

PROBABILISTIC REASONING FOR COMPLEX SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Avrom J. Pfeffer
December 1999

© Copyright 2000 by Avrom J. Pfeffer
All Rights Reserved

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Daphne Koller
Computer Science Department
Stanford University
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

John Mitchell
Computer Science Department
Stanford University

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Stuart Russell
Computer Science Department
University of California, Berkeley

Approved for the University Committee on Graduate Studies:

To Debby Gelber, for giving me the strength to write this.

Preface

Reasoning under uncertainty is a central issue in artificial intelligence. Real-world agents must deal with noisy sensor information, non-deterministic effects of actions, and unpredictable exogenous events. Probabilistic reasoning methods, and Bayesian networks (BNs) in particular, have emerged as an effective and principled method for reasoning under uncertainty. BNs exploit conditional independence relationships to create natural and compact domain models, thereby supporting useful reasoning patterns, and providing effective probabilistic inference and learning algorithms. However, BNs are inherently limited by their attribute-based nature, making it difficult to apply them to large, complex domains.

This thesis addresses the issue of representing and reasoning about probabilistic models of complex systems. We believe that the key to reasoning effectively about complex systems is to provide a language that supports the expression of system structure. We present a powerful object-based representation language, that integrates logical and probabilistic representations. Our language provides the ability to create structured, modular probabilistic models. The language maintains the key advantages of BNs, exploiting conditional independence relationships. In addition, it is capable of representing other aspects of system structure not represented in BNs. In particular, it supports the decomposition of complex systems into weakly interacting subsystems, and the reuse of models for many different components of a system.

Another key benefit of our language is that it is very flexible. The same probabilistic representations can be applied in many different situations, with very different configurations. In fact, our language can even represent uncertainty over the system configuration itself, and integrate that uncertainty directly with uncertainty over the

basic properties of objects in the system. Our framework also supports the representation of powerful recursive probability models.

We present inference algorithms for our language that exploit the structure that can be expressed in it — not only the conditional independence structure normally exploited by BN algorithms, but also encapsulation, reuse of computation and symmetry resulting from the object-based representation. We describe an implemented system that supports representation and reasoning with models in our language, and provide experimental results demonstrating the advantages of exploiting structure in inference.

Acknowledgements

First of all, I would like to thank Daphne Koller, who has been an advisor, mentor, co-author and friend. This work has been a joint venture from the start, and I could not have asked for a better collaborator. I have learned so many things from Daphne: how to choose interesting problems and tackle them by the horns; how to grasp the big picture while working on the nitty-gritty details; most of all, how to push myself beyond my self-perceived limitations to discover capabilities I did not know I possessed. Thank you so much for being a wonderful advisor!

This work could not have happened without the involvement of a number of co-authors. Alon Levy gave me the opportunity to spend a summer at AT&T Research, where we hatched some of the initial ideas that made their way into this thesis. While at AT&T I also met David McAllester, whose sparkling intelligence and creativity have been exciting and inspirational. Nir Friedman has been a pleasure to work with. His technical excellence and tremendous grasp of the issues have provided me with a model to emulate. Lise Getoor has been more than a collaborator and co-author, she is also a wonderful friend and office-mate. We have talked about so many things over the last few years, I cannot begin to list the many things I have learned from her. I especially appreciate her ability to integrate ideas from interesting and surprising directions. Over the last two years I have had the joy of working with Brian Milch and Ken Takusagawa. Brian's wisdom, insight, and intuitive sense of what is right have often guided us in the right direction, while Ken's originality and creativity have turned insoluble problems into solved problems.

Michael Stonebraker first introduced me to computer science research. I am tremendously grateful to him for giving me the opportunity to work in his group

while I was an undergraduate at Berkeley. Stuart Russell was my first AI teacher, and is responsible for my collaboration with Daphne. He has been an excellent source of advice and support over the years. Stuart has also provided me with valuable comments and feedback on this thesis. I am also grateful to other people who have read and commented on this work: John Mitchell, Ross Shachter and Richard Fikes. I am also grateful to Joe Hellerstein, Manfred Jaeger, Ron Parr, Mehran Sahami and Yoav Shoham for many stimulating discussions.

I would like to thank Uri Lerner, Lise Getoor, Xavier Boyen, Eric Bauer, Dragomir Angelov, Ben Taskar and Barbara Engelhardt for developing the Phrog system; Brian Milch, Ken Takusagawa and Ryan Shaw for their help in developing SPOOK; Simon Tong, Barbara Engelhardt and Urszula Chajewska for help with knowledge engineering; Suzanne Mahoney, KC Ng, Geoff Woodward and Tod Levitt of IET Inc. for the original battlespace models; and Jim Rice, who has been a tremendous help in getting SPOOK to integrate with Ontolingua.

I would also like to thank my different sources of funding: a Stanford School of Engineering Fellowship, a National Science Foundation Graduate Research Fellowship, ONR contract N66001-97-C-8554 under DARPA's HPKB program, and DARPA contract DACA76-93-C-0025 under subcontract to Information Extraction and Transport, Inc.

I am very grateful to the many friends both at Stanford and elsewhere who have made the last few years so enjoyable. I would like to thank the Quail gang — Eyal Amir, Urszula Chajewska, Patrick Doyle, Lise Getoor, Pedrito Maynard-Reid, and Sunil Vemuri — for turning the qualifying exam into a social opportunity. I am grateful to Greg Davidson, Jim Frazin, Bob Givan, Miriam Lewis, Doug Mandell, Fabio Rojas, and Beth and Eric Zuckerman, for enriching my life with their presence. I would also like to mention Hannah Ben-Hanan, who died so tragically and will always be missed.

To my mother: your constant love and encouragement over the years have been a tremendous blessing. You always knew I would grow up to be an absent-minded professor! To my brothers, sisters, nieces, nephews, uncles, aunts and cousins: It is wonderful to be part of such a large and close-knit family. You have provided me

with fantastic support and community.

Above all I would like to thank Debby Gelber. You have come into my life and turned it upside down. You have helped me dig deep inside many times, kept me going, and given me something to live for beyond this thesis. I dedicate this thesis to you.

Contents

Preface	v
Acknowledgements	vii
1 Introduction	1
1.1 Probabilistic Knowledge Representation	1
1.2 Summary of Contributions	4
1.3 Thesis Overview	6
2 Possible Worlds	8
2.1 Attribute-Based Models	9
2.2 Relational Models	10
3 Bayesian Networks	20
3.1 Introduction	20
3.2 Probability Distributions	21
3.2.1 Conditional probabilities, independence, conditional independence	22
3.3 Definition of Bayesian Networks	25
3.4 Bayesian Network Semantics	28
3.5 Conditional Independence and d-Separation	31
3.6 Bayesian network reasoning	34
3.7 Inference	36
3.8 Conclusion	47

4	Object-oriented Bayesian Networks	49
4.1	Introduction	49
4.1.1	Hierarchical Systems	53
4.2	Hierarchical Relational Models	55
4.2.1	Passing Information Between Objects	57
4.2.2	Hierarchical Worlds	60
4.2.3	The Space of Possible Worlds	66
4.3	Specifying the probability model	69
4.4	Semantics	73
4.4.1	Generative Process Semantics	74
4.4.2	Equivalent BN	78
4.4.3	Semantics of a class probability model	81
4.5	Structured Inference	87
4.5.1	Interfaces and Encapsulation	87
4.5.2	Structured Variable Elimination	90
4.5.3	Reuse of Inference	94
4.5.4	Complexity	98
4.5.5	Discussion	106
4.6	Working with OOBNs	111
4.6.1	Defining Class and Subclass Models	111
4.6.2	Abstraction and Refinement	115
4.7	Conclusion	116
5	Relational Probability Models	118
5.1	Introduction	118
5.2	Basic Language Definition	121
5.3	Probability Model and Acyclicity	130
5.3.1	Generative Semantics	130
5.4	Semantics	137
5.4.1	Generative Semantics	137
5.4.2	Probability Measures	140

5.4.3	Measure-Theoretic Semantics for RPMs	141
5.5	Inference	145
5.6	Discussion	153
5.6.1	Integration with OOBNS	153
5.6.2	Isomorphic Worlds	156
5.6.3	Conclusion	156
6	Structural Uncertainty	159
6.1	Introduction	159
6.2	Multi-Valued Attributes	160
6.2.1	Language	160
6.2.2	Semantics	164
6.2.3	Inference with Quantifiers	168
6.3	Closed World Models	172
6.4	Number Uncertainty	176
6.4.1	Possible Worlds	177
6.4.2	Semantics	181
6.4.3	Inference	189
6.5	Reference Uncertainty	191
6.5.1	Instance-Level Reference Uncertainty	191
6.5.2	Inference With Reference Uncertainty	194
6.5.3	Class-Level Reference Uncertainty	199
6.5.4	Enumerated Classes and the Ace of Spades Problem	203
6.5.5	Type Uncertainty	205
6.6	Discussion and Possible Extensions	206
7	Recursive Probability Models	208
7.1	Introduction and Examples	208
7.2	Language Definition	216
7.3	Measure-Theoretic Semantics	218
7.4	Approximate Inference	227
7.5	Structured Approximation Algorithms	232

7.5.1	Iterative SVE	232
7.5.2	Analysis	239
7.5.3	ISVE and Fixed-Point Equations	247
7.6	Conclusion	250
8	Implementation and Applications	252
8.1	The SPOOK System	252
8.2	Example: Military Situation Awareness	257
8.3	Experimental Results	266
8.4	Example: Computer System Diagnosis	269
8.5	Example: Modeling a University	272
8.6	Discussion	276
9	Related Work	279
9.1	Axiomatic Approaches	279
9.2	Model-Based Approaches	281
9.2.1	Structure Bayesian Networks	281
9.2.2	Knowledge-Based Model Construction	282
9.2.3	Network Fragments	285
9.2.4	Other Model-Based Approaches	286
9.3	Miscellaneous	287
10	Conclusion and Future Work	289
10.0.1	Summary	289
10.0.2	Future Work	291
10.0.3	Conclusion	293
	Bibliography	295

List of Tables

List of Figures

3.1	A simple Bayesian network.	26
3.2	Conditional probability tables for the example network.	27
3.3	Initial set of factors for Example 3.7.5	40
3.4	Graphs for computation of Example 3.7.5.	44
3.5	The CPCS network.	46
4.1	Four levels of hierarchy in an OOBN model of a computer system. . .	73
4.2	Flat BN equivalent of Hard Drive object	80
4.3	Interface of Drive-Mechanism object within Hard-Drive	88
4.4	Bounds from Theorems 4.5.11 and 4.5.12 are not tight.	103
4.5	Decomposable interfaces.	108
5.1	(a) Infinite and (b) cyclic dependency models.	133
6.1	Relevant number variables.	185
6.2	Induced graph for decomposed CPF with reference uncertainty. . . .	199
7.1	Self-contained sets: (a) Normal case. (b) Pathological case.	219
7.2	Illustrative figure for Lemma 7.3.7.	222
8.1	The SPOOK system architecture.	253
8.2	(a) SCUD Battalion Bayesian network. (b) SCUD Battery Bayesian Network.	259
8.3	SCUD battalion network with repeated substructures circled.	260

8.4	(a) SA2 Battalion Bayesian network, (b) SA3 Battalion Bayesian network.	261
8.5	SCUD-Battalion class model.	263
8.6	Class hierarchy of basic units in the battlespace model.	264
8.7	Comparison of unstructured and structured inference algorithms.	267
8.8	Comparison of naive and combinatoric approaches to inference with quantifiers.	269
8.9	A Bayesian network describing a single student taking a single course.	273
8.10	(a) BN for single student in two courses. (b) BN for two students in single course.	274

Chapter 1

Introduction

1.1 Probabilistic Knowledge Representation

In designing agents to operate in the real world, one cannot avoid dealing with the issue of uncertainty. Uncertainty crops up in a variety of ways. An agent's sensors typically give it limited information about the world, and the information it does receive is often noisy. The effects of an agent's actions are often non-deterministic. Also, an agent often has to consider unpredictable events that are completely outside its control.

Probability theory provides a sound mathematical basis for reasoning under uncertainty. It has a number of nice properties. It allows one to envision multiple possible states of the world with different degrees of likelihood. It allows one to incorporate multiple pieces of evidence obtained from various sensors in a coherent way, using the principle of Bayesian conditioning. In combination with decision theory, it allows one to make optimal decisions that lead to the maximum expected utility. Also, statistics provides excellent tools for learning probabilistic models from data.

If we want to use probability theory as the foundation for reasoning under uncertainty, we need to address three fundamental issues. First, we need a representation language for describing probabilistic models. The language should allow us to build compact, natural models, and allow us to represent as much domain structure as

possible. Second, we need an algorithm that will allow us to perform efficient probabilistic inference. That is, given a model described in our language, and a set of observations, we should be able to compute various interesting probabilities in a reasonable amount of time. Finally, we need algorithms that will allow us to learn models in our language from data.

This thesis is concerned with representation languages and inference algorithms for probabilistic reasoning about complex systems. The types of applications we are interested in include reasoning about complex situations, such as a battle scenario; diagnosis of large-scale systems, such as an airplane or satellite; monitoring dynamic systems, such as a nuclear reactor; and controlling agents in complex environments. These systems are characterized both by their complexity, in that they consist of many interacting components, and by the fact that there is usually a good deal of uncertainty about the state of the system. We need probabilistic representation languages and inference algorithms that are capable of handling complex scenarios such as these. We also need languages for learning probabilistic models of complex systems from data. This thesis focuses on the representation and inference issues, with the understanding that once these issues have been tackled, learning algorithms can be developed on top of the representation and inference algorithms. Indeed, we have already done some work on learning the types of models described in this thesis [56, 26].

Over the past fifteen years or so, *Bayesian networks (BNs)* [79] have emerged as the leading technology for probabilistic reasoning. In a BN, a system is characterized by some set of attributes, and a state of the system is an assignment of values to each of the attributes. A network consists of a directed acyclic graph over the attributes, and a local probability model for each attribute. The graph structure specifies conditional independence relationships that hold between different attributes, while a local probability model encodes the conditional probability distribution over the values of an attribute given the values of its parents in the graph. Between them, the graph structure and local models encode a probability distribution over all the possible states of the system.

The Bayesian network representation is both compact and natural. The conditional independence relationships asserted by the graphical structure are exploited to allow the complete probability distribution to be defined in terms of a small number of local conditional probabilities. Furthermore, the graph structure can be exploited to support effective probabilistic inference and learning algorithms. BNs, therefore, successfully address the three issues discussed above, at least for systems that can be conveniently characterized by a set of attributes.

However, BNs are inherently limited by their attribute-based nature. It is very difficult to model a complex system using a BN. The basic problem is size — complex systems may have many thousands of variables, and trying to construct such a large network is simply not feasible. Imagine trying to model a battlefield situation — there may be hundreds of units connected to each other in various ways, with each unit described by a number of variables. In addition, systems may have many different configurations, as, for example, a battlespace with different collections of units. We do not want to have to construct a different BN for each possible configuration of the system. Furthermore, we often have uncertainty about the configuration itself. For example, we do not generally know exactly which enemy units are in a battlespace.

For another example, consider trying to model an airplane. An airplane has a huge number of components; modeling each variable associated with each component explicitly, and describing all the dependencies between these variables in a single network, is simply not feasible. It would be far better to describe the properties of the different types of components in an airplane in a general way. In addition, many of the components of an airplane are interchangeable. We should be able to replace a component with a similar component without having to completely reconstruct the model.

From the first days of artificial intelligence, first-order logic was proposed as a foundation for knowledge representation due to its modularity and generality [67]. By allowing general statements that can be applied to many different entities, first-order logic allows the knowledge needed to describe a large system to be represented in a compact manner. Also, because the representation is described in terms of domain entities and the relationships between them, the same knowledge can be applied to

systems with different configurations. However, logical representations do not support probabilistic reasoning, and are inadequate for dealing with uncertainty.

1.2 Summary of Contributions

It seems that what we need is a language that combines the generality and modularity of logical representations, with the ability of Bayesian networks to represent probabilistic models. In this thesis we develop such a language. Our language represents a complex system in a structured manner, in terms of entities in the system and the relationships between them. Probabilistic knowledge is encoded in our language in the form of probability models associated with objects in the domain. The object probability models resemble BN models; a graph structure encodes conditional independencies that hold between properties of an object, and each property has its own local probability model. The probability model for an object describes how each property of the object depends on other properties of the object and on properties of related objects. Probability models are associated with classes of objects, allowing us to reapply the same knowledge to many different instances.

It is vital to make sure that our probabilistic representation language has coherent semantics in terms of a probability distribution or probability measure over possible worlds. The basic approach to achieving this is to imagine a knowledge base as specifying a generative stochastic model, that randomly generates possible worlds. Using this intuition, we can imagine a structured process that generates a world with relational structure, rather than a simple process that generates values for a set of attributes. We provide formal semantics for our language that matches this intuitive idea.

We also develop inference algorithms for our language that exploit the different types of system structure that can be represented in the language. Our algorithms exploit the conditional independence structure in the same way as BN inference algorithms. Our algorithms also utilize the object structure to organize inference, exploiting the fact that the interactions between different objects in a system can often be mediated by a small number of variables. In addition, our algorithms use the fact

that many objects share the same probability model to reuse inference between the different objects.

By developing an object-based probabilistic representation language, we obtain a powerful, expressive language, that is capable of describing rich and interesting domains, and also supports efficient inference that allows us to draw useful and interesting conclusions. The key point is that our language is not only expressive, but also structured. It is our firm belief that expressive languages are good, provided that they allow useful structure in the domain to be represented explicitly. Structure is vital both for helping the model designer construct a good model, and for supporting efficient, structure-exploiting inference algorithms. Our language provides a number of features that allow the structure of a domain to be represented and exploited:

- Different properties of a system are localized within distinct subsystems, exploiting the fact that the different subsystems only interact weakly. Our inference algorithms exploit the fact that many properties can be encapsulated within a single subsystem.
- The language allows general probabilistic models to be provided for entire classes of objects, exploiting the fact that systems often contain many objects of the same kind. Our inference algorithms exploit this feature by reasoning on a general level wherever possible.
- The knowledge representation is separated into a set of class probability models, describing the general behavior of objects, and statements describing the particular configuration of a system. The probability models remain fixed as the system configuration changes. This approach provides a smooth and flexible way of creating probability models for systems with variable configurations.
- The language provides the ability to express uncertainty about the configuration of a system itself. Furthermore, uncertainty over system configuration is thoroughly integrated into the probabilistic models. This integration allows us to apply the same techniques to modeling uncertainty over system configuration

that we use for modeling uncertainty over basic system properties. In particular, we are able to exploit conditional independence between different aspects of system configuration.

- The language allows an object to depend on other objects in an aggregate manner. Such aggregate dependencies introduce symmetry into a model; the same aggregate effect can be achieved in many different ways, all of which have the same effect on the dependent object. Our inference algorithm is able to detect and exploit this symmetry.

1.3 Thesis Overview

In the next chapter, we describe the possible worlds over which our probability models are defined. We first describe attribute-based worlds, and then describe possible worlds with relational structure. In Chapter 3, we review Bayesian networks, which define probability distributions over attribute-based worlds.

The new contributions of this thesis begin in Chapter 4, in which we present *Object-oriented Bayesian Networks (OOBNs)*, which allow us to specify probabilistic models for hierarchically structured systems. The representation and inference are specifically designed to take advantage of the hierarchical system structure.

