Graphical Query Language. *IEEE Transaction on Knowledge and Data Engineering*, 7(3):436–453, 1995.
82. D. Park. Concurrency and automata on infinite sequences. In *Lecture Notes in Computer Science*, volume 104, pages 167–183. Springer-Verlag, Berlin, 1980.
83. P. Peelman. *G–Log: a deductive language for a graph–based data model.* PhD thesis, Antwerpen University, 1993.
84. D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying Semistructured Heterogeneus Inforrmation. In *Proceedings of the International*

Conference on Deductive and Object-Oriented Databases, volume 1013 of *Lecture Notes in Computer Science*, pages 319–344, 1995.

85. N. L. Sarda. HSQL: A historical query language. In *Temporal Databases: Theory, Design, and Implementation*, pages 110–140. Benjamin/Cummings, 1993.
86. R. Snodgrass. The Temporal Query Language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, 1987.
87. R. Snodgrass. Temporal Databases. In *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space, International Conference GIS - From Space to Territory: Theories and Methods of Spatio-Temporal Reasoning*, volume 639 of *Lecture Notes in Computer Science*, pages 22–64, 1992.
88. R. Snodgrass. TSQL2 Language Specification. *SIGMOD Record*, 23(1):65–86, 1994.
89. R. Snodgrass. A TSQL2 Tutorial. *SIGMOD Record*, 23(3):27–34, 1994.
90. R. Snodgrass and I. Ahn. A taxonomy of time in databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 236–246. ACM Press, 1985.
91. M. Spiliopoulou and L. Faulstich. WUM - A Tool for WWW Ulitization Analysis. In *The World Wide Web and Databases, International Workshop WebDB'98, Selected Papers*, volume 1590 of *Lecture Notes in Computer Science*, pages 184–203, 1996.
92. J. Srivastava, R. Cooley, M. Deshpande, and P. N. Tan. Web Usage Mining: Discovery and Applications of Usage Patterns from Web Data. *SIGKDD Explorations*, 1(2):12–23, 2000.
93. A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings, 1993.
94. J. van Benthem. *Modal Correspondence Theory.* PhD thesis, Universiteit van Amsterdam, Instituut voor Logica en Grondslagenonderzoek van Exacte Wetenschappen, 1978.
95. World Wide Web Consortium. Namespaces in XML. http://www.w3.org/TR/REC-xml-names/. W3C Reccomendation 14 January 1999.
96. World Wide Web Consortium. XML Path Language (XPath) version 1.0. http://www.w3.org/TR/xpath.html. W3C Reccomendation 16 November 1999.
97. A. Wolski, J. Kuha, T. Luukkanen, and A. Pesonen. Design of RapidBase - An Active Measurement Database System. In *Proceedings of International Database Engineering and Applications Symposium (IDEAS 2000)*, pages 75–82. IEEE Computer Society Press, 2000.
98. O. R. Zaiane, M. Xin, and J. Han. Discovering Web Access Patterns and Trends by Applying OLAP and Data Mining Technology on Web Logs. In *Proceedings of the IEEE Forum on Reasearch and Technology Advances in Digital Libraries, ADL '98*, pages 19–29. IEEE Computer Society, 1998.
99. C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems.* Morgan Kaufmann, 1997.
100. N. Zin and M. Levene. Constructing Web Views from Automated Navigation Sessions. In *ACM Digital Library Workshop on Organizing Web Space 1999 (WOWS)*, pages 54–58, 1999.