Beginning with Chapter 5, we consider probabilistic models for more general relational systems. We present *relational probability models (RPMs)*, which integrate a Bayesian network-like probabilistic representation language with traditional relational representations. In particular, RPMs allow us to describe probability models for systems involving many objects that are related to each other in multiple ways. We show how the semantics and inference algorithms for OOBNs can be extended to deal with these more general systems.

The language of relational probability models is extended in Chapter 6 to allow an object to depend on related objects in an aggregate manner. We also introduce *structural uncertainty*, which allows us to represent uncertainty about the relational

structure of a system, and incorporate that uncertainty directly into the class probability models.

In Chapter 7 we consider probability models that may contain infinite chains of influences between different objects. These *recursive probability models* provide a powerful extension to relational probability models. We show that the semantics of RPMs generalizes naturally to these recursive models, and present an object-based iterative approximation algorithm for these models.

In Chapter 8 we describe the implemented SPOOK system for representing and reasoning with relational probability models. We also present three example applications, to military situation awareness, computer system diagnosis, and modeling a university. We present experimental results that demonstrate the advantages of exploiting domain structure in inference. We describe other approaches to integrating probabilistic and logical representations in Chapter 9, and in Chapter 10 we conclude.

Some of the material in this thesis has appeared previously in conference papers. Chapter 4 is based on [57], while Chapters 5 and 6 are largely based on [58] and [82]. Chapter 7 is largely new, but a preliminary version of these ideas appeared in [55]. The implementation and military example in Chapter 8 were described in [82]. Other papers which contain material related to this thesis are [54, 56, 27, 26].

Chapter 2

Possible Worlds

We want a mechanism for dealing with uncertainty that allows us to envision multiple different states of the world as being possible, to specify how likely each of the different possible states is, to update our belief about the state of the world based on our observations about the world, to incorporate multiple sources of evidence in a coherent way, and to determine our degree of belief in some particular property of the world, based on our general beliefs about the world and on our observations.

A fundamental principle of our approach to knowledge representation is to represent our knowledge in the form of a *model* of the world. Since we are concerned with reasoning under uncertainty, our model will be probabilistic. All reasoning is performed with respect to a probability model, according to the well-established mathematical principles of probability theory. Thus, to the extent that our model is a reasonable one, all conclusions we draw from the model will also be reasonable.

Our basic approach, then, to dealing with uncertainty, is to envision a set of possible states of the world, and to specify a probability distribution over the set of possible worlds. Each possible world is a complete specification of all relevant properties of the world, and the set of possible worlds includes all of the worlds that we consider to be at all possible. The probability distribution over the set of possible worlds describes how likely we think each of the worlds is *a priori*, when we have no specific knowledge about the state of the world. This distribution is therefore called the *prior probability distribution* over the set of possible worlds.

We can incorporate our observations about the world by eliminating from consideration all possible worlds that are inconsistent with our observations. As a result, we will get a new probability distribution over the worlds, called the *posterior probability distribution* over the set of possible worlds, after *conditioning* on our observations. With this approach, we can easily incorporate multiple observations — we simply eliminate from consideration all worlds that conflict with any of the observations.

Also, we can easily determine our degree of belief in any property of the world from the probability distribution over the set of possible worlds. Each property of the world determines a subset of the possible worlds, namely, the set of worlds satisfying the property. The probability of the property is therefore simply the cumulative probability of all the worlds satisfying the property.

Thus, the probabilistic approach to dealing with uncertainty satisfies all the criteria listed above, at least in principle. The key question, of course, is how to achieve the desired effects in practice. In particular, we must address the following issues:

1. What is the set of possible worlds?
2. How do we represent a probability distribution over this set of worlds?
3. How do we reason with this distribution, to determine the likelihood of some property of the world, given our observations about the world?

In this chapter we address the first question, describing the different sets of possible worlds considered in this thesis. We consider questions 2 and 3 in subsequent chapters.

2.1 Attribute-Based Models

A possible world is a complete description of all relevant features of the world. In the simplest case, the state of the world is captured by some finite set of attributes, each describing a particular aspect of the world. For example, if we are modeling a person, we might represent the world using the attributes Height, Weight, Age, Sex, Eye-color and Hair-color.

In this thesis, we will assume that all attributes take on values in a discrete, finite range. For example, the range of the **Eye-color** attribute may be the set $\{\text{brown}, \text{blue}, \text{green}, \text{grey}\}$. The technologies for probabilistic reasoning in artificial intelligence, and of Bayesian networks in particular, have traditionally been most strongly developed for domains with discrete attributes. Many of the ideas from discrete domains carry over to continuous domains, but probabilistic inference is often more difficult in continuous domains. In this thesis, we focus solely on discrete domains. However, much of what we have to say about structured representation of probability models carries over to the continuous case.

A model in which the state of the world is captured by a finite number of discrete attributes is called *attribute-based*. A possible world is then an assignment of values to each attribute. Formally, we make the following definitions.

Definition 2.1.1: An *attribute-based model* \mathcal{M} consists of a finite set of attributes X_1, \dots, X_n . Each attribute X_i has an associated range $Val[X_i]$, where $Val[X_i]$ is a finite set of values.

A *possible world* for \mathcal{M} is a tuple $\langle X_1 : x_1, \dots, X_n : x_n \rangle$, where $x_i \in Val[X_i]$. The *set of possible worlds* for \mathcal{M} consists of all such tuples. We will denote the set of possible worlds for \mathcal{M} by $\Omega_{\mathcal{M}}$, or simply Ω when \mathcal{M} is clear. A single possible world will be denoted by ω .

The *value* of attribute X_i in a world $\omega = \langle X_1 : x_1, \dots, X_n : x_n \rangle$ is x_i , and we write $X_i(\omega) = x_i$. ■

In Chapter 3, we will discuss Bayesian networks, which are a technology for probabilistic reasoning about attribute-based models.

2.2 Relational Models

The technology for probabilistic reasoning in attribute-based models has been well-developed. This thesis is concerned with probabilistic reasoning for richer domains, that cannot easily be captured by a simple set of attributes. For these domains, we use the richer language of relational models. A relational model allows us to

talk about the different entities in the domain, the relationships between them, and properties of each of the entities. Our formulation of typed relational languages and interpretations is fairly generic, and can be adapted quite easily to a variety of formalisms, such as standard first-order logic [24], relational and object-oriented database models [97, 7, 1], description logics [15, 64, 63], and frame representation languages [25, 61], to name a few. We have chosen a presentation that is particularly convenient for developing a theory of representing probability models over relational worlds.

When a possible world is relational, it will consist of a set of domain entities. The entities will be typed — each will belong to some class in a class hierarchy. We will have functions and binary relations that relate entities of one class to entities of another class. Entities will also have attributes describing their particular properties, just as in attribute-based models. We will have names for some particular entities in the world, while others will be unnamed, “generic” entities.

For example, a world may consist of a set of **Person** entities and a set of **Place** entities. We also have the **Student** subclass of **Person** and **School** subclass of **Place**. We have the function **Lives-at** relating people to places, and the function **Studies-at** relating students to schools. We also have the relation **Knows** relating people to people, the relation **Teaches** relating people to students, and the relation **Home-of** relating places to people. Entities of the **Person** class will have the same six attributes as in the previous section. We have particular entities *Jane-Doe* of type **Student** and *Stanford* of type **School**.

Formally, we define the language of relational models as follows.

Definition 2.2.1: A *typed relational language* \mathcal{L} is a structure $\langle \mathbf{C}, \sqsubseteq, \mathbf{A}, \mathbf{f}, \mathbf{R}, \mathbf{I} \rangle$ as follows:

- \mathbf{C} is a set of *classes*.
- \sqsubseteq is a partial ordering over \mathbf{C} , defining the *class hierarchy*. If $C_1 \sqsubseteq C_2$, we say that C_1 is a *subclass* of C_2 , and C_2 is a *superclass* of C_1 . We require that if C is a subclass of both C_1 and C_2 , then either C_1 is a subclass of C_2 or vice versa, (thus ruling out multiple inheritance). Note that a class is a subclass and

superclass of itself. If C_1 is a subclass (superclass) of C_2 , and $C_1 \neq C_2$, we will call C_1 a *proper* subclass (superclass) of C_2 .

- \mathbf{A} is a set of *simple attributes*. Each attribute $A \in \mathbf{A}$ has an associated *domain type*, which is a class in \mathbf{C} , and an associated range $Val[A]$, which is a finite set. If the domain type of A is a superclass of C , A is called a *simple attribute of C* . The range $Val[A]$ will also be called the *range type* of A .
- A set \mathbf{f} of *functions*, and a set \mathbf{R} of *relations*. Each function and relation has an associated *domain type* and an associated *range type*, both of which are classes in \mathbf{C} . A function or relation whose domain type is a superclass of C is called a *complex attribute of C* .
- A set \mathbf{I} of *named instances*, each of which has an associated *type*, which is a class in \mathbf{C} .

We assume that the number of language elements is finite. ■

In this thesis, we limit the language to binary relations, but these can be used to define relations of higher arity in the standard way. The terms “domain type” and “range type”, denote the types of the first and second arguments of a relation, indicate that we think of a binary relation as a multi-valued function having a domain and range. We think of functions and relations as defining attributes of a class, and distinguish them from simple attributes. If we have an entity c whose type is C , each of the simple attributes A of C represents a basic property of c , whose value is an element in $Val[A]$. Meanwhile, if we have a function or relation R whose domain type is C , R will relate c to an entity or set of entities in the range type of R . It is natural to view R as also defining a property of c , whose value is the set of entities related to c by R , and therefore we view R as being an attribute of C . We call it a *complex attribute* to emphasize that its values are domain elements that themselves have properties, whereas the value of a simple attribute is an atomic symbol. We could have defined simple attributes to be functions themselves, but we prefer to emphasize the fact that their values are chosen from a pre-defined set of atomic values, and they will play a special role later in the definition of the probability model. However, we

use the generic term *attribute* for simple attributes, functions and relations, and we shall often use the symbols A , B and D to denote attributes of any kind, whether simple or complex.

We allow attribute names to be overloaded. Two attributes with the same name and different domain types are considered to be distinct attributes. We do not allow two attributes to have the same name, the same domain type and different range types.

Example 2.2.2: Let us make our running example more formal. Our language \mathcal{L} is defined as follows: The set of classes \mathbf{C} is $\{\text{Person}, \text{Place}, \text{Student}, \text{School}\}$. The order relation \sqsubseteq on \mathbf{C} is the set of pairs $\{(\text{Person}, \text{Person}), (\text{Student}, \text{Person}), (\text{Place}, \text{Place}), (\text{School}, \text{Place}), (\text{Student}, \text{Student}), (\text{School}, \text{School})\}$. For simplicity we will have only two attributes in \mathbf{A} : **Eye-color**, whose domain type is **Person** and range is $\{\text{brown}, \text{blue}, \text{green}, \text{grey}\}$, and **Weather**, whose domain type is **Place** and range is $\{\text{sunny}, \text{cloudy}, \text{raining}, \text{snowing}\}$. The set of functions \mathbf{f} consists of the functions **Lives-At**, whose domain type is **Person** and range type is **Place**, and **Studies-At**, whose domain type is **Student** and range type is **School**. The set of relations \mathbf{R} consists of the relation **Knows**, whose domain and range types are both **Person**, and the relation **Home-of**, whose domain type is **Place** and range type is **Person**. Finally, the set of named individuals \mathbf{I} consists of two individuals: *Jane-Doe* whose type is **Student**, and *Stanford* whose type is **School**. ■

Definition 2.2.3: Let $\mathcal{L} = \langle \mathbf{C}, \sqsubseteq, \mathbf{A}, \mathbf{f}, \mathbf{R}, \mathbf{I} \rangle$ be a typed relational language. An *interpretation* ω for \mathcal{L} consists of the following:

- A set Δ^ω of domain elements.
- For each class $C \in \mathbf{C}$, a subset $[C]^\omega \subseteq \Delta^\omega$. The sets $[C]^\omega$ must satisfy the following constraints:
 1. $\cup_{C \in \mathbf{C}} [C]^\omega = \Delta^\omega$.
 2. If $C_1 \sqsubseteq C_2$, $[C_1]^\omega \subseteq [C_2]^\omega$.
 3. If $C_1 \not\sqsubseteq C_2$ and $C_2 \not\sqsubseteq C_1$, $[C_1]^\omega \cap [C_2]^\omega = \emptyset$.

- For each simple attribute $A \in \mathbf{A}$, with domain type C , a function $[A]^\omega : [C]^\omega \rightarrow \text{Val}[A]$.
- For each function $f \in \mathbf{f}$, with domain type C_1 and range type C_2 , a function $[f]^\omega : [C_1]^\omega \rightarrow [C_2]^\omega$.
- For each relation $R \in \mathbf{R}$, with domain type C_1 and range type C_2 , a relation $[R]^\omega \subseteq [C_1]^\omega \times [C_2]^\omega$.
- For each named instance $I \in \mathbf{I}$ of type C , an element $[I]^\omega \in [C]^\omega$. We require that if $I_1 \neq I_2$, $[I_1]^\omega \neq [I_2]^\omega$. ■

The conditions on the $[C]^\omega$ enforce the constraint that every domain element must belong to some class, and must be situated at a unique point in the class hierarchy. That is, there must be a unique class such that it belongs to that class and all of its superclasses, but to no others. The condition on $[I]^\omega$ enforces an assumption that we shall always make, namely that distinct named instances are associated with distinct domain entities.

Definition 2.2.4: Let ω be an interpretation for a typed relational language \mathcal{L} , and let c be an element of Δ^ω . The *class of c in ω* is the class C of \mathcal{L} such that $c \in [C]^\omega$, but $c \notin [C']^\omega$ for any proper subclass C' of C . ■

Example 2.2.5: Consider the language of Example 2.2.2. A possible world ω for

this language is as follows:

Δ^ω	$=$	$\{c_1, c_2, c_3, c_4\}$
$[\text{Person}]^\omega$	$=$	$\{c_1, c_2\}$
$[\text{Place}]^\omega$	$=$	$\{c_3, c_4\}$
$[\text{Student}]^\omega$	$=$	$\{c_1\}$
$[\text{School}]^\omega$	$=$	$\{c_3\}$
$[\text{Eye-color}]^\omega(c_1)$	$=$	<i>green</i>
$[\text{Eye-color}]^\omega(c_2)$	$=$	<i>brown</i>
$[\text{Weather}]^\omega(c_3)$	$=$	<i>sunny</i>
$[\text{Weather}]^\omega(c_4)$	$=$	<i>cloudy</i>
$[\text{Lives-At}]^\omega(c_1)$	$=$	c_4
$[\text{Lives-At}]^\omega(c_2)$	$=$	c_4
$[\text{Studies-At}]^\omega(c_1)$	$=$	c_3
$[\text{Knows}]^\omega$	$=$	$\{(c_1, c_2), (c_2, c_1)\}$
$[\text{Home-Of}]^\omega$	$=$	$\{(c_4, c_1), (c_4, c_2)\}$
$[\text{Jane-Doe}]^\omega$	$=$	c_1
$[\text{Stanford}]^\omega$	$=$	c_3

■

Most relational languages have similar semantics; it is in fact the semantics, that defines the world in terms of entities and the relationships between them, that makes these languages relational. Where they differ most is in the types of statements they allow one to make about the world. In this thesis, we will not be concerned with arbitrary statements of first-order logic. We shall mainly consider statements about named domain entities, or entities that are related to the named entities via some finite sequence of functions or relations.

Definition 2.2.6: Let $\mathcal{L} = \langle \mathbf{C}, \sqsubseteq, \mathbf{A}, \mathbf{f}, \mathbf{R}, \mathbf{I} \rangle$ be a typed relational language, and let C be a class in \mathbf{C} . An *attribute chain*, or simply *chain* on C is a sequence (possibly empty) $A_1 \cdots A_\ell$ of attributes (i.e., simple attributes, functions, or relations), such that the domain type of A_1 is a superclass of C , and for $i > 1$ the domain type of A_i is a superclass of the range type of A_{i-1} . The condition implies that only the last

attribute in a chain can be simple, since a simple attribute has no range type.

An attribute chain is called *simple* or *complex*, depending on whether the last attribute in the chain is simple or complex. It is *single-valued* if none of the attributes in the chain are relations, otherwise it is *multi-valued*. ■

We will use the letters σ , ρ and τ to denote attribute chains. The empty attribute chain will be denoted by ϵ . We can view an attribute chain as representing a derived function or relation on domain entities. More precisely, we have the following definitions.

Definition 2.2.7: Let ω be a possible world, and σ an attribute chain on C .

If σ is the empty chain ϵ , $[\sigma]^\omega$ is the identity function on $[C]^\omega$.

If $\sigma = A_1 \cdots A_\ell$ is single-valued, then for any element $c \in [C]^\omega$, $[\sigma]^\omega(c) = [A_\ell]^\omega([A_{\ell-1}]^\omega(\dots([A_1]^\omega(c))\dots))$. Thus, if σ is simple, $[\sigma]^\omega$ is a function from $[C]^\omega$ to $Val[A_\ell]$. If σ is complex, $[\sigma]^\omega$ is a function from $[C]^\omega$ to $[C']^\omega$, where C' is the range type of A_ℓ .

If $\sigma = A_1 \cdots A_\ell$ is complex and multi-valued, then $[\sigma]^\omega(c)$ is the relation

$$\{(c_0, c_\ell) : \exists c_1, \dots, c_{\ell-1} \text{ s.t. } c_0 \in [C]^\omega, (c_{i-1}, c_i) \in [A_i]^\omega\}. \quad \blacksquare$$

For a particular named instance I , we can talk about the value of the chain σ on I in a possible world ω .

Definition 2.2.8: Let ω be a possible world, I a named instance of type C , and σ an attribute chain on C . If σ is single-valued, $[I.\sigma]^\omega = [\sigma]^\omega([I]^\omega)$. If σ is multi-valued and complex, $[I.\sigma]^\omega = \{c : ([I]^\omega, c) \in [\sigma]^\omega\}$. $[I.\sigma]^\omega$ is called the *value of σ on I in ω* . ■

We have not defined $[\sigma]^\omega$ if σ is simple and multi-valued. A natural, but not particularly useful, definition would have it be a relation on $C \times Val[A_\ell]$. A more useful definition would have it be a multiset of elements in $Val[A_\ell]$, since we will be interested in the number of times it takes on one of its possible values. However, rather than refer to $[\sigma]^\omega$ directly, we deal with it as follows.

Definition 2.2.9: Let $\sigma = \rho.A$ be an attribute chain, where ρ is multi-valued complex and A is simple. For some value $a \in \text{Val}[A]$, we will let $\#[I.\sigma \equiv a]$ denote the number of values of $I.\rho$ that have the value a for A . ■

The kinds of statements about possible worlds that we shall make, then, involve attribute chains on named instances. In particular, we shall make the following statements:

- $I.\sigma = a$, where σ is the single-valued chain $\rho.A$, A is simple, and $a \in \text{Val}[A]$. This statement holds in ω if $[I.\sigma]^\omega = a$.
- $I.\sigma = J$, where σ is the single-valued chain $\rho.A$, A is complex, and J is an instance whose type is the range type of A . This statement holds in ω if $[I.\sigma]^\omega = [J]^\omega$.
- $J \in I.\sigma$, where σ is the multi-valued chain $\rho.A$, A is complex, and J is an instance whose type is the range type of A . This statement holds in ω if $[J]^\omega \in [I.\sigma]^\omega$.
- $\#[I.\sigma] = n$, where σ is multi-valued and complex, and n is a non-negative integer. This statement holds in ω if $|[I.\sigma]^\omega| = n$.
- $\#[I.\sigma \equiv a] = n$, where σ is the multi-valued simple chain $\rho.A$, $a \in \text{Val}[A]$, and n is a non-negative integer. This statement holds in ω if $|\{c \in [I.\rho]^\omega : [A]^\omega(c) = a\}| = n$.

These statements by no means exhaust the expressive power that we could include in the language. For example, we could allow statements of the form $I.\sigma = J.\rho$, where ρ is non-empty, or statements of the form $\#[I.(\sigma \equiv a \wedge \rho \equiv b)] = n$. We shall not allow these more general types of statements, but restrict ourselves to those specified above.

Example 2.2.10: In the language of Example 2.2.2, *Lives-at.Weather* is a simple single-valued attribute chain, *Studies-at* is a complex single-valued chain, *Knows.Lives-at*

is a complex multi-valued chain, and `Knows.Lives-at.Weather` is a simple multi-valued chain. The following statements are true about the possible world of Example 2.2.5:

$$\begin{aligned}
 \text{Jane-doe.Lives-at.Weather} &= \text{cloudy} \\
 \text{Jane-doe.Studies-at} &= \text{Stanford} \\
 \text{Stanford} &\in \text{Jane-doe.Knows.Knows.Studies-at} \\
 \#[\text{Jane-doe.Knows.Lives-at}] &= 1 \\
 \#[\text{Jane-doe.Knows.Lives-at.Weather} \equiv \text{snowing}] &= 0 \quad \blacksquare
 \end{aligned}$$

Statements of the above kind are called *ground statements*. We shall also sometimes make statements that express a connection between one relation and some other relations. One type of statement of this kind is the *inverse statement*, which states that one function or relation is an inverse of another function or relation. Formally:

Definition 2.2.11: Let R_1 be a relation with domain type C_1 and range type C_2 , and let R_2 be a relation whose domain type is C_2 and whose range type is a superclass of C_1 . The statement “ C_2 is an inverse of C_1 ”, written $C_2 = (C_1)^{-1}$, holds in a world ω , if, for every $(c_1, c_2) \in R_1$, $(c_2, c_1) \in R_2$.

The definition holds if either or both of R_1 and R_2 are functions, which are treated as relations for the purpose of this definition. \blacksquare

Note that the definition of inverse is one-sided. It is quite possible for R_2 to be an inverse of R_1 , but not the other way round. Two-sided inverses are however very common. In our running example, `Home-to` and `Lives-at` are natural inverses of each other.

Another type of statement is the *alias* or *same-as* statement. Such a statement defines a function or relation in terms of some complex attribute chain. Formally:

Definition 2.2.12: Let R be a function or relation, and σ a non-empty complex attribute chain, such that the domain type of the first attribute in σ is the domain type of R , and the range type of the last attribute in σ is the range type of R . σ should be single-valued or multi-valued, according to whether R is a function or a relation. The statement “ R is an alias of σ ”, written $R \text{ same-as } \sigma$, holds in a world ω , if $[R]^\omega = [\sigma]^\omega$. \blacksquare

We have said what a possible world looks like for a particular relational language. Given such a language, what is the complete set of possible worlds that we envision, and over which we will define a probability model? The answer to our question depends on what assumptions we are willing to make about what is known. In particular, we may assume that the relational structure of the model is known. In other words, we assume that we know the set of entities, their types, and all the relationships between them, and our uncertainty is only over the values of the simple attributes of the domain entities. Alternatively, we may have uncertainty over the relational structure of the domain itself. In Chapter 4, we deal with a limited class of relational models, in which we make strong assumptions about the structure of the domain, namely, that there are a finite number of entities in the domain and that they are hierarchically organized. This class provides a bridge between attribute-based and relational models, because the state of the world can be captured entirely by a finite set of attributes of the domain entities. In Chapter 5, and Sections 6.2 and 6.3, we deal with increasingly elaborate models, while still making the assumption that the relational structure is fully known. Beginning with Section 6.4, we will consider models in which the relational structure is uncertain.

Chapter 3

Bayesian Networks

3.1 Introduction

Our basic approach to reasoning under uncertainty is to envision a set of possible states of the world, and to define a probability distribution over that set. Such a distribution tells us the relative likelihood of the different possible states of the world. If we have particular observations about the world, we can condition the distribution based on our evidence, to obtain a new posterior distribution over the different possible worlds.

For us, the state of the world will be characterized by a set of attributes, each of which describes some property of the world. We will assume that each attribute takes on values in some discrete, finite domain.¹ A possible world is then an assignment of values to each of the attributes, and the set of possible worlds is the set of all possible combined assignments of values to all the attributes.

Our task is then to represent a probability distribution over this set. In other words, we want to represent a function that assigns a number in $[0, 1]$ to each possible world. It is clear that we cannot hope to represent such a function explicitly, because

¹The technology of Bayesian networks, and particularly the algorithms for probabilistic inference, have been most strongly developed for discrete attributes, but they can also be applied to attributes with continuous domains. In this thesis, we shall assume that all attributes are discrete, but much of what we have to say about structured representation of probability models carries over to the continuous case.

the set of possible worlds is too large. If the world is characterized by n binary-valued attributes, then the number of possible worlds is 2^n . We need a more compact representation.

Over the past fifteen years or so, a technology for representing probability distributions has emerged, called Bayesian networks (BNs) [79]. Bayesian networks exploit domain structure to represent probability distributions in a compact manner. The particular type of structure exploited in Bayesian networks is the conditional independence that holds between different attributes. This chapter presents a review of Bayesian networks, including their definition and semantics, the conditional independence relationships encoded in their structure, the types of reasoning supported, and an algorithm for probabilistic inference. None of the material presented here is new, and formal proofs are not provided for most of the theorems, which can be found in the literature. There are several textbooks available on Bayesian networks, such as [79] and [50].

3.2 Probability Distributions

When the set of possible worlds is finite, as it is for the case of attribute-based models, we can characterize our uncertainty about the state of the world by specifying a probability distribution over the set of worlds. Such a distribution assigns a numerical probability to each possible world, expressing our degree of belief that the true world is actually that particular possible world. Formally, a probability distribution is defined as follows.

Definition 3.2.1: Let Ω be a finite set of possible worlds. A *probability distribution* over Ω is a function $P : \Omega \rightarrow [0, 1]$ such that $\sum_{\omega \in \Omega} P(\omega) = 1$. ■

For a given subset E of Ω , $P(E)$ is defined to be the probability that the actual world is one of the worlds in E . It is equal to $\sum_{\omega \in E} P(\omega)$.

In an attribute-based model \mathcal{M} , we will normally be interested in the probability that some subset of the attributes take on particular values. If X is an attribute, and

x is a value in $Val[X]$, the condition $X = x$ specifies a subset of $\Omega_{\mathcal{M}}$, namely, the set of worlds ω such that $X(\omega) = x$. We can thus speak of $P(X = x)$.

Similarly, for multiple attributes X_1, \dots, X_n , we can speak of $P(X_1 = x_1, \dots, X_n = x_n)$, which is the probability of the set of worlds satisfying all of these conditions. We shall also use boldface notation for sets of attributes, so the same probability can be written $P(\mathbf{X} = \mathbf{x})$, where \mathbf{X} is the set of attributes X_1, \dots, X_n and \mathbf{x} is the set of assignments x_1, \dots, x_n .

We will extend our notation to define functions on the ranges of attributes. Namely, we will write $P(X, \mathbf{Y} = \mathbf{y})$ for the function from $Val[X]$ to $[0, 1]$ defined by

$$P(X, \mathbf{Y} = \mathbf{y})(x) = P(X = x, \mathbf{Y} = \mathbf{y}).$$

Similarly for a set of attributes \mathbf{X} , $P(\mathbf{X}, \mathbf{Y} = \mathbf{y})$ is the function from $Val[\mathbf{X}] = Val[X_1] \times \dots \times Val[X_n]$ to $[0, 1]$ defined by

$$P(\mathbf{X}, \mathbf{Y} = \mathbf{y})(\mathbf{x}) = P(\mathbf{X} = \mathbf{x}, \mathbf{Y} = \mathbf{y}).$$

3.2.1 Conditional probabilities, independence, conditional independence

Definition 3.2.2: Let P be a probability distribution over Ω , and $E, F \subseteq \Omega$, with $P(F) \neq 0$. The *conditional probability of E given F* , written $P(E \mid F)$, is equal to $P(E \cap F)/P(F)$. ■

When $P(F) = 0$, $P(E \mid F)$ is meaningless in standard probability theory. Various extensions to standard probability theory have been proposed that would allow such *conditioning on an event of measure 0*, but we shall not appeal to them in this thesis. Whenever we use the conditional probability notation, we shall always assume that the conditioning event has non-zero probability.

We can extend our notation in the same way as above:

$$\begin{aligned} P(X_1 = x_1, \dots, X_n = x_n \mid Y_1 = y_1, \dots, Y_m = y_m) = \\ P(\{\omega \in \Omega : X_1(\omega) = x_1, \dots, X_n(\omega) = x_n, Y_1(\omega) = y_1, \dots, Y_m(\omega) = y_m\} \mid \\ \{\omega \in \Omega : Y_1(\omega) = y_1, \dots, Y_m(\omega) = y_m\}). \end{aligned}$$

Similarly, $P(X_1, \dots, X_n \mid Y_1, \dots, Y_m)$ is a function on $Val[X_1] \times \dots \times Val[X_n] \times Val[Y_1] \times \dots \times Val[Y_m]$, by analogy with the above. We also use the notation with some variables instantiated, as before.

The definition of conditional probability can be used to derive the ubiquitous *Bayes rule*. If E and F are events, we have from the definition that $P(E \wedge F) = P(E)P(F \mid E)$ and $P(E \wedge F) = P(F)P(E \mid F)$. From these we derive Bayes rule:

$$P(E \mid F) = \frac{P(E)P(F \mid E)}{P(F)}. \quad (3.1)$$

Bayes rule is very useful in computing conditional probabilities in one direction when we are given conditional probabilities in the other direction. In Bayesian networks, we will typically specify the conditional probability of an effect given its causes; Bayes rule can be used to compute the conditional probability of causes given effects.

Another immediate consequence of Definition 3.2.2 is the *chain rule*. If E_1, \dots, E_n and F are events, the chain rule says that

$$\begin{aligned} P(E_1 \wedge \dots \wedge E_n \mid F) &= \\ P(E_1 \mid F)P(E_2 \wedge \dots \wedge E_n \mid E_1 \wedge F) &= \dots = \\ \prod_{i=1}^n P(E_i \mid E_1 \wedge \dots \wedge E_{i-1} \wedge F). \end{aligned} \quad (3.2)$$

A key idea in probability theory is the notion of independence.

Definition 3.2.3: Let E and F be two events. We say that E and F are *independent*, written $I(E, F)$, if $P(E \wedge F) = P(E)P(F)$. ■

The definition can be formulated equivalently in terms of conditional probabilities. Events E and F are independent if $P(E \mid F) = P(E)$, or equivalently, $P(F \mid E) = P(F)$. (We assume here that $P(E) \neq 0$ and $P(F) \neq 0$.) Intuitively, that means that

if E and F are independent, telling me that a world $\omega \in F$ does not give me any information about the probability that $\omega \in E$, and vice versa.

We can generalize the notion of independence to the conditional independence of two events given a third event.

Definition 3.2.4: Let E , F and G be three events. We say that E and F are *conditionally independent given G* , written $I(E, F \mid G)$, if $P(E \wedge F \mid G) = P(E \mid G)P(F \mid G)$. ■

Again, the definition can be written in terms of conditional probabilities, assuming $P(E \wedge G) \neq 0$ and $P(F \wedge G) \neq 0$. Events E and F are conditionally independent given G if $P(E \mid F \wedge G) = P(E \mid G)$, and equivalently, if $P(F \mid E \wedge G) = P(F \mid G)$. Intuitively, this means that if I already know G , telling me about F gives me no additional information as to the probability of E , and vice versa.

Independence and conditional independence can also be defined for variables, or more generally, for sets of variables.

Definition 3.2.5: Let $\mathbf{X} = \{X_1, \dots, X_n\}$, $\mathbf{Y} = \{Y_1, \dots, Y_m\}$ and $\mathbf{Z} = \{Z_1, \dots, Z_\ell\}$ be three sets of variables (not necessarily disjoint). We say that \mathbf{X} is conditionally independent of \mathbf{Y} given \mathbf{Z} (written $I(\mathbf{X}, \mathbf{Y} \mid \mathbf{Z})$), if, for *every* choice of values $\mathbf{x} \in \text{Val}[\mathbf{X}]$, $\mathbf{y} \in \text{Val}[\mathbf{Y}]$, and $\mathbf{z} \in \text{Val}[\mathbf{Z}]$,

$$P(\mathbf{X} = \mathbf{x}, \mathbf{Y} = \mathbf{y} \mid \mathbf{Z} = \mathbf{z}) = P(\mathbf{X} = \mathbf{x} \mid \mathbf{Z} = \mathbf{z})P(\mathbf{Y} = \mathbf{y} \mid \mathbf{Z} = \mathbf{z}). \quad \blacksquare$$

Conditional independence of variables is a strong notion: it implies conditional independence of events for any possible values of the variables. We will sometimes be interested in the weaker notion of *context-specific independence*, where two sets of variables \mathbf{X} and \mathbf{Y} are conditionally independent given specific values \mathbf{z} of a third set of variables \mathbf{Z} . We will write this case as $I(\mathbf{X}, \mathbf{Y} \mid \mathbf{Z} = \mathbf{z})$.

We have defined conditional probabilities in this section in terms of a probability distribution. In a representation language such as Bayesian networks, the process is actually the other way around. We define our probability distribution in terms of the local conditional probabilities of a bunch of attributes. We want a way to talk

about such conditional probabilities as building blocks, even before we have defined the probability distribution.

Definition 3.2.6: Let D_1, \dots, D_n and E be domains. A *conditional probability function (CPF)* from D_1, \dots, D_n to E is a function $f : D_1 \times \dots \times D_n \times E \rightarrow [0, 1]$, such that, for each $(x_1, \dots, x_n) \in D_1 \times \dots \times D_n$, $\sum_{y \in E} f(x_1, \dots, x_n, y) = 1$. If f is a conditional probability function, we will normally use the notation $f(y \mid x_1, \dots, x_n)$ for $f(x_1, \dots, x_n, y)$. ■

3.3 Definition of Bayesian Networks

In a Bayesian network, a joint probability distribution over a set of attributes is encoded by specifying a local probability model for each attribute. The local model for an attribute specifies how the value of the attribute depends probabilistically on the values of a small set of other attributes.

Definition 3.3.1: A *Bayesian network* \mathcal{B} consists of the following:

- A set of attributes $\mathbf{A}[\mathcal{B}] = X_1, \dots, X_n$.
- A directed acyclic graph $\mathcal{G}[\mathcal{B}]$ over \mathbf{X} . For each attribute X_i , we denote the parents of X_i in $\mathcal{G}[\mathcal{B}]$ by $\mathbf{U}^i = \{U_1^i, \dots, U_{m_i}^i\}$ where m_i is the number of parents of X_i .
- For each attribute X_i
 - A domain $\text{Val}[X_i]$.
 - A conditional probability function CPF_{X_i} from $\text{Val}[\mathbf{U}_1^i], \dots, \text{Val}[\mathbf{U}_{m_i}^i]$ to $\text{Val}[X_i]$. ■

Intuitively, if there is an edge in the graph from one attribute to another, we think of the first attribute as a direct cause of the second. There is nothing in the definition of a Bayesian network that requires this causal interpretation, but it is useful to think of it as a causal model. The value of each attribute is determined probabilistically by

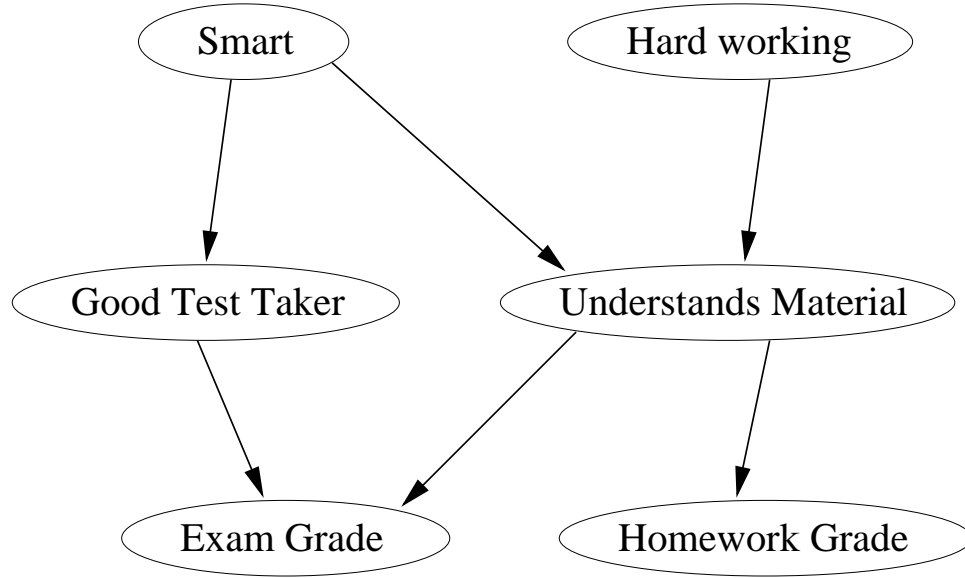


Figure 3.1: A simple Bayesian network.

the values of its parents according to the local conditional probability function of the attribute, so it is natural to think of the values of the parents as causing the values of the children.

The conditional probability function for X is often specified explicitly, listing the value of $P(X = x \mid U_1 = u_1, \dots, U_m = u_m)$ for each combination of v_1, \dots, v_m and w . In that case, the function is called a *conditional probability table*. However, a conditional probability function may be specified more compactly. For example, some of the parents may only be relevant for particular values of other parents. In such cases, the CPF may be specified more compactly using a tree structure [12]. Another example is the noisy-or family of CPFs [39], which describe situations in which the different parents all impact the child independently. We shall ignore the specific manner in which the CPF is encoded, and assume that all its values are explicitly available to us.

Example 3.3.2:

Figure 3.1 shows a simple example of a Bayesian network, describing the performance of a student in a course. The state is captured here through six attributes:

Smart	
<i>True</i>	<i>False</i>
0.5	0.5

Hard Working	
<i>True</i>	<i>False</i>
0.7	0.3

S	Good Test Taker	
	<i>True</i>	<i>False</i>
<i>True</i>	0.75	0.25
<i>False</i>	0.25	0.75

S	HW	Understands Material	
		<i>True</i>	<i>False</i>
<i>True</i>	<i>True</i>	0.95	0.05
<i>True</i>	<i>False</i>	0.6	0.4
<i>False</i>	<i>True</i>	0.6	0.4
<i>False</i>	<i>False</i>	0.2	0.8

GTT	UM	Exam Grade				
		A	B	C	D	F
<i>True</i>	<i>True</i>	0.7	0.25	0.03	0.01	0.01
<i>True</i>	<i>False</i>	0.3	0.4	0.2	0.05	0.05
<i>False</i>	<i>True</i>	0.4	0.3	0.2	0.08	0.02
<i>False</i>	<i>False</i>	0.05	0.2	0.3	0.3	0.15

UM	Homework Grade				
	A	B	C	D	F
<i>True</i>	0.7	0.25	0.03	0.01	0.01
<i>False</i>	0.2	0.3	0.4	0.05	0.05

Figure 3.2: Conditional probability tables for the example network.

Smart, Hard Working, Good Test Taker and Understands Material are all boolean attributes, while Exam Grade and Homework Grade have as their type the set of grades $\{A, B, C, D, F\}$. We will usually refer to the attributes by their initials.

The graphical structure of the network reflects the causal structure of the domain. Whether or not a student is a good test taker depends on her smartness, and we indicate this by setting *S* to be a parent of *GTT* in the network. Similarly, whether or not she understands the material depends both on her smartness and how hard working she is, so the parents of *UM* are *S* and *GTT*. The student's test taking skills and understanding of the material go on to determine how well she does on the exam and homeworks. The former depends on both *UM* and *GTT*, while the latter depends only on *UM*.

Figure 3.2 shows the conditional probability functions for our example network. The functions are represented here as tables. For example, the function CPF_{GTT} specifies that if the student is smart, she is a good test taker with probability 0.75, but if she is not smart, the probability is only 0.3. Not all the entries in the tables

are independent. The constraint in Definition 3.2.6 specifies that the sum of entries in each row must be 1. ■

3.4 Bayesian Network Semantics

A Bayesian network \mathcal{B} defines an attribute-based model, and a probability distribution over the possible worlds $\Omega_{\mathcal{B}}$ of the attribute-based model. That is, \mathcal{B} defines a probability distribution over the possible assignments of values to all the attributes.

Definition 3.4.1: A Bayesian network \mathcal{B} defines a probability distribution $P_{\mathcal{B}}$ over $\Omega_{\mathcal{B}}$, as follows. If ω is the world $\langle X_1 : x_1, \dots, X_n : x_n \rangle$,

$$P_{\mathcal{B}}(\omega) = \prod_{i=1}^n CPF_{X_i}(X_i = x_i \mid U_1^i = u_1^i \dots, U_{m_i}^i = u_{m_i}^i). \quad (3.3)$$

■

In words, the probability of a world ω is the product of the conditional probability of the value of each attribute given the values of its parents, as specified by the CPFs. It is easy to see that this definition has the desired effect, i.e., that $P_{\mathcal{B}}$ is in fact a probability distribution, and the conditional probability of each attribute given its parents is as specified in its CPF:

Proposition 3.4.2: *Let \mathcal{B} be a Bayesian network. Then $P_{\mathcal{B}}$ is a probability distribution over $\Omega_{\mathcal{B}}$, and for each attribute X_i , $P_{\mathcal{B}}(X_i \mid \mathbf{U}^i) = CPF_{X_i}$.*

Proof²: By induction on the number of attributes in \mathcal{B} . In the base case, \mathcal{B} has one attribute X . $\Omega_{\mathcal{B}}$ is equal to $Val[X]$, and $P_{\mathcal{B}}$ is by definition equal to CPF_X . Since X has no parents, CPF_X is a probability distribution over $Val[X] = \Omega_{\mathcal{B}}$.

For the induction step, let \mathcal{B} be a network with attributes X_1, \dots, X_n , and assume without loss of generality that X_n is a leaf of $\mathcal{G}[\mathcal{B}]$. Let \mathcal{B}' be \mathcal{B} with X_n removed.

²The proof is provided since we will later prove a similar theorem for OOBNS.

Then

$$\begin{aligned}
\sum_{\omega \in \Omega_{\mathcal{B}}} P_{\mathcal{B}}(\omega) &= \\
\sum_{\langle X_1:x_1, \dots, X_n:x_n \rangle} \prod_{i=1}^n CPF_{X_i}(x_i \mid \mathbf{u}^i) &= \\
\sum_{\langle X_1:x_1, \dots, X_{n-1}:x_{n-1} \rangle} \prod_{i=1}^{n-1} CPF_{X_i}(x_i \mid \mathbf{u}^i) \sum_{x_n \in \text{Val}[X_n]} CPF_{X_n}(x_n \mid \mathbf{u}^n) \\
&= \sum_{\langle X_1:x_1, \dots, X_{n-1}:x_{n-1} \rangle} \prod_{i=1}^{n-1} CPF_{X_i}(x_i \mid \mathbf{u}^i) \quad (\text{since } CPF_{X_n} \text{ is a CPF}) \\
&= \sum_{\omega' \in \Omega_{\mathcal{B}'}} P_{\mathcal{B}'}(\omega') \\
&= 1. \quad (\text{by induction hypothesis})
\end{aligned}$$

Therefore $P_{\mathcal{B}}$ is a probability distribution over $\Omega_{\mathcal{B}}$.

In addition, for any values x for X_n and \mathbf{u} for \mathbf{U}^n ,

$$\begin{aligned}
P_{P_{\mathcal{B}}}(X_n = x \mid \mathbf{U}^n = \mathbf{u}) &= \frac{\sum_{\omega \in \Omega_{\mathcal{B}}: X_n(\omega)=x, \mathbf{U}^n(\omega)=\mathbf{u}} P_{\mathcal{B}}(\omega)}{\sum_{\omega \in \Omega_{\mathcal{B}}: \mathbf{U}^n(\omega)=\mathbf{u}} P_{\mathcal{B}}(\omega)} \\
&= \frac{\sum_{\omega' \in \Omega_{\mathcal{B}'}: \mathbf{U}^n(\omega')=\mathbf{u}} P_{\mathcal{B}'}(\omega') CPF_{X_n}(X_n=x \mid \mathbf{U}^n=\mathbf{u})}{\sum_{\omega' \in \Omega_{\mathcal{B}'}: \mathbf{U}^n(\omega')=\mathbf{u}} P_{\mathcal{B}'}(\omega')} \\
&= CPF_{X_n}(X_n = x \mid \mathbf{U}^n = \mathbf{u}).
\end{aligned}$$

So $P_{P_{\mathcal{B}}}(X_n \mid \mathbf{U}^n) = CPF_{X_n}$. If $i < n$, $P_{P_{\mathcal{B}}}(X_i) = P_{P_{\mathcal{B}'}}(X_i)$ (by a similar calculation), which by induction hypothesis is equal to CPF_{X_i} . ■

Example 3.4.3: In the network of Example 3.3.2, let ω be the world defined by $\omega(\text{S}) = \text{False}$, $\omega(\text{HW}) = \text{True}$, $\omega(\text{GTT}) = \text{False}$, $\omega(\text{UM}) = \text{True}$, $\omega(\text{EG}) = B$, and $\omega(\text{HG}) = A$. The probability of ω is given by

$$\begin{aligned}
P\omega &= P(\text{S} = \text{False}) \cdot P(\text{HW} = \text{True}) \cdot \\
&\quad P(\text{GTT} = \text{False} \mid \text{S} = \text{False}) \cdot P(\text{UM} = \text{True} \mid \text{S} = \text{False}, \text{HW} = \text{True}) \cdot \\
&\quad P(\text{EG} = B \mid \text{GTT} = \text{False}, \text{UM} = \text{True}) \cdot P(\text{HG} = A \mid \text{UM} = \text{True}) \\
&= 0.5 \cdot 0.7 \cdot 0.75 \cdot 0.6 \cdot 0.3 \cdot 0.7 \\
&= 0.033075.
\end{aligned}$$

■

A useful view of Bayesian networks is that they define a *generative model*, i.e., a process that randomly generates values from a distribution. Here, of course, the

generated values are possible worlds, i.e., assignments of values to all the attributes. This idea of a generative model is sometimes used for probabilistic inference, using some sort of sampling algorithm. However, we can also use this idea to *define* the probability distribution specified by a network: the probability of any possible world is defined to be the probability that the process generates that world.

The generative process for a Bayesian network \mathcal{B} is extremely simple. A value is chosen for each of the attributes in turn, following an order consistent with $\mathcal{G}[\mathcal{B}]$, so that values will always have been assigned to all the parents of an attribute before a value is chosen for the attribute. By the end of the process, a value will have been assigned to each of the attributes. We use the notation $X(\omega) \leftarrow v$ to indicate that ω is updated by assigning the value v to X . The process is defined in pseudocode as follows.

GenerateWorld(\mathcal{B})

Let ω be a new, empty world.

For each attribute X_i of \mathcal{B} , following the order of $\mathcal{G}[\mathcal{B}]$:

 Choose $v \in \text{Val}[X_i]$ with probability $CPF_{X_i}(U_1^i(\omega), \dots, U_{m_i}^i(\omega), v)$.

$X_i(\omega) \leftarrow v$.

Return ω .

It is easy to see intuitively that this generative process defines the same distribution as that presented in the previous section. Since the values of all the attributes are chosen separately from each other, the probability of a particular set of choices of values for all the attributes is the product of the probabilities of each of the individual choices. According to the process, the probability of choosing a value v for a particular attribute X is given by $CPF_{X_i}(U_1^i(\omega), \dots, U_{m_i}^i(\omega), v)$.

While viewing Bayesian networks as generative models may not seem particularly enlightening at this point, its usefulness will become clear in later chapters, when we define probabilistic models with more structure. We will think of such models as defining a program that stochastically generates a possible world, using a generative process that is richer than the one here. Again, the generative process will be used to define the semantics, not to perform inference in these more complex models.

3.5 Conditional Independence and d-Separation

Bayesian networks are a very compact representation. A complete joint probability distribution over n binary-valued attributes requires $2^n - 1$ independent parameters to specify. In contrast, a BN over n binary-valued attributes, in which each node has at most k parents, requires at most $2^k n$ independent parameters. Clearly, such a network can encode only a very small fraction of the possible distributions over these attributes, since it has relatively few parameters. (In fact, any probability distribution over a set of discrete attributes can be encoded in some Bayesian network over those attributes, but only a small number can be represented using a particular network structure.) The fact that the structure of a BN eliminates the vast majority of distributions from consideration indicates that the network structure itself encodes information about the domain. This information takes the form of the conditional independence relationships that hold between attributes in the network.

Recall the definition of conditional independence of variables, restated here for convenience.

Definition 3.5.1: Let $\mathbf{X} = \{X_1, \dots, X_n\}$, $\mathbf{Y} = \{Y_1, \dots, Y_m\}$ and $\mathbf{Z} = \{Z_1, \dots, Z_\ell\}$ be three sets of variables (not necessarily disjoint). We say that \mathbf{X} is conditionally independent of \mathbf{Y} given \mathbf{Z} (written $I(\mathbf{X}, \mathbf{Y} \mid \mathbf{Z})$), if, for *every* choice of values $\mathbf{x} \in \text{Val}[\mathbf{X}]$, $\mathbf{y} \in \text{Val}[\mathbf{Y}]$, and $\mathbf{z} \in \text{Val}[\mathbf{Z}]$,

$$P(\mathbf{X} = \mathbf{x}, \mathbf{Y} = \mathbf{y} \mid \mathbf{Z} = \mathbf{z}) = P(\mathbf{X} = \mathbf{x} \mid \mathbf{Z} = \mathbf{z})P(\mathbf{Y} = \mathbf{y} \mid \mathbf{Z} = \mathbf{z}). \quad \blacksquare$$

The key idea is that the graphical structure of a Bayesian network forces certain conditional independences to hold, *regardless of the CPFs*. Whether or not a conditional independence must hold for a certain network structure can be determined by the *d-separation criterion*:

Definition 3.5.2: Let $\pi = [X_1, \dots, X_m]$ be an undirected path in a Bayesian network \mathcal{B} . We say that X_j has *converging arrows in π* if $\mathcal{G}[\mathcal{B}]$ contains edges from X_{j-1} to X_j and from X_{j+1} to X_j . \blacksquare

Definition 3.5.3: Let π be an undirected path in \mathcal{B} , and \mathbf{X} a set of attributes of \mathcal{B} . We say that π is *blocked by \mathbf{X}* if there exists a node Y in π such that either

1. Y has converging arrows in π , and none of Y or its descendants are in \mathbf{X} , or
2. Y does not have converging arrows, and Y is in \mathbf{X} . ■

Definition 3.5.4: Let \mathbf{X} , \mathbf{Y} , \mathbf{Z} be three disjoint subsets of nodes of a Bayesian network \mathcal{B} . We say that \mathbf{X} is *d-separated from \mathbf{Y} by \mathbf{Z}* if every path from an attribute in \mathbf{X} to an attribute in \mathbf{Y} is blocked by \mathbf{Z} . ■

Theorem 3.5.5: Let \mathcal{G} be a directed acyclic graph over a set of attributes, and \mathbf{X} , \mathbf{Y} and \mathbf{Z} be sets of attributes in the graph. Then $I(\mathbf{X}, \mathbf{Y} \mid \mathbf{Z})$ in every distribution $P_{\mathcal{B}}$ defined by a Bayesian network \mathcal{B} such that $\mathcal{G}[\mathcal{B}] = \mathcal{G}$, if and only if \mathbf{X} is d-separated from \mathbf{Y} by \mathbf{Z} in \mathcal{G} .

Note the prominent place given to the notion of converging arrows in the definition of d-separation, indicating that the directionality of the graph is crucial to the independence assumptions it encodes.

Example 3.5.6: Let us look at some examples from the network in Example 3.3.2. It is clear from Figure 3.1 that $I(\{\mathbf{S}, \mathbf{HW}\}, \{\mathbf{EG}, \mathbf{HS}\} \mid \{\mathbf{GTT}, \mathbf{UM}\})$. Any path from \mathbf{S} or \mathbf{HW} to \mathbf{GTT} or \mathbf{UM} must pass through either \mathbf{EG} or \mathbf{HS} , in such a way that the intermediate node does not have converging arrows. This is intuitively correct. A student's understanding of the material and test-taking skills are direct causes of the student's performance in the exam and homeworks, whereas a person's smarts and hard-working nature are predisposing features that only influence the grades via the direct causes. Once I know the values of the direct causes, telling me the values of the predisposing features gives no new information as to the results.

On the other hand, it is not the case that $I(\{\mathbf{S}\}, \{\mathbf{EG}\} \mid \{\mathbf{GTT}\})$, because the path $\mathbf{S} \rightarrow \mathbf{UM} \rightarrow \mathbf{EG}$ is not blocked by \mathbf{GTT} . Even if I know that a student is a good test taker, telling me that the student is smart gives me reason to believe that the student understands the material, which in turn affects my belief about the exam grade.

Now, consider S and HW . These are independent predisposing features, and indeed $I(\{S\}, \{HW\} \mid \emptyset)$, because every path from S to HW passes through a node with converging arrows. However, S and HW are not conditionally independent given UM . The path $S \rightarrow UM \leftarrow HW$ is not blocked by $\{UM\}$. Intuitively this makes sense. Both smartness and hard work (or their combination) are possible causes of a student understanding the material. If you tell me that the student understands the material, my belief in any possible explanation of that fact will go up, therefore my belief in both the student's smarts and her hard working nature will go up. If you then tell me that the student is smart, that provides me with an explanation of her success in understanding the material, and thereby discounts the explanation that the student understood the material due to hard work, and so my belief in the student's hard work will go down a little. This phenomenon is called “explaining away” — the observation that the student is smart explains away the fact that she understood the material, and discounts the other possible explanation. ■

The above example illustrates the surprising fact that two variables that are independent of each other when there is no other information may become dependent when new information is obtained. This type of reasoning is common in humans. For example, consider the case of a person who discovers he has a symptom of a serious illness, who then finds out that he has a much less serious illness that causes the same symptom. The person experiences relief on discovering he has the less serious illness, indicating that once the person knows he has the symptom, discovering that he has the less serious illness decreases his degree of belief in his having the more serious illness. However, if the person has not observed the symptom, then discovering that he has the less serious illness sheds no new light on whether he has the more serious illness, and does not cause relief. The ability of BNs to correctly handle this type of situation is illustrative of their power to support sophisticated reasoning patterns. We shall see more examples of the sophisticated types of reasoning that can be performed in BNs in the next section.

3.6 Bayesian network reasoning

Because a BN \mathcal{B} specifies a complete joint probability distribution over $\Omega_{\mathcal{B}}$, it determines the probability of any subset of $\Omega_{\mathcal{B}}$. The semantics of a Bayesian network determines the conditional probability of any event given any other event. When computing such a conditional probability, the conditioning event is called the *evidence*, while the event for which we want to determine its conditional probability given the evidence is called the *query*. The general capability of a BN to compute conditional probabilities allows it to exhibit many particular patterns of reasoning.

Example 3.6.1: Causal reasoning is the pattern of reasoning that reasons from a cause to its effects. A hard working student is more likely to understand the material, which in turn makes her more likely to do well on the homeworks, so

$$P(\text{HG} = A \mid \text{HW} = \text{True}) > P(\text{HG} = A) > P(\text{HG} = A \mid \text{HW} = \text{False}). \quad \blacksquare$$

Example 3.6.2: Evidential reasoning is the reasoning from effects to its possible causes. Since a student who understands the material is more likely to do well on the homework than one who does not, observing that the student did well on the homework provides evidence that the student understood the material, which in turn provides evidence that the student is a hard worker.

$$P(\text{HW} = \text{True} \mid \text{HG} = A) > P(\text{HW} = \text{True}) > P(\text{HW} = \text{True} \mid \text{HG} = F). \quad \blacksquare$$

Example 3.6.3: Mixed reasoning combines both causal and evidential reasoning. If I observe that a student did well on the exam, that provides evidence that she understood the material, which in turn makes it more likely that she did well on the homework.

$$P(\text{HG} = A \mid \text{EG} = A) > P(\text{HG} = A) > P(\text{HG} = A \mid \text{EG} = F).$$

Note, however, that HG and EG are conditionally independent given UM, so if I already know that the student understands the material, telling me that she did well

on the exam gives me no extra information about how well she did on the homework.

$$\begin{aligned}
 & P(\text{HG} = A \mid \text{EG} = A, \text{UM} = \text{True}) \\
 &= P(\text{HG} = A \mid \text{UM} = \text{True}) \\
 &= P(\text{HG} = A \mid \text{EG} = F, \text{UM} = \text{True}). \quad \blacksquare
 \end{aligned}$$

Example 3.6.4: Intercausal reasoning involves reasoning between two different causes that have an effect in common. This example corresponds to the “explaining away” phenomenon described at the end of the previous section. If I do not know the value of UM, S and HW are independent.

$$P(\text{HW} = \text{True} \mid \text{S} = \text{True}) = P(\text{HW} = \text{True}) = P(\text{HW} = \text{True} \mid \text{S} = \text{False}).$$

However, if I know that $\text{UM} = \text{True}$, then telling me that $\text{S} = \text{True}$ decreases the probability that $\text{HW} = \text{True}$.

$$\begin{aligned}
 & P(\text{HW} = \text{True} \mid \text{S} = \text{True}, \text{UM} = \text{True}) \\
 &< P(\text{HW} = \text{True} \mid \text{UM} = \text{True}) \\
 &< P(\text{HW} = \text{True} \mid \text{S} = \text{False}, \text{UM} = \text{True}). \quad \blacksquare
 \end{aligned}$$

Example 3.6.5: Sometimes, more than one reasoning pattern is in effect. Suppose for example, that I know that a student received an A on the exam, and I then observe that she is a good test taker. This observation has two opposing effects on my belief about whether the student understood the material. By intercausal reasoning, the observation that the student is a good test taker explains away the observation that she did well on the exam, and discounts the possibility that she understood the material. On the other hand, by mixed reasoning, the fact that she is a good test taker provides evidence that she is smart, which makes it more likely that she understood the material. In this case, we cannot determine whether $P(\text{UM} = \text{True})$ goes up or down based on purely qualitative considerations. However, the Bayesian network still allows the probabilities $P(\text{UM} = \text{True} \mid \text{GTT} = \text{True}, \text{EG} = A)$ and $P(\text{UM} = \text{True} \mid \text{EG} = A)$ to be computed and compared. \blacksquare

Example 3.6.6: A similar situation holds when we have multiple sources of evidence that provide contrary indications. Suppose I observe that a student did well on the exam but not on the homework. The former provides positive evidence that the student understood the material, the latter negative evidence. We cannot determine which evidence is stronger based purely on qualitative considerations, but the Bayesian network can answer the question. ■

The last two examples illustrate situations that are too complex to handle using purely qualitative reasoning methods. It is often quite difficult for a human to guess whether a certain combination of evidence increases or decreases the probability of a query. The situation becomes much more complex as the number of sources of evidence increases. The power of Bayesian networks lies partly in their ability to combine plausible qualitative reasoning patterns, as shown in the first four examples, with quantitative reasoning methods that can carefully weigh different sources of evidence.

3.7 Inference

In this section, we describe the central ideas behind inference in Bayesian networks. The algorithm we shall describe, called **Variable Elimination (VE)**, lies at the core of most exact algorithms for BN inference, including the well-known junction tree algorithm [60]. (Another class of inference algorithms is based on the cutset conditioning method [95], which we shall not describe here.) The presentation in this section is based on the approach of [22].

The basic inference task is as follows: given a BN \mathcal{B} compute $P(\mathbf{Y} \mid \mathbf{Z} = \mathbf{z})$, where \mathbf{Y} and \mathbf{Z} are sets of attributes of \mathcal{B} , and \mathbf{z} is a value in $Val[\mathbf{Z}]$. The value to be computed, $P(\mathbf{Y} \mid \mathbf{Z} = \mathbf{z})$, is the conditional probability distribution over the query variables \mathbf{Y} , given the evidence that the value of \mathbf{Z} is \mathbf{z} ; it is a function that assigns, for each $\mathbf{y} \in Val[\mathbf{Y}]$, a probability $P(\mathbf{Y} = \mathbf{y} \mid \mathbf{Z} = \mathbf{z})$. This conditional

probability is equal to

$$\frac{\sum_{\omega: \mathbf{Y}(\omega)=\mathbf{y}, \mathbf{Z}(\omega)=\mathbf{z}} P_{\mathcal{B}}(\omega)}{\sum_{\omega: \mathbf{Z}(\omega)=\mathbf{z}} P_{\mathcal{B}}(\omega)}.$$

Note that the denominator does not depend on \mathbf{y} , and furthermore, we have the constraint that

$$\sum_{\mathbf{y} \in \text{Val}[\mathbf{Y}]} P_{\mathcal{B}}(\mathbf{Y} = \mathbf{y} \mid \mathbf{Z} = \mathbf{z}) = 1.$$

It is therefore sufficient for us to compute $f(\mathbf{y})$, where for each \mathbf{y} ,

$$f(\mathbf{y}) = \sum_{\omega: \mathbf{Y}(\omega)=\mathbf{y}, \mathbf{Z}(\omega)=\mathbf{z}} P_{\mathcal{B}}(\omega).$$

By the constraint, the denominator $\sum_{\omega: \mathbf{Z}(\omega)=\mathbf{z}} P_{\mathcal{B}}(\omega)$ is equal to $\frac{1}{\sum_{\mathbf{y}} f(\mathbf{y})}$, which is a *normalizing factor* that scales the values of f so that they total 1.

Before showing in detail how we compute the function f , we will introduce some terminology.

Definition 3.7.1: A *factor* over variables \mathbf{Y} is a function $f : \text{Val}[\mathbf{Y}] \mapsto \mathcal{R}$. We say that f *mentions* the variables \mathbf{Y} . ■

In database terms, we can think of a factor as a relation that contains a tuple for each possible assignment of values to \mathbf{Y} , and that associates a number with each tuple. We can define the product of two factors f and g over \mathbf{Y} and \mathbf{Z} to be the join of f and g , where the number associated with each tuple t in the result is the product of the numbers associated with $\mathbf{Y}(t)$ in f and with $\mathbf{Z}(t)$ in g .

Definition 3.7.2: Let f be a factor over \mathbf{Y} and g be a factor over \mathbf{Z} . The *product of f and g* , denoted $f \cdot g$, is a factor h over $\mathbf{Y} \cup \mathbf{Z}$ defined by $h(t) = f(\mathbf{Y}(t))g(\mathbf{Z}(t))$. ■

The product of factors is commutative and associative. We extend it naturally to the product of a set \mathbf{f} of factors, and write $\prod \mathbf{f}$ for the product.

If we have a factor f over \mathbf{Y} , and Z is an attribute in \mathbf{Y} , we can sum over Z to obtain a new factor g over $\mathbf{Y} - \{Z\}$, in which the number associated with each tuple t in g is the sum of the numbers associated with tuples in f that project onto t .

Definition 3.7.3: Let f be a factor over \mathbf{Y} and let Z be an attribute in \mathbf{Y} . The *sum over Z of f* , denoted $\sum_Z f$, is a factor g over $\mathbf{Y} - \{Z\}$, defined by $g(t) = \sum_{z \in \text{Val}[Z]} f(\langle t, Z : z \rangle)$. (The notation $\langle t, Z : z \rangle$ denotes the tuple formed from extending t by assigning z to Z .) ■

The summation operation is analogous to the projection operation in databases. We also define the analogue of the selection operation, as follows:

Definition 3.7.4: Let f be a factor over \mathbf{Y} , let $Z \in \mathbf{Y}$, and let z be a value in $\text{Val}[Z]$. The *conditioning of f by $Z = z$* , written $f[Z = z]$, is the factor g over \mathbf{Y} defined by

$$g(t) = \begin{cases} f(t) & Z(t) = z \\ 0 & Z(t) \neq z \end{cases}.$$

One may also condition a factor on a set of values. If $\{z_1, \dots, z_k\} \subseteq \text{Val}[Z]$, $f[Z \in \{z_1, \dots, z_k\}]$ is the factor g over \mathbf{Y} defined by

$$g(t) = \begin{cases} f(t) & Z(t) \in \{z_1, \dots, z_k\} \\ 0 & Z(t) \notin \{z_1, \dots, z_k\} \end{cases}. \quad \blacksquare$$

Now that we have the concept of a factor, along with some basic operations on factors, we will show how to compute our target function f as a series of operations on factors. Note that a conditional probability function is a factor. The network \mathcal{B} specifies a conditional probability function for each attribute X_i , which is a factor f_X over $X \cup \mathbf{U}^i$. The joint probability distribution defined by \mathcal{B} can be written as the product of factors $\prod_{X \in \mathcal{A}[\mathcal{B}]} f_X$. In addition, the function f that we want to compute is a factor over \mathbf{Y} .

The first step in computing f is to condition on the evidence $\mathbf{Z} = \mathbf{z}$. We do this

by rewriting f as follows:

$$f(\mathbf{y}) = \sum_{\omega: \mathbf{Y}(\omega)=\mathbf{y}, \mathbf{Z}(\omega)=\mathbf{z}} P_{\mathcal{B}}(\omega) = \sum_{\omega: \mathbf{Y}(\omega)=\mathbf{y}} g(\omega), \text{ where}$$

$$g(\omega) = \begin{cases} \prod_X f_X & \mathbf{Z}(\omega) = \mathbf{z} \\ 0 & \mathbf{Z}(\omega) \neq \mathbf{z} \end{cases}.$$

In order to write the function g as a product of factors, we replace, for each $Z_i \in \mathbf{Z}$, the factor f_{Z_i} with $g_{Z_i} = f_{Z_i}[Z_i = z_i]$. For an attribute $Y \notin \mathbf{Z}$, we set $g_Y = f_Y$. We can now write $g = \prod_{X \in \mathbf{A}[\mathcal{B}]} g_X$.

It remains to compute f , where $f(\mathbf{y}) = \sum_{\omega: \mathbf{Y}(\omega)=\mathbf{y}} \prod_{X \in \mathbf{A}[\mathcal{B}]} g_X(\omega)$. Let $\mathbf{W} = \{W_1, \dots, W_\ell\}$ be $\mathbf{X} - \mathbf{Y}$, i.e., the set of attributes of \mathcal{B} that are not part of the query. The factor f is equal to the sum of products of factors:

$$f = \sum_{W_1} \dots \sum_{W_\ell} \prod_X g_X.$$

In order to compute this expression, we eliminate the attributes W_1, \dots, W_ℓ one by one. When eliminating the variable W_i , we can push in the \sum_{W_i} so that it only encloses factors that mention W_i . We then multiply those factors together, and sum over W_i in the result. It is this phase of the algorithm that gives it the name **Variable Elimination**.

The full **Variable Elimination** algorithm is presented below. It takes four arguments: a Bayesian network \mathcal{B} , a set of query variables \mathbf{Y} , a set of evidence variables \mathbf{Z} , and an assignment of values \mathbf{z} to \mathbf{Z} . It returns a joint probability distribution over \mathbf{Y} , which is $P_{P_{\mathcal{B}}}(\mathbf{Y} \mid \mathbf{Z} = \mathbf{z})$. The algorithm maintains a set \mathbf{f} of factors, and updates it as it goes along. After eliminating all the variables, the algorithm is left with a factor f over \mathbf{Y} . The factor f is *unnormalized*, so it calls **Normalize**(f), which normalizes f by scaling its values so that they sum to 1.

VariableElimination(\mathcal{B} , \mathbf{Y} , \mathbf{Z} , \mathbf{z})

For each X in $\mathbf{A}[\mathcal{B}]$ do

 If X is $Z_i \in \mathbf{Z}$

$g_X = CPF_X[X = z_i]$.

S	g_S
True	0.5
False	0.5

HW	g_{HW}
True	0.7
False	0.3

S	GTT	g_{GTT}
True	True	0.75
True	False	0.25
False	True	0.25
False	False	0.75

S	HW	UM	g_{UM}
True	True	True	0.95
True	True	False	0.05
True	False	True	0.6
True	False	False	0.4
False	True	True	0.6
False	True	False	0.4
False	False	True	0.2
False	False	False	0.8

GTT	UM	EG	g_{EG}
True	True	A	0.7
True	False	A	0.3
False	True	A	0.4
False	False	A	0.05

UM	HG	g_{HG}
True	C	0.03
False	C	0.4

Figure 3.3: Initial set of factors for Example 3.7.5

```

Else
   $g_X = CPF_X$ .
   $\mathbf{f} = \{g_X : X \in \mathbf{A}[\mathcal{B}]\}$ .
  For each  $W \in \mathbf{A}[\mathcal{B}] - \mathbf{Y}$  do
    Let  $\mathbf{g}$  be  $\{g \in \mathbf{f} : g \text{ mentions } W\}$ .
     $h_X = \prod \mathbf{g}$ .
     $k_X = \sum_X h_X$ .
     $\mathbf{f} = (\mathbf{f} - \mathbf{g}) \cup \{k_X\}$ .
   $\mathbf{f} = \prod \mathbf{f}$ .
  Return Normalize( $\mathbf{f}$ ).

```

Example 3.7.5: Let us illustrate the variable elimination algorithm by computing the answer to the query discussed in Example 3.6.6. We want to compute $P(\text{UM} \mid \text{EG} = A, \text{HG} = C)$. After conditioning on the evidence, we have the set of factors shown in Figure 3.3. (All entries with value 0 have been omitted from g_{EG} and g_{HG} .)

We will eliminate the variables HG, GTT, EG, S and HW, in that order. Note that HG appears only in g_{HG} , and that its value is fixed by the evidence. Eliminating HG

replaces g_{HG} with the following factor h_1 :

UM	h_1
<i>True</i>	0.03
<i>False</i>	0.4

Next, we eliminate GTT, which appears in g_{GTT} and g_{EG} . We multiply those together to obtain

S	GTT	UM	EG	
<i>True</i>	<i>True</i>	<i>True</i>	A	$0.75 \cdot 0.7 = 0.525$
<i>True</i>	<i>True</i>	<i>False</i>	A	$0.75 \cdot 0.3 = 0.225$
<i>True</i>	<i>False</i>	<i>True</i>	A	$0.25 \cdot 0.4 = 0.1$
<i>True</i>	<i>False</i>	<i>False</i>	A	$0.25 \cdot 0.05 = 0.0125$
<i>False</i>	<i>True</i>	<i>True</i>	A	$0.25 \cdot 0.7 = 0.175$
<i>False</i>	<i>True</i>	<i>False</i>	A	$0.25 \cdot 0.3 = 0.075$
<i>False</i>	<i>False</i>	<i>True</i>	A	$0.75 \cdot 0.4 = 0.3$
<i>False</i>	<i>False</i>	<i>False</i>	A	$0.75 \cdot 0.05 = 0.0375$

We then sum GTT from the result, and replace g_{GTT} and g_{EG} with h_2 , defined as follows:

S	UM	EG	h_2
<i>True</i>	<i>True</i>	A	$0.525 + 0.1 = 0.625$
<i>True</i>	<i>False</i>	A	$0.225 + 0.0125 = 0.2375$
<i>False</i>	<i>True</i>	A	$0.175 + 0.3 = 0.475$
<i>False</i>	<i>False</i>	A	$0.075 + 0.0375 = 0.1125$

We continue in a similar manner. Eliminating EG involves simply summing EG out of h_2 , replacing it with the factor h_3 over S and UM. At this point, the running set of factors consists of g_{S} , g_{HW} , g_{UM} , h_1 and h_3 . To eliminate S, we multiply together g_{S} , g_{UM} and h_3 , and sum over S in the result, replacing them with a factor h_4 over HW

and **UM**. Next we compute $h_5 = \sum_{\text{HW}} g_{\text{HW}} h_4$. We are left with h_1 and h_5 which are factors over **UM** alone. Multiplying these two factors together, we obtain the factor f over **UM**:

UM	f
<i>True</i>	0.0113
<i>False</i>	0.0907

To finish the computation, we normalize f to obtain

$$P(\text{UM} = \textit{True} \mid \text{EG} = A, \text{HG} = C) = 0.8892$$

$$P(\text{UM} = \textit{False} \mid \text{EG} = A, \text{HG} = C) = 0.1108$$

■

What is the cost of the **Variable Elimination** algorithm? In the following analysis, we let $|f|$ denote the number of variables mentioned in a factor. If each variable has at most b values, then the size of f is $O(b^{|f|})$. Now, each variable elimination involves a series of multiplications to compute an intermediate result, followed by a series of additions. The number of multiplications is proportional to the size of the intermediate result, and each row in the intermediate result contributes to one summation term, so the cost of eliminating a variable is proportional to the size of the intermediate result. We therefore have the following:

Theorem 3.7.6: *The cost **Variable Elimination** is $O(nb^M)$ where n is the number of variables, b the largest number of values of a variable, and M the largest number of variables mentioned by an intermediate result produced during the computation.*

From this discussion we see that it is M that is crucial in determining the complexity of BN inference.

In order to better understand the inference computation, it is useful to represent it graphically.

Definition 3.7.7: Let \mathbf{f} be a set of factors, and let \mathbf{X} be the union of the attributes mentioned by the factors in \mathbf{f} . The *graph* of \mathbf{f} is an undirected graph over \mathbf{X} , in

which there is an edge between X and Y if X and Y appear in the same factor in \mathbf{f} .

■

The inference computation can be represented by a series of graphs, corresponding to the current set of factors at each computation stage. Let us denote the set of factors at the i -th stage by \mathbf{f}_i , and the corresponding graph by $G_{\mathbf{f}_i}$. The initial set of factors \mathbf{f}_0 consists of a conditional probability function for each attribute of \mathcal{B} , which is a factor over the attribute and its parents. Therefore, the graph $G_{\mathbf{f}_0}$ will be an undirected graph over the attributes of \mathcal{B} , in which there is an edge between two attributes if one is the parent of the other, or if both are parents of the same attribute (because then they both appear in the CPF of their common child). This graph is called the *moral graph* of \mathcal{B} . Eliminating the variable X at the i -th stage involves an operation on $G_{\mathbf{f}_{i-1}}$ to produce $G_{\mathbf{f}_i}$. Let Y_1, \dots, Y_m be the neighbors of X in $G_{\mathbf{f}_{i-1}}$. Since X appears in a term with each of the Y_i in \mathbf{f} , and with no other variables, the result of eliminating X will be a new factor over $\{Y_1, \dots, Y_m\}$. Thus $G_{\mathbf{f}_i}$ can be obtained from $G_{\mathbf{f}_{i-1}}$ by removing X and adding an edge between every pair of neighbors of X .

Definition 3.7.8: Let $\mathbf{f}_0, \dots, \mathbf{f}_n$ be the sequence of factors produced by a **VE** computation. The *induced graph* of the computation is the union of the graphs of $\mathbf{f}_0, \dots, \mathbf{f}_n$. ■

Figure 3.4 shows the series of graphs for the computation of Example 3.7.5. Figure 3.4(a) shows the moral graph of the Bayesian network, and Figures 3.4(b) to (f) show the graphs after eliminating HG, GTT, EG, S and HW respectively. Note that eliminating GTT requires that an edge be added between S and EG. Figure 3.4(g) shows the induced graph of the entire computation. It is this graph that is crucial in determining the value of M , the size of the largest intermediate result produced by the computation.

Consider any clique (i.e., a maximal fully-connected subgraph) in the induced graph of a **VE** computation. Let X be the first attribute eliminated in this clique, and let the stage of its elimination be i . No edge involving X can be added to

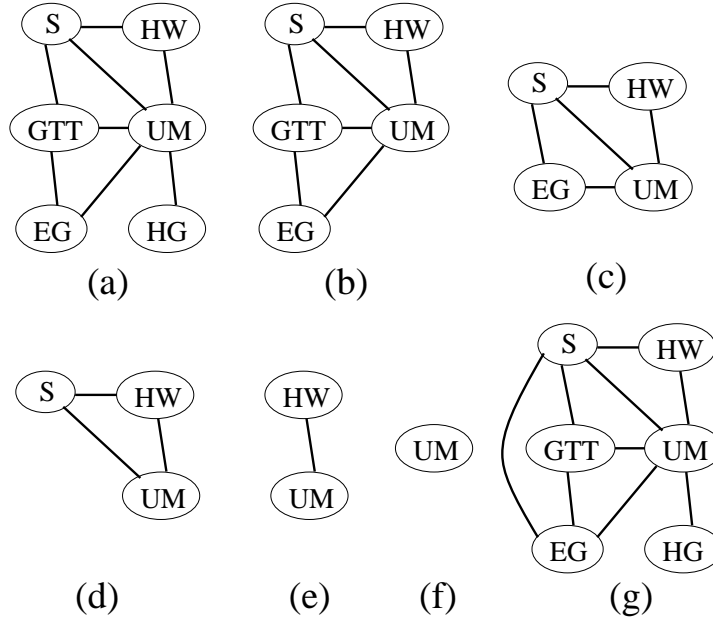


Figure 3.4: Graphs for computation of Example 3.7.5.

the induced graph after X has been eliminated, so \mathbf{f}_i must have contained a factor relating X to each of the other attributes in the clique. The intermediate result produced at the i -th stage will be a factor that mentions at least all the attributes in the clique. So M must be at least the size of the clique. Conversely, consider the intermediate result produced at the i -th stage, and let X be the variable eliminated at that stage. Let \mathbf{Y} be the variables other than X mentioned by the intermediate result. X must have been related to each of the \mathbf{Y} before the i -th stage, while the \mathbf{Y} will all be related to each other after that stage. Therefore $\{X\} \cup \mathbf{Y}$ must be a clique in the induced graph. We conclude that there must be a clique in the induced graph of size M , and therefore M is the size of the largest clique in the induced graph. We have therefore shown the following:

Theorem 3.7.9: *The time and space cost of a **Variable Elimination** computation is $O(nb^M)$, where M is the size of the largest clique in the induced graph for the computation.*

The value of M for a particular query is highly dependent on the elimination order used, so the choice of a good elimination order is crucial to effective BN inference.

The maximum number of neighbors of a node in the induced graph for a **VE** computation, which is equal to $M - 1$, is called the *induced-width* of the computation. Given an initial set of factors \mathbf{f} , one may consider many possible elimination orders. The minimal induced width over all possible elimination order is called the *tree-width* of \mathbf{f} [2]. The tree-width characterizes the cost of computation if the best possible elimination order is found, while the induced width characterizes the cost for a particular order.

An important special case occurs when the Bayesian network is a *polytree*. A polytree is a network in which there are no (undirected) loops. If the network is a polytree, an elimination order can be found in which no edges need to be added to the induced graph. Thus the tree-width is equal to the maximum number of parents of any node in the network.

Theorem 3.7.10: *If \mathcal{B} is a polytree, and Q is a query over a single variable, Q can be solved in time and space $O(nb^p)$, where p is the maximum number of parents of an attribute in \mathcal{B} .*

It is well-known that BN inference is NP-hard in theory. In particular, for **VE**, the cost of inference is $O(nb^M)$ where M is the size of the largest clique in the induced graph. For a graph of size n , the question of whether its tree-width is less than or equal to k is NP-complete [3]. However, there do exist algorithms that can find an optimal ordering in time $O(n^{k+2})$, where k is the tree-width of the graph [3, 90].

These algorithms are complex and expensive, and most practical implementations of BN inference algorithms do not try to find an optimal elimination order. Rather, they use a greedy algorithm to try to find a good order. One heuristic, which is used in many implementations (see, e.g., [50]), is the *minimum discrepancy* heuristic. According to this heuristic, the node that adds fewest edges to the current graph is chosen as the next node to be eliminated.

Despite the fact that BN inference is theoretically hard, in practice inference can be performed on fairly large networks. A well-constructed network possesses a good deal of structure that can be exploited by the inference algorithm. With the right variable elimination order, the value of M can often be kept quite small even for fairly

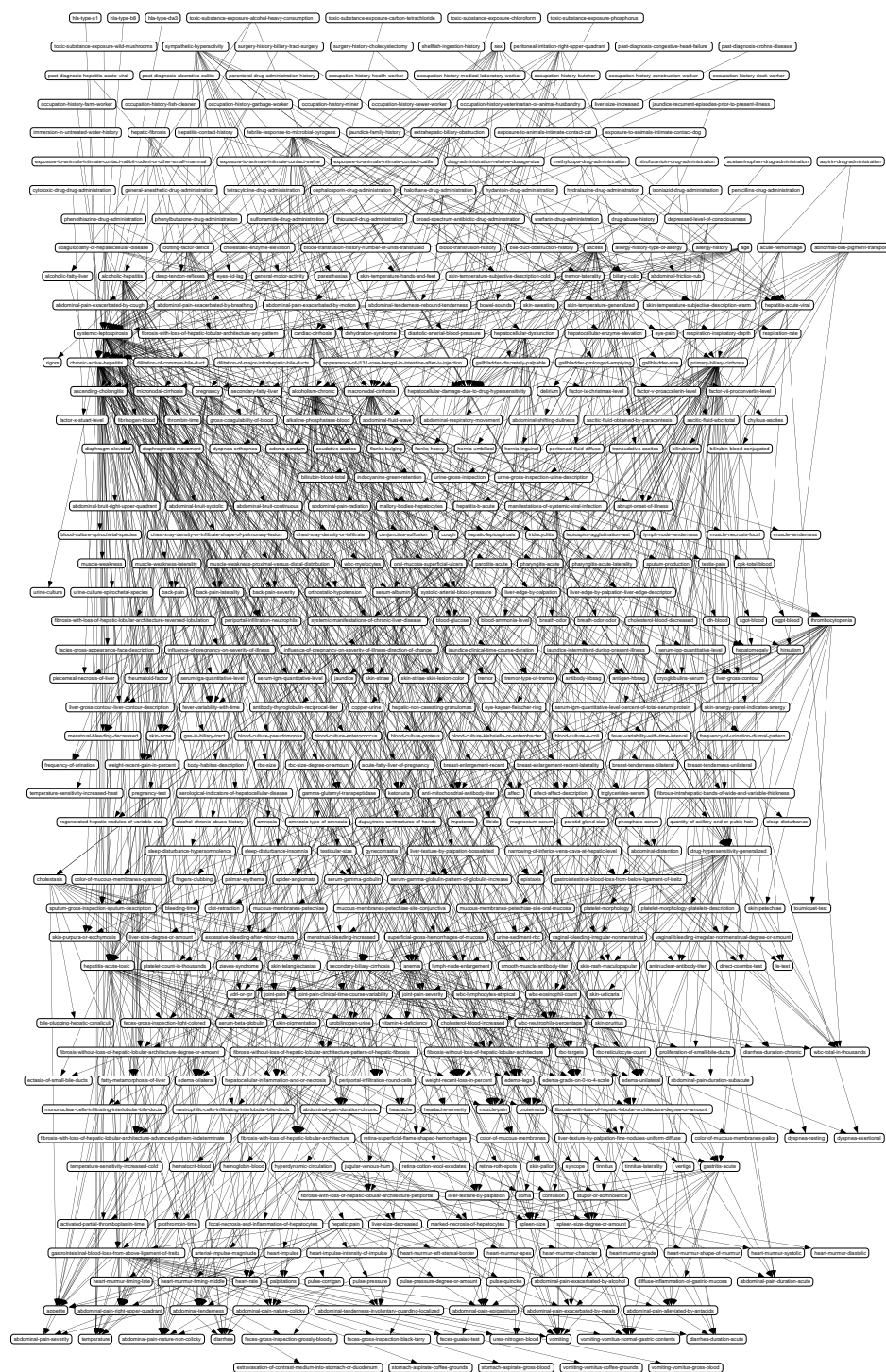


Figure 3.5: The CPCS network.

large networks. An example is the CPCS network for diagnosis of internal medical disorders, shown in Figure 3.5. This network has about 500 nodes, and was specially designed so as to support efficient inference.

If exact inference is too expensive in a network, an approximate inference algorithm may be able to give a fairly accurate answer to a query in a reasonable amount of time. A wide variety of approximate inference algorithms has been developed. These include sampling methods [42, 88], Markov Chain Monte Carlo algorithms [28, 72], algorithms that selectively ignore some of the nodes or edges in the network [23], branch-and-bound style algorithms for finding likely instantiations of the variables [84], and more [71]. One class of algorithms that has drawn attention recently is the family of variational methods [51], which have proven to be extremely successful where they can be applied [47].

3.8 Conclusion

In recent years, BNs have had a growing number of applications, both academic and commercial. Many of the early successes were in the area of medical diagnosis (e.g. [40]). Since then, the range of diagnostic applications has broadened considerably. For example, BNs are now used in the Microsoft Windows troubleshooter, and in troubleshooters for RICOH photocopiers. BNs are also being applied to a growing variety of non-diagnostic applications. For example, they have been applied to monitoring systems such as electric generators [69], to display of information for time-critical decision making [44], for determining the needs of software users [43], and filtering junk email [86].

The key to the success of Bayesian networks is the explicit representation of domain structure: in this case, conditional independence structure in their domain. As we have seen, this structure leads to compact representations and efficient inference algorithms. There are also a number of learning algorithms for BNs that we have not discussed (see [38] for a survey). As a result, BNs have been widely accepted in the artificial intelligence community as being the best way to reason under uncertainty in attribute-based domains. It is our goal in subsequent chapters to extend the

capabilities of BNs to more complex domains.

Chapter 4

Object-oriented Bayesian Networks

4.1 Introduction

As described in the last chapter, Bayesian networks have emerged as a viable technology for representing and reasoning about probabilistic models. As we have seen, they have a variety of successful applications. We now want to take the next step, to scale BNs to more complex domains than they have been applied to in the past. Rather than diagnosis of one particular aspect of medicine, such as internal medicine, we want to be able to diagnose the entire human body. Rather than diagnose only the engine of an aircraft, we want to diagnose an entire aircraft. Rather than provide separate troubleshooters for different components of a computer, we want a single troubleshooter for an entire computer system, and move beyond that to an entire network. Rather than model the behavior of a single military unit in a battlespace, we want to create a model of the entire battlespace. Rather than model an individual student in a single course, we want to be able to model an entire university.

To this point, Bayesian networks have not been applied to these larger, more complex types of applications. We believe that there are a number of reasons for this. The basic difficulty is that in a large system, the number of variables needed to capture the state of the world is very large, and a BN would have to model all of these variables explicitly.

Why are big networks a problem? One issue is that it is very hard to construct

big networks. Building a network to model a system requires a clear conception of how the system works. It requires being able to enumerate all the relevant variables in the system, and to know exactly which variables causally influence other variables. As the number of variables grows, the difficulty of envisioning such a model increases enormously. Consider the CPCS network in Figure 3.5. This network models only a single aspect of medicine, yet it is already immensely complicated. Imagine trying to model the entire human body using the same type of technology.

Modelling using Bayesian networks is a lot like programming using logical circuits. At some point a model becomes simply too complex to fathom in its entirety. It is clear that some facility is needed to divide a model into separate chunks that can each be fathomed separately, and for linking those chunks together into a single coherent model. BNs provide no such facility.

Another problem with standard BN technology is that BNs provide no support for the recurrence of the same elements many times within a model. The same variables show up again and again, and the graph structure and CPFs are repeated many times. The only way to exploit this in current BN technology is by using cut and paste to copy one part of a network to another. The part of the network that is copied has no separate identity, it just consists of a set of nodes in the network. After copying, there is no record that the copied part represents the same set of elements as the original. As a result, the network is very hard to maintain. If a change is needed to the model representing a repeated element, all the copies need to be changed. Furthermore, as copies of different elements are merged together in a variety of ways, it becomes very hard to keep track of which node came from which element. It is clear that we need a way to represent the fact that the same components appear again and again in a BN model, and to give these components their own identity.

A related problem with BNs is their lack of flexibility. We want to be able to model many similar situations with different configurations. We need a mechanism for quickly recombining the components of a network to provide a model for a particular configuration. BNs are static models, defined once and for all. They provide no facilities for reconfiguring the model to adapt to a changing domain.

In addition to the representational inadequacies of BNs for large domains, there

is also the problem of inference. If we do manage somehow to build a model of a very large system, how do we know that it will support efficient inference? Inference in BNs is NP-hard in theory. As we discussed in Section 3.7, the cost of inference in a BN depends on subtle properties of the network structure and variable elimination order used. There is generally no way to know while constructing a large network whether it will support efficient inference in practice. We would like a language and methodology for building large probabilistic models that we know will support efficient inference.

Our approach to representing large systems is to take into account the fact that they typically have a good deal of structure. The structure can be represented in a relational model, by decomposing the system into the separate objects in it and the relationships between them. We will utilize the good properties of Bayesian networks, in particular their ability to explicitly represent conditional independence relationships within a graphical structure, while also taking advantage of the additional relational structure.

This approach provides us with many of the properties we want in a probabilistic representation language. By representing a system in terms of distinct objects, we get to decompose the probabilistic model for the system into models for the objects. These models interact with each other, to be sure, but the representation is decomposed into separate models for the different objects. By associating the probability models with classes of objects rather than with individual objects themselves, we get to exploit the fact that many objects have the same type and therefore share the same probability model. We can also exploit the fact that similar types of objects have similar probability models by allowing a subclass to inherit its model from a superclass. Using a relational language also provides us with a flexible way to reconfigure a system. We can specify the structure of a particular system simply by specifying which objects are in it and how they are related to each other. As we will show, the probability models of the different classes will automatically fit together to produce a probability model for the entire system.

To sum up, we get many of the same properties in a probabilistic modeling language that are provided by higher-level programming languages. Our language has

been inspired in particular by ideas from object-oriented programming languages [31, 10]. As in object-oriented languages, our models are represented in terms of modular components that can be put together in a variety of ways. We associate models with classes of objects, create instances of our classes, and use inheritance to create new models out of old ones. Because we use a typed relational language, our probability models are also well-typed. Of course, our language is not a programming language but a modeling language. Its purpose is to facilitate the construction and analysis of models, not the construction and execution of programs. By presenting our language in the framework of relational models, rather than in a programming language framework, we are able to integrate our probabilistic representation language with more traditional logic-based representation languages.

However, we can use the programming language analogy in thinking about how to design a modular probabilistic language. In particular, the idea of representing a probability model as a *generative stochastic program*, which we discussed in Section 3.4 suddenly becomes very useful. If we take this view, we can associate a generative stochastic function with each of our classes. The stochastic function for one class is defined in terms of the functions for the classes to which it is related, just as a function in an ordinary programming language is defined in terms of other functions. The generative program for the entire system is then defined very naturally in terms of these functions. Of course, our intent is not to execute the generative program, but to use it to define the probability model, and then to analyze that model. Nevertheless, defining the probability model in terms of composing functions is both intuitive and useful.

To deal with the complexity of inference, we can borrow another idea from programming languages: *encapsulation*. The complete description of the state of a particular object I may be very rich. It consists of properties of I itself, as well as properties of related objects. If some other object J is related to I , it may depend probabilistically on properties of I . Typically, the dependence of J on I can be summarized through a very small subset of the properties of I , that serves as the *interface* between I and J . All other properties of I are encapsulated from J . In probabilistic

terms, J is conditionally independent of the encapsulated properties of I given the interface between them. This property can be exploited for effective inference, because it means that the only information that needs to be communicated between I and J during the inference process is a factor over the interface. Most of the work performed during inference can be localized within individual objects. When we utilize the object structure for inference, we can also exploit the fact that many objects have the same class. Different instances of the same class have the same probability model, and communicate the same factor to their related objects. We can therefore perform inference on the class level, reusing the computations performed for one instance of a class to other instances of the same class.

4.1.1 Hierarchical Systems

Before we tackle the integration of probability models into full-scale relational languages, we focus on a common and useful special case: *hierarchical systems*. A hierarchical system consists of a set of objects organized into a *part-of hierarchy*. There is a top-level object, that may have several components, and each of these components may have other components, and so on. The part-of hierarchy is finite, and forms a tree — we don't allow the situation where an object is simultaneously a direct component of two different objects. All interactions and relationships in a hierarchical system flow through the part-of hierarchy. Information flows up the hierarchy from an object to its container, and down the hierarchy from an object to its components. Many complex systems can be naturally modelled in this way, such as, for example, an airplane, a military battalion or a computer system.

In his famous essay, “The Architecture of Complexity” [92], Simon postulated that many complex systems have a natural hierarchical structure. He highlighted two particular features of hierarchical systems that are of great interest to us. First, the different components of a hierarchical system tend to interact *weakly* with each other. Most of the interactions within the system are contained within the individual components, but there are also some interactions between the components. The second aspect of hierarchical systems that is useful to us is the fact that these systems

contain a great deal of *redundancy*. Many of the objects in a system are often of the same type, and in fact the same types of objects often appear in many different systems. These are precisely the two types of system structure that we have proposed to exploit for efficient inference. The weakness of the interactions between objects can be exploited by limiting the interactions between them to small interfaces, while the redundancy can be exploited by performing inference at the class level. Thus, hierarchical systems are a natural starting point for designing our language.

Another reason why hierarchical systems are a good starting point is that they provide a bridge between attribute-based and more general relational models. The set of objects in a hierarchical system is finite and fixed, as are the relationships between them, so the system can be completely characterized by the values of the simple attributes of these objects. In other words, there is a finite set of attributes that fully characterizes any possible world, and a probability distribution over this set of attributes gives us a probability distribution over the state of the system. More general relational models may potentially contain infinitely many objects, and the relationships between them are not necessarily fixed, so the state of the world may not be describable by a finite set of simple attributes. As a result, the probabilistic semantics for these types of systems are necessarily more complex. By presenting hierarchical models first, we can focus on the basic ideas behind structured representation and inference for probability models, and deal with the more general case later.

In this chapter, we present *Object-oriented Bayesian networks (OOBNs)*, a probabilistic representation language for hierarchical systems that exploits the hierarchical structure while maintaining the ability of BNs to represent conditional independence information. We first show how to represent hierarchical systems using a relational language. We then describe how to augment the relational representation with probabilistic knowledge to define an OOBN. Next, we present the semantics of OOBNs, and a structured algorithm for probabilistic inference. Finally, we discuss various aspects of working with OOBNs, focusing in particular on using subclasses and inheritance.

4.2 Hierarchical Relational Models

Our first task is to show how to represent a hierarchical model using a relational language. We know what we want the possible worlds to look like. The set of entities will consist of all the objects in the hierarchy and no others. Each object will be related to its components by a function. If we draw a graph over the objects, with an edge from I to J if I is related to J by a function, the resulting structure should be a tree, rooted at the top-level object. Objects will also have values for simple attributes. In order to allow information to be passed around the hierarchy, some of the attributes of an object will actually be placeholders for values of attributes of other objects. We will force the placeholders to take on the same values as their targets in any possible world.

We achieve the desired effect in four steps. First we define a restricted kind of typed relational language called a *hierarchical relational language*. Then we show how to specify the way information is passed around the hierarchy using an *information-passing structure*. Next, we stipulate constraints on *hierarchical worlds*, that force them to take on the structure we want. Finally, we define what our *set of possible worlds* look like. We show that they do in fact satisfy the constraints, and in addition that any possible world will, for all intents and purposes, be identical to one of the worlds in our set.

Definition 4.2.1: A *hierarchical relational language* is a typed relational language $\langle \mathbf{C}, \sqsubseteq, \mathbf{A}, \mathbf{f}, \mathbf{R}, \mathbf{I} \rangle$, with the following restrictions:

- $\mathbf{R} = \emptyset$.
- \mathbf{I} consists of the single instance I_T , called the *top-level instance*. The type of I_T is called the *top-level class*, and it is denoted C_T .
- The classes \mathbf{C} are *stratified*, in that it is possible to rank them in such a way that for every function f in \mathbf{f} , the range type of f has a lower rank than the domain type. C_T is the highest ranking class. ■

The restriction that $\mathbf{R} = \emptyset$ tells us that all attributes are single-valued. This means that for any object, the number of objects that are related to it is fixed, because all the relationships are functional. All functions map an object to a lower-ranking object; components have lower rank than their containers. The stratification requirement guarantees that the component hierarchy is not recursive. If there is an attribute chain on C whose range type is C' , we will say that C is *defined in terms of* C' . The stratification requirement then guarantees that no two classes are defined in terms of each other. The requirement that C_T be the highest ranking class means that C_T cannot be the range type of any function, so the top-level instance cannot be contained in any other object.

Example 4.2.2: For our running example in this chapter, we will create a model for diagnosis of a computer system. In our example, the hierarchical relational language contains the following elements (among others):

- The classes **Computer**, **Motherboard**, **OS**, **Hard-Drive**, **Drive-Mechanism**, **Drive-Motor**, **Disk-Surface**. For stratification purposes, the classes are listed in order of descending rank. This is not the only possible ranking. For example, since neither the **OS** nor the **Hard-Drive** are defined in terms of the other, either one could be ranked higher than the other.
- A variety of functions mapping objects to their components. For example, the functions **Has-Hard-Drive** with domain type **Computer** and range type **Hard-Drive**, **Has-Drive-Mechanism** with domain type **Hard-Drive** and range type **Drive-Mechanism**, and **Has-Motor** with domain type **Drive-Mechanism** and range type **Drive-Motor**. We can represent the situation where an object has several components of the same type, using multiple numbered functions. For example, the functions **Has-Surface-1**, **Has-Surface-2**, **Has-Surface-3** and **Has-Surface-4**, all with domain type **Hard-Drive** and range type **Disk-Surface**.
- A large number of simple attributes describing properties of the different components. For example, **Status** with domain type **Hard-Drive** and range $\{ \textit{Good}, \textit{Minor-Damage}, \textit{Major-Damage}, \textit{Unreadable} \}$. Most of the different components, will in fact, have a **Status** attribute. Although they have the same name,

they are in fact different attributes, because their domain types and possibly ranges are different.

- A single instance *My-Computer*, whose type is *Computer*. ■

4.2.1 Passing Information Between Objects

When building a probabilistic model for a hierarchical system, we need to be able to describe the ways in which the components interact. For example, the behavior of the hard drive depends on the status of the operating system, because if the OS is corrupt the drive may not function correctly. In order to describe the interactions between objects, we supplement our hierarchical relational language with an *information-passing structure*, describing how information flows between the objects. The information-passing structure consists of three parts. The first part specifies which attributes of an object are actually part of the object itself, and which are simply placeholders for information passed to the object. The placeholders are called *inputs* of the object. The second part of the structure specifies what information about a component object is passed to its container. Attributes which are passed back to the container of an object are called *outputs* of the object.¹ Unlike input attributes, the values of output

¹Unlike input attributes, which are essential to defining which parts of an object are inherent to it, and which parts are actually derived from other objects, output attributes are not an essential feature of the language. Rather, they are a convenient way of specifying which parts of an object are visible to other objects, and thereby defining the interface of the object. We could alternatively have left outputs out of the language definition, and made them a derived property, with the outputs of an object being those aspects of the object that are actually used by other objects. However, making outputs an explicit feature of the language makes the presentation easier.

In addition, making the distinction between output and encapsulated attributes explicit can serve a similar role to the distinction between public and private attributes in a programming language. The designer of a class need only supply models for all the output attributes in order to make sure that the class integrates well with other classes. However, using the distinction between output and encapsulated attributes for this purpose conflates two different interfaces. There is the interface between an object and other objects, as specified by the output and encapsulated attributes. There is also the interface between an object and the user, specifying which attributes of the object are an inherent part of the object model and are therefore queryable, and which attributes are only present for modeling convenience, and are not queryable. One may want an encapsulated attribute to be a guaranteed part of the class model, that can be examined and queried. For example, the *Has-Drive-Mechanism* attribute of the *Hard-Drive* class is encapsulated, but any model for diagnosis of a hard drive should allow the state of its drive mechanism to be examined.

attributes are part of the value of an object. The input and output attributes between them make up the *interface* of an object. The third part of the information-passing structure describes how the information is passed to the contained components of an object. A *binding* is provided for each of the inputs, specifying how the input is defined in terms of other objects. Because only the output attributes of other objects are visible in the containing object, we require that bindings use only their outputs.

Definition 4.2.3: Let $\mathcal{L} = \langle \mathbf{C}, \sqsubseteq, \mathbf{A}, \mathbf{f}, \mathbf{R}, \mathbf{I} \rangle$ be a hierarchical relational language. An *information-passing structure* \mathcal{I} for \mathcal{L} consists of the following:

- For each class $C \in \mathbf{C}$, a subset \mathbf{H}_C of the attributes of C .² The attributes in \mathbf{H}_C are called *input attributes*, while the other attributes of C are called *value attributes*. A complex value attribute is also called a *component function*. The top-level class cannot have inputs, i.e., $\mathbf{H}_{C_T} = \emptyset$.
- For each class $C \in \mathbf{C}$, a subset \mathbf{K}_C of the value attributes of C . Value attributes that are not output attributes are called *encapsulated*. The *interface* of C consists of the input and output attributes of C .
- For each function $f \in \mathbf{f}$, with domain type C and range type C' , and for each input H of C' , a *binding* $\Theta[f.H]$, where $\Theta[f.H]$ is an attribute chain σ on C whose range type is a subclass of the range type of H . All attributes in σ except the first must be output attributes. ■

The information-passing structure regulates the flow of information through the hierarchical system. Information flows from an object to a contained component, and back from the component to the containing object. The pattern is similar to the call-return model of a traditional programming language. We can view the passing of information from the containing object to its component as a function call. The inputs of the component are the formal parameters of the function, and the bindings map the formal parameters to actual parameters.

²We use the letter H if we wish to emphasize that an attribute is an input, and similarly we use K for output attributes. (The letters I and O are taken). The letters A and B will denote attributes of any kind.

In order for the information-passing structure to be coherent, we must make sure that it is *acyclic*. It must be possible to process the components of every object in such a way that any information required by a component has already been produced by the time that component is processed. This means that we cannot allow two objects to depend on each other. We also cannot allow an object to depend on itself. We can determine acyclicity by defining the following graph on the attributes of each class.

Definition 4.2.4: Let \mathcal{L} be a hierarchical relational language, and let \mathcal{I} be an information-passing structure for \mathcal{L} . For each class C of \mathcal{L} , we define the directed graph $G^{\mathcal{I}}[C]$ as follows: The nodes of $G^{\mathcal{I}}[C]$ are the attributes of C , and there is an edge from A to B if B is a complex value attribute with range type C' , and there is some input H of C' such that $\Theta[B.H]$ begins with A .

If $G^{\mathcal{I}}[C]$ is acyclic for every class, we say that \mathcal{I} is acyclic. ■

Note that all simple attributes and input attributes are sources of $G^{\mathcal{I}}[C]$. Later, however, we will define a dependency model for the simple value attributes of objects, so that they too will depend on other attributes, and we will augment the graph associated with C to reflect that.

Example 4.2.5: Let us define some of the information-passing structure for the hierarchical relational language defined in Example 4.2.2. Since **Computer** is the top-level class, it has no inputs. The **Hard-Drive** class has inputs **Temperature**, **Age** and **OS-Status**, and the outputs **Status** and **Full**. Although the hard drive has a rich internal state, the only aspects of its state that influence objects outside the hard drive are whether or not it is working properly and whether or not it is full. In the **Computer** class, the binding $\Theta[\text{Has-Hard-Drive.Temperature}]$ is **Temperature**. This means that the value of the **Temperature** input of the hard drive in a computer will be obtained from the value of the **Temperature** attribute of the computer itself. Similarly, $\Theta[\text{Has-Hard-Drive.Age}]$ is **Age**. To pass information about the status of the operating system to the hard drive, $\Theta[\text{Has-Hard-Drive.OS-Status}]$ is defined to be **Has-OS.Status**, **Status** being an output of the **OS** class. As a result, there will be an edge from **Has-OS** to **Has-Hard-Drive** in $G^{\mathcal{I}}[\text{Computer}]$. ■

4.2.2 Hierarchical Worlds

In order to ensure that the possible worlds have the desired structure, we need to impose some constraints on the interpretations we will consider for the hierarchical relational language. Intuitively, we require the following three conditions to hold: that the object hierarchy is indeed a tree and not a graph; that the values of input attributes agree with their bindings; and that the model does not contain extraneous entities that are not part of the hierarchy. The first two conditions are simple to specify formally, but the third requires the notion of a *subworld*.

Definition 4.2.6: Let $\mathcal{L} = \langle \mathbf{C}, \sqsubseteq, \mathbf{A}, \mathbf{f}, \mathbf{R}, \mathbf{I} \rangle$ be a typed relational language, and let ω and ω' be interpretations of \mathcal{L} . We say that ω' is a *subworld* of ω if the following conditions hold:

- For each $C \in \mathbf{C}$, $[C]^{\omega'} \subseteq [C]^\omega$.
- For each $A \in \mathbf{A}$ with domain type C , and each $c \in [C]^{\omega'}$, $[A]^{\omega'}(c) = [A]^\omega(c)$.
- For each $R \in \mathbf{R}$, with domain type C , and each $c \in [C]^{\omega'}$, $(c, d) \in [R]^{\omega'} \iff (c, d) \in [R]^\omega$.
- For each $f \in \mathbf{f}$, with domain type C , and each $c \in [C]^{\omega'}$, $[f]^{\omega'}(c) = [f]^\omega(c)$.
- For each $I \in \mathbf{I}$, $[I]^{\omega'} = [I]^\omega$.

If any of the $[C]^{\omega'}$ is a proper subset of $[C]^\omega$, ω' is a *proper subworld* of ω . ■

The definition of subworld is basically quite simple, but there is a subtlety. Intuitively, the definition means that any element in ω' is also an element in ω , and the value of any attribute defined on an element in ω' is the same as its value in ω . It looks like we could have used a slightly simpler definition, in which we stipulate that $\Delta^{\omega'} \subseteq \Delta^\omega$, and that for each entity in $\Delta^{\omega'}$, its class in ω' is the same as its class in ω . If we did that, however, we would have lost some cases of subworlds. Intuitively, we want the definition to mean that if ω' is a proper subworld of ω , then ω' is simpler than ω , but its structure is essentially that of ω projected onto the elements in

ω' . The most obvious way in which ω' can be simpler than ω is if it contains fewer elements. However, it is possible for ω' to contain the same elements as ω and yet have a simpler structure, even when the structure is essentially derived from ω . This can happen if the class of an entity in ω' is a proper superclass of its class in ω . I.e., there is some entity c and some classes C and C' with C' a superclass of C , such that $c \in [C]^\omega$, $c \in [C']^{\omega'}$, but $c \notin [C]^{\omega'}$. If C has an attribute A that is not defined on the superclass, c will be in the domain of $[A]^\omega$ but not in the domain of $[A]^{\omega'}$, so ω will have more structure than ω' . The first clause of Definition 4.2.6 achieves the desired effect: it requires only that if an entity belongs to $[C]^{\omega'}$, it must also belong to $[C]^\omega$, so that the structure of ω is at least as rich as that of ω' , but the structure of ω' may be less rich even if it contains the same elements. This clause also guarantees that every entity in $\Delta^{\omega'}$ is also in Δ^ω , because every entity in $\Delta^{\omega'}$ is in $[C]^{\omega'}$ for some C . However, there may be entities in $[C]^\omega$ that are not in $[C]^{\omega'}$, either because they are not in $\Delta^{\omega'}$ or because their class in ω' is a proper superclass of C .

Now that we have formally defined the concept of subworld, we can define a hierarchical world as follows:

Definition 4.2.7: Let \mathcal{L} be a hierarchical relational language, with acyclic information structure \mathcal{I} . A *hierarchical world* for \mathcal{L}, \mathcal{I} is an interpretation ω for \mathcal{L} satisfying the following conditions:

1. Every component function f is one-to-one. In other words, if c_1 and c_2 are in $[C]^\omega$, where C is the domain type of f , and $c_1 \neq c_2$, then $[f]^\omega(c_1) \neq [f]^\omega(c_2)$.
2. Distinct component functions have disjoint ranges. In other words, if f_1 and f_2 are distinct component functions, with domain types C_1 and C_2 , and c_1 and c_2 are entities in $[C_1]^\omega$ and $[C_2]^\omega$ respectively, $[f_1]^\omega(c_1) \neq [f_2]^\omega(c_2)$.
3. If f is a component function with domain type C , $c \in [C]^\omega$, and H is an input attribute of the range type of f , then $[f.H]^\omega(c) = [\Theta[f.H]]^\omega(c)$.
4. There is no proper subworld of ω satisfying conditions 1, 2 and 3. ■

Let us show that these conditions do indeed achieve the effect we want. We will show the following: that a hierarchical possible world contains a distinct entity for

each of the complex chains of component functions defined on the top-level instance; that the relational structure of the possible world matches the structure of the component chains; and that the possible world contains no additional entities.

First we show that in any hierarchical world, different complex chains of value attributes on the top-level instance have different values.

Definition 4.2.8: Let \mathcal{L}, \mathcal{I} be a hierarchical relational language and information-passing structure. A *component chain* for \mathcal{L}, \mathcal{I} is an attribute chain on C_T that consists only of value attributes. (The empty chain is a component chain.) ■

Lemma 4.2.9: If σ_1 and σ_2 are distinct complex component chains, and ω is a hierarchical world, $[I_T.\sigma_1]^\omega \neq [I_T.\sigma_2]^\omega$.

Proof: Suppose first that $\sigma_1 = \rho_1.f_1$, and $\sigma_2 = \rho_2.f_2$, where f_1 and f_2 are different. The result then follows from the fact that the images of $[f_1]^\omega$ and $[f_2]^\omega$ are disjoint.

Now suppose that $\sigma_1 = \rho_1.f$, while $\sigma_2 = \rho_2.f$. ρ_1 and ρ_2 must be distinct, and the result now follows from the fact that $[f]^\omega$ is one-to-one. ■

Next, we show that in a hierarchical world, the value of any attribute chain on the top-level instance is equal to the value of a component chain. This is where we first exploit the requirement that the information-passing structure be acyclic. To use this fact, we define a partial order over the attributes of each class consistent with the information-passing structure, and then lexicographically extend it to a partial order over attribute chains, as follows:

Lemma 4.2.10: Let ω be a hierarchical world for \mathcal{L}, \mathcal{I} , and σ an attribute chain on C_T . There is a component chain ρ such that $[I_T.\sigma]^\omega = [I_T.\rho]^\omega$. If σ is complex, then ρ is unique.

Proof: First, for each class C of \mathcal{L} we define the partial order \preceq_C over the attributes of C , by defining $A \preceq_C B$ iff A is a predecessor of B in $G^{\mathcal{I}}[C]$. Next, we define the partial order \preceq_C^* over attribute chains on C as follows:

- If $\sigma = A.\sigma'$ and $\rho = B.\rho'$ and $A \preceq_C B$, then $\sigma \preceq_C^* \rho$.

- If $\sigma = A.\sigma'$ and $\rho = A.\rho'$ and the range type of A is C' and $\sigma' \preceq_{C'}^* \rho'$, then $\sigma \preceq_C^* \rho$.

We simply write \preceq^* for $\preceq_{C_T}^*$.

Now, suppose σ is not a component chain. Then it must contain an input attribute, which must be preceded by a component function, since C_T has no inputs. So σ must have the form $\tau.A.H.\tau'$, where τ is a component chain, A is a component function, and H is an input attribute. Let σ_1 be $\tau.\Theta[A.H].\tau'$. Since \mathcal{I} is acyclic, $\Theta[A.H]$ must begin with an attribute B such that $B \preceq_C A$ (where C is the range type of τ). Therefore $\Theta[A.H].\tau' \preceq_C^* A.H.\tau'$, and $\tau.\Theta[A.H].\tau' \preceq^* \tau.A.H.\tau'$, or $\sigma_1 \preceq^* \sigma$. We have, using condition 3 of Definition 4.2.7:

$$[I_T.\sigma]^\omega = [\tau']^\omega([A.H]^\omega([\tau]^\omega([I_T]^\omega))) = [\tau']^\omega([\Theta[A.H]]^\omega([\tau]^\omega([I_T]^\omega))) = [I_T.\sigma_1]^\omega.$$

If σ_1 is a component chain, it is our required chain ρ . Otherwise, we can repeat the process, producing σ_2 such that $\sigma_2 \preceq^* \sigma_1$, and $[I_T.\sigma_1]^\omega = [I_T.\sigma_2]^\omega$, and so on. Since the classes of \mathcal{L} are stratified, the length of any chain on C_T is at most the number of classes, and so the number of distinct chains on C_T is finite. Since \preceq^* is a partial order, the above process must therefore terminate in some σ_m , which will be our required chain ρ .

Finally note that if σ is complex, uniqueness follows from Lemma 4.2.9. ■

Note that the proof provides a simple way to compute a component chain ρ such that $[I_T.\sigma]^\omega = [I_T.\rho]^\omega$. We will denote the chain produced by this procedure $\theta(\sigma)$.³

$\rho = \sigma$.

While ρ contains input attributes do:

Write ρ as $\tau.A.H.\tau'$,

where τ is a component chain,

A is a component function,

and H is an input attribute.

$\rho = \tau.\Theta[A.H].\tau'$.

³Big Θ is the notation for bindings, from which small θ is derived.

$$\theta(\sigma) = \rho.$$

Note that by Definition 4.2.3, the range type of $\Theta[A.H]$ must be a subclass of the range type of H , so each replacement operation produces a well-defined attribute chain, and the range type of $\theta\sigma$ is a subtype of the range type of σ .

On the basis of these two lemmas, we can see that a hierarchical world has the sort of structure we want. We have shown that, at least with regard to the entities that can be reached from I_T via some attribute chain, the entities are indeed structured in a tree. I_T is at the root, and the nodes in the tree correspond to the complex component chains defined on C_T . We can use the fact that a hierarchical world has no proper subworlds to show that this tree contains all the entities in the world, and there are no other extraneous entities. Formally, we prove the following.

Theorem 4.2.11: *Let $\mathcal{L} = \langle \mathbf{C}, \sqsubseteq, \mathbf{A}, \mathbf{f}, \mathbf{R}, \mathbf{I} \rangle$ be a hierarchical relational language, \mathcal{I} an acyclic information-passing structure for \mathcal{L} , and ω a hierarchical world for \mathcal{L}, \mathcal{I} . Let Σ denote the set of complex component chains in \mathcal{L} . There exists a one-to-one correspondence ϕ from Σ to Δ^ω , such that for any component chain $\sigma \in \Sigma$, $\phi(\sigma) = [I_T.\sigma]^\omega$, and the class of $\phi(\sigma)$ (Definition 2.2.4) is the range type of σ .*

Proof: Let us define ϕ as required by the condition, so that $\phi(\sigma) = [I_T.\sigma]^\omega$. It follows from Lemma 4.2.9 that ϕ is one-to-one.

To show that ϕ is onto, let $\Delta' \subseteq \Delta^\omega$ be the image of ϕ . We will construct a possible world ω' for \mathcal{L} , and show that it is a subworld of ω , as follows:

- $\Delta^{\omega'} = \Delta'$.
- For each $C \in \mathbf{C}$, we define $[C]^{\omega'}$ to be the image of ϕ under the component chains whose range type is C or a subclass of C . In other words, $c \in [C]^{\omega'}$ iff $c = \phi(\sigma)$ for some σ whose range type is C' , and C' is a subclass of C . Note first that the $[C]^{\omega'}$ defined this way satisfy the conditions of Definition 2.2.3:

1. Every entity $c \in \Delta'$ is $\phi(\sigma)$ for some σ , and σ has a range type C , so $c \in [C]^{\omega'}$ for some C .

2. If $C_1 \sqsubseteq C_2$, and the range type of σ is a subclass of C_1 , then the range type of σ must also be a subclass of C_2 , so $c \in [C_1]^{\omega'}$ implies $c \in [C_2]^{\omega'}$.
3. By Definition 2.2.1, if C is a subclass of both C_1 and C_2 , either $C_1 \sqsubseteq C_2$ or vice versa. So if $C_1 \not\sqsubseteq C_2$ and $C_2 \not\sqsubseteq C_1$, there is no chain σ whose range type is a subclass of both C_1 and C_2 . Therefore $[C_1]^{\omega'} \cap [C_2]^{\omega'} = \emptyset$.

Now note that $c \in [C]^{\omega'}$ implies that $c = [I_T.\sigma]^\omega$, where the range type of σ is a subtype C' of C . Hence, $c \in [C']^\omega$, which implies $c \in [C]^\omega$ since C' is a subclass of C . So the first condition of Definition 4.2.6 is satisfied.

- For each $A \in \mathbf{A}$, with domain type C , and each $c \in [C]^{\omega'}$, $[A]^{\omega'}(c) = [A]^\omega(c)$. The second condition of Definition 4.2.6 is satisfied by definition.
- The third condition of Definition 4.2.6 is satisfied trivially, since \mathbf{R} is empty.
- For each $f \in \mathbf{f}$, with domain type C , let c be in $[C]^{\omega'}$. Then $c = [I_T.\sigma]^\omega$ for some σ whose range type is a subclass C' of C . Therefore $\sigma.f$ is an attribute chain on C_T , and, by Lemma 4.2.10, $[I_T.\sigma.f]^\omega = [I_T.\rho]^\omega$ for some component chain ρ . So $[f]^\omega(c) = [I_T.\rho]^\omega$, which is in Δ' . We can therefore safely define $[f]^{\omega'}(c) = [f]^\omega(c)$, and the fourth condition of Definition 4.2.6 is satisfied.
- $[I_T]^{\omega'} = [I_T]^\omega$, and the fifth condition of Definition 4.2.6 is satisfied.

Thus the possible world ω' is a subworld of ω . In addition, it inherits conditions 1, 2 and 3 of Definition 4.2.7 from ω . From the fact that ω is a hierarchical world, it follows that ω' cannot be a proper subworld of ω . Therefore $[C]^{\omega'}$ must be equal to $[C]^\omega$ for every C . Therefore $\Delta' = \cup_{C \in \mathbf{C}} [C]^{\omega'} = \cup_{C \in \mathbf{C}} [C]^\omega = \Delta$. Since Δ' is the image of ϕ , ϕ is onto.

Finally, for any $\sigma \in \Sigma$, let C be the range type of σ . Then $\phi(\sigma) \in [C]^{\omega'}$, but for a proper subclass C' of C , the range type of σ is not a subclass of C' so $\phi(\sigma) \notin [C']^\omega$. So the class of $\phi(\sigma)$ is C . ■

4.2.3 The Space of Possible Worlds

The above theorem tells us that the relational structure of a hierarchical world is fixed. We know exactly how many domain entities there are, and how they are related to each other. The only things left uncertain are the values of the simple value attributes of the domain entities. We would like to say that Δ^ω , $[C]^\omega$ for each C , $[f]^\omega$ for each f , and $[I_T]^\omega$, are the same for all hierarchical worlds. Actually, there is a technicality: the identity of the elements in Δ^ω could be different, while the structure remains the same otherwise. Obviously, the names of the domain elements are of no interest to us, and we should regard two worlds that differ only in the names of the domain elements to be the same, for all intents and purposes. Technically, we say that two such worlds are isomorphic.

Definition 4.2.12: Let \mathcal{L} be a typed relational language, and ω and ω' two interpretations for \mathcal{L} . We say that ω and ω' are *isomorphic* if there exists a one-to-one correspondence ϕ from Δ^ω to $\Delta^{\omega'}$, such that

- For each $C \in \mathbf{C}$, and each $c \in \Delta^\omega$, $c \in [C]^\omega \iff \phi(c) \in [C]^{\omega'}$.
- For each simple attribute $A \in \mathbf{A}$ of domain type C , and each $c \in [C]^\omega$, $[A]^\omega(c) = [A]^{\omega'}(\phi(c))$.
- For each relation $R \in \mathbf{R}$, and each $c, d \in \Delta^\omega$, $(c, d) \in [R]^\omega \iff (\phi(c), \phi(d)) \in [R]^{\omega'}$.
- For each function $f \in \mathbf{f}$ of domain type C , and each $c \in [C]^\omega$, $\phi([f]^\omega(c)) = [f]^{\omega'}(\phi(c))$.
- For each instance $I \in \mathbf{I}$, $\phi([I]^\omega) = [I]^{\omega'}$. ■

When we define the set of possible worlds over which we define a probability distribution, then, we would like to say that isomorphic interpretations are actually the same possible world. There are at least three approaches to achieving the desired effect. The most elegant uses measure theory — we can actually view isomorphic worlds as distinct, but not allow our probability measure to distinguish between

them. As we will not discuss measure theory until Chapter 5, where we need it for other reasons, we will not use this approach for now. Another approach is to define the set of possible worlds as being equivalence classes of isomorphic interpretations. A third approach is to fix the set of domain elements. We can define Δ^ω to consist of the symbols I_T and $I_T.\sigma$ for each component chain σ .⁴ We shall use this approach in this chapter, since it makes the presentation easier.

Definition 4.2.13: Let \mathcal{L}, \mathcal{I} be a hierarchical relational language and acyclic information-passing structure. The *set of possible worlds* for \mathcal{L}, \mathcal{I} , written $\Omega_{\mathcal{L}, \mathcal{I}}$, or simply Ω when \mathcal{L} and \mathcal{I} are clear, is the set of hierarchical worlds ω for \mathcal{L} in which Δ is $\{I_T.\sigma : \sigma \text{ is a complex component chain for } \mathcal{L}, \mathcal{I}\}$, and the map ϕ from Theorem 4.2.11 maps each σ to $I_T.\sigma$. ■

At this point, we can safely say that all possible worlds really are identical, except for the value of $[A]^\omega$ for the simple attributes. This means that a possible world is characterized by the values of the simple attributes of domain entities. Since the number of domain entities is finite, a possible world is actually characterized by a finite set of simple attributes.

Definition 4.2.14: Let \mathcal{L}, \mathcal{I} be a hierarchical relational language and acyclic information-passing structure. A *basic variable* of \mathcal{L}, \mathcal{I} has the form $I_T.\sigma.A$, where σ is a complex component chain, and A is a simple attribute whose domain type is the range type of σ . We will use the letters X, Y, Z to denote basic variables.

The *value* of the basic variable X in the possible world ω is $[X]^\omega$. ■

The basic variables play a similar role in hierarchical probability models to the attributes in a Bayesian network. In fact, any probability distribution over $\Omega_{\mathcal{L}}$, could be expressed as a probability distribution over an equivalent attribute-based model.

Definition 4.2.15: Let \mathcal{L}, \mathcal{I} be a hierarchical relational language and acyclic information-passing structure. The *attribute-based equivalent* to \mathcal{L}, \mathcal{I} is the attribute-based model \mathcal{M} containing an attribute $X^{\mathcal{M}}$ for each basic variable X in \mathcal{L}, \mathcal{I} , in which $Val[X^{\mathcal{M}}] = Val[X]$. ■

⁴This is akin to restricting our interpretations to the *Herbrand universe*.

Theorem 4.2.16: *Let \mathcal{L}, \mathcal{I} be a hierarchical relational language and acyclic information-passing structure. Let \mathbf{X} be the set of basic variables in \mathcal{L}, \mathcal{I} , and let \mathbf{x} be an assignment of values to \mathbf{X} such that for each $X \in \mathbf{X}$, the assigned value x is in $\text{Val}[X]$. There exists a unique possible world $\omega \in \Omega_{\mathcal{L}, \mathcal{I}}$ such that for each $X \in \mathbf{X}$, $[X]^\omega = x$.*

Proof: The set of entities Δ^ω , and the map ϕ used in Theorem 4.2.11 are fixed by Definition 4.2.13. Theorem 4.2.11 fixes the class of each $c \in \Delta^\omega$, which in turn fixes $[C]^\omega$ for each class C . The values of $[f]^\omega$ for each component function f , and $[I_T]^\omega$, are also fixed by Theorem 4.2.11. For a complex input attribute f with domain type C , let $I_T.\sigma.A$ be some element in $[C]^\omega$. By Lemma 4.2.10, we must have $[f]^\omega(I_T.\sigma.A) = [I_T.\sigma.A.f]^\omega = [I_T.\theta(\sigma.A.f)]^\omega$, so $[f]^\omega$ is also fixed for complex input attributes.

So the only leeway for variation in the possible worlds is in $[A]^\omega$ for simple attributes A . If we require that for every basic variable X , $[X]^\omega = x$, then $[A]^\omega$ is uniquely determined for all simple attributes A . To see this, let A be a simple attribute with domain type C , and $c \in [C]^\omega$. Since $c \in \Delta^\omega$, $c = [I_T.\sigma]^\omega$ for some σ , and $[A]^\omega(c) = [I_T.\sigma.A]^\omega$. By Lemma 4.2.10, $[I_T.\sigma.A]^\omega = [I_T.\theta(\sigma.A)]^\omega$, which must be simple since A is simple. But $I_T.\theta(\sigma.A)$ is some basic variable X , so $[A]^\omega(c)$ must be equal to the value x assigned to X .

It follows that there can be at most one possible world $\omega \in \Omega_{\mathcal{L}, \mathcal{I}}$ satisfying the conditions of the theorem, and we have shown how ω must be defined. We must still show that ω is indeed a possible world, i.e., that it satisfies the four conditions of Definition 4.2.7.

If f is defined on both $I_T.\sigma$ and $I_T.\rho$ and $\sigma \neq \rho$, $[f]^\omega(I_T.\sigma) = I_T.\sigma.f \neq I_T.\rho.f = [f]^\omega(I_T.\rho)$, so condition 1 is satisfied. Clearly distinct component functions have disjoint ranges, because an element in the range of f ends in f , so condition 2 is satisfied.

For condition 3, let f be a component function with domain type C and range type C' , let $I_T.\sigma \in [C]^\omega$, and let H be an input attribute of C' . We have required that $[I_T.\sigma.f.H]^\omega = [I_T.\theta(\sigma.f.H)]^\omega$, and that $[I_T.\sigma.\Theta[f.H]]^\omega = [I_T.\theta(\sigma.\Theta[f.H])]^\omega$. But σ is a component chain, so in the computation of $\theta(\sigma.f.H)$, the first computation step will transform $\sigma.f.H$ into $\sigma.\Theta[f.H]$, so $\theta(\sigma.f.H) = \theta(\sigma.\Theta[f.H])$, and $[f.H]^\omega(I_T.\sigma) =$

$[\Theta[f.H]]^\omega(I_T.\sigma)$, as required.

Finally, the proof of Theorem 4.2.11 shows that no world satisfying conditions 1, 2, and 3 can be a proper subworld of ω , so condition 4 is satisfied. Therefore ω as defined is indeed a possible world in $\Omega_{\mathcal{L},\mathcal{I}}$. ■

Corollary 4.2.17: *Let \mathcal{L}, \mathcal{I} be a hierarchical relational language and acyclic information-passing structure, and \mathcal{M} its attribute-based equivalent. There is a one-to-one correspondence ψ from $\Omega_{\mathcal{M}}$ to $\Omega_{\mathcal{L},\mathcal{I}}$ such that for any world $\omega \in \Omega_{\mathcal{M}}$ and any basic variable X , if $X^{\mathcal{M}}(\omega) = x$, then $[X]^{\psi(\omega)} = x$.*

Proof: Immediate from Theorem 4.2.16. ■

4.3 Specifying the probability model

We have shown how to define a relational model over objects in a part-of hierarchy, and how to specify the way information is passed between the different objects. We have also shown that a possible world is characterized by the values of the simple attributes of the objects in the hierarchy, that is, by the values of the basic variables. We could, therefore, define a probability model over the set of possible worlds simply by specifying a Bayesian network over the basic variables. Strictly speaking, this would define a probability distribution over the set of possible worlds $\Omega_{\mathcal{M}}$ for the attribute-based model \mathcal{M} , but since by Corollary 4.2.17 there is a one-to-one correspondence between $\Omega_{\mathcal{M}}$ and the set $\Omega_{\mathcal{L},\mathcal{I}}$ of possible worlds for the hierarchical model, we could define the probability of each world in $\Omega_{\mathcal{L},\mathcal{I}}$ to be the probability of the corresponding world in $\Omega_{\mathcal{M}}$, to obtain a probability distribution over $\Omega_{\mathcal{L},\mathcal{I}}$.

However, if we follow this approach, we lose all the good properties we were hoping to achieve from using a structured representation language. Instead, we associate a probability model with each class of object, so that every instance of the class shares the same probability model. The class probability models will then determine a probability model over the properties of all objects in the part-of hierarchy. We

will show that defining the probability model in this way produces a model that is equivalent to one defined by a BN over the basic variables.

Associating a local probability model with each class of object has several advantages over defining a BN directly over the basic variables. It is a more *compact* representation: if many of the objects in the part-of hierarchy share the same class, only one model needs to be specified for each of them. We will see also in Section 4.6 that the probability model for a subclass can be derived in large part from the model for a superclass, allowing similar objects to share parts of their models. Defining a local model for each class is *modular*: the local probability model for a class is specified only in terms of attribute chains on that class. This modularity, in turn, makes the representation *flexible*: objects of various classes can be combined into a part-of hierarchy in many different ways, and the class probability models will always fit together to define a probability model over the properties of all the objects in the hierarchy.

A class probability model is specified by defining a model for each of its simple attributes. The model for a simple attribute specifies how the value of that attribute is determined probabilistically by the values of other attributes of the same object and of attributes of related objects. It is very similar to the local model of an attribute in a BN, and consists of a set of parents and a conditional probability function.

Definition 4.3.1: Let A be a simple attribute of class C . A *local probability model* for A consists of:

- A set of parents $\mathbf{v} = v_1, \dots, v_m$, where each v_i is a simple attribute chain on C . All attributes in v_i except the first must be output attributes.
- A conditional probability function CPF_A from $Val[\mathbf{v}]$ to $Val[A]$.

A *class probability model* for a class C consists of a local probability model for each of the simple value attributes of C . ■

Example 4.3.2: The local probability model for the **Status** attribute of the **Hard-Drive** class is defined as follows. It has three parents: **Has-Drive-Mechanism.Status**, **Surface-Damage**,

and FAT (describing the status of the “File Allocation Table” containing the information needed to access files on a Windows machine). CPF_{Status} is

HDM.S	SD	FAT	Status			
			<i>Good</i>	<i>Minor-Dmg</i>	<i>Major-Dmg</i>	<i>Unreadable</i>
<i>Functional</i>	<i>None</i>	<i>Not-Corrupt</i>	0.997	0.001	0.001	0.001
<i>Functional</i>	<i>None</i>	<i>Corrupt</i>	0.3589	0.3597	0.1805	0.1009
<i>Functional</i>	<i>Minimal</i>	<i>Not-Corrupt</i>	0.4308	0.2878	0.1805	0.1009
<i>Functional</i>	<i>Minimal</i>	<i>Corrupt</i>	0.1551	0.3623	0.2918	0.1908
<i>Functional</i>	<i>Major</i>	<i>Not-Corrupt</i>	0.0897	0.2097	0.3	0.4006
<i>Functional</i>	<i>Major</i>	<i>Corrupt</i>	0.0323	0.1833	0.3239	0.4605
<i>Broken</i>	<i>None</i>	<i>Not-Corrupt</i>	0.1246	0.1249	0.25	0.5005
<i>Broken</i>	<i>None</i>	<i>Corrupt</i>	0.0449	0.1348	0.2699	0.5504
<i>Broken</i>	<i>Minimal</i>	<i>Not-Corrupt</i>	0.0539	0.1258	0.2699	0.5504
<i>Broken</i>	<i>Minimal</i>	<i>Corrupt</i>	0.0194	0.11	0.2752	0.5994
<i>Broken</i>	<i>Major</i>	<i>Not-Corrupt</i>	0.0112	0.0637	0.2248	0.7003
<i>Broken</i>	<i>Major</i>	<i>Corrupt</i>	0.004	0.0499	0.2159	0.7302

■

As in a BN, we require that the probability model for a class be acyclic. We cannot allow two simple attributes to mutually influence each other. We check this condition by defining a graph over the attributes of the class.

Definition 4.3.3: Let C be a class with a class probability model \mathcal{P} . The directed graph $G^{\mathcal{P}}[C]$ is defined as follows: The nodes of $G^{\mathcal{P}}[C]$ are the attributes of C , and there is an edge from A to B if B is a simple value attribute, and some parent of B has the form $A.\rho$. ■

Note that complex attributes and input attributes are sources of $G^{\mathcal{P}}[C]$. This is in contrast to $G^{\mathcal{I}}[C]$ (Definition 4.2.4), in which simple and input attributes are always sources. In fact, the two graphs need to be put together to make sure that a class has a coherent local probability model that will also integrate well with other objects. We cannot allow a situation where a simple attribute depends probabilistically on

the value of some complex attribute, but is itself passed as an input to that complex attribute.

Definition 4.3.4: Let \mathcal{L}, \mathcal{I} be a hierarchical relational language and information-passing structure, and C a class in \mathcal{L} with class probability model \mathcal{P} . The *dependency graph* for C , denoted $\mathcal{G}[C]$, is a directed graph over the attributes of C , in which there is an edge from A to B if either $G^{\mathcal{I}}[C]$ or $G^{\mathcal{P}}[C]$ contains an edge from A to B . ■

We will see in the next section that if the dependency graph for the class of every object in a part-of hierarchy is acyclic, the probability models for all the classes define a coherent probability distribution over the properties of the objects in the hierarchy. We call a hierarchical relational language with an information-passing structure in which every class dependency model is acyclic an *Object-Oriented Bayesian Network*.

Definition 4.3.5: An *Object-Oriented Bayesian Network (OOBN)* is a triple $\langle \mathcal{L}, \mathcal{I}, \mathcal{P} \rangle$ in which $\mathcal{L} = \langle \mathcal{C}, \sqsubseteq, \mathbf{A}, \mathbf{f}, \mathbf{R}, \mathbf{I} \rangle$ is a hierarchical relational language, \mathcal{I} is an acyclic information-passing structure for \mathcal{L} , and \mathcal{P} consists of a local probability model \mathcal{P}_C for each $C \in \mathcal{C}$, such that the dependency graph $\mathcal{G}[C]$ is acyclic.

Each of the basic variables of \mathcal{L}, \mathcal{I} are called *basic variables* of the OOBN, while if σ is a complex component chain of \mathcal{L}, \mathcal{I} , $I_T.\sigma$ is called an *instance* in the OOBN. ■

Example 4.3.6: Let us turn our running example into an OOBN by supplying an acyclic probability model for each component of the computer system. Class models for four levels of hierarchy are shown in Figure 4.1.⁵ Besides showing the dependency graph for the classes **Computer**, **Hard-Drive**, **Drive-Mechanism** and **Drive-Motor**, the figure also indicates other aspects of the class model. Complex attributes are shown as rectangles, while simple attributes are ellipses. Each class model is contained in a box. Input attributes intersect the top edge of the box, indicating the fact that their values are received from outside the class, while output attributes intersect the bottom. The rectangles representing the complex components also have little bubbles

⁵The figures shown here contain many more attributes than those described in Example 4.2.2. Some of them are described in Chapter 8.4. For the sake of presentation, we have left out the prefix **Has-** before some of the complex attributes.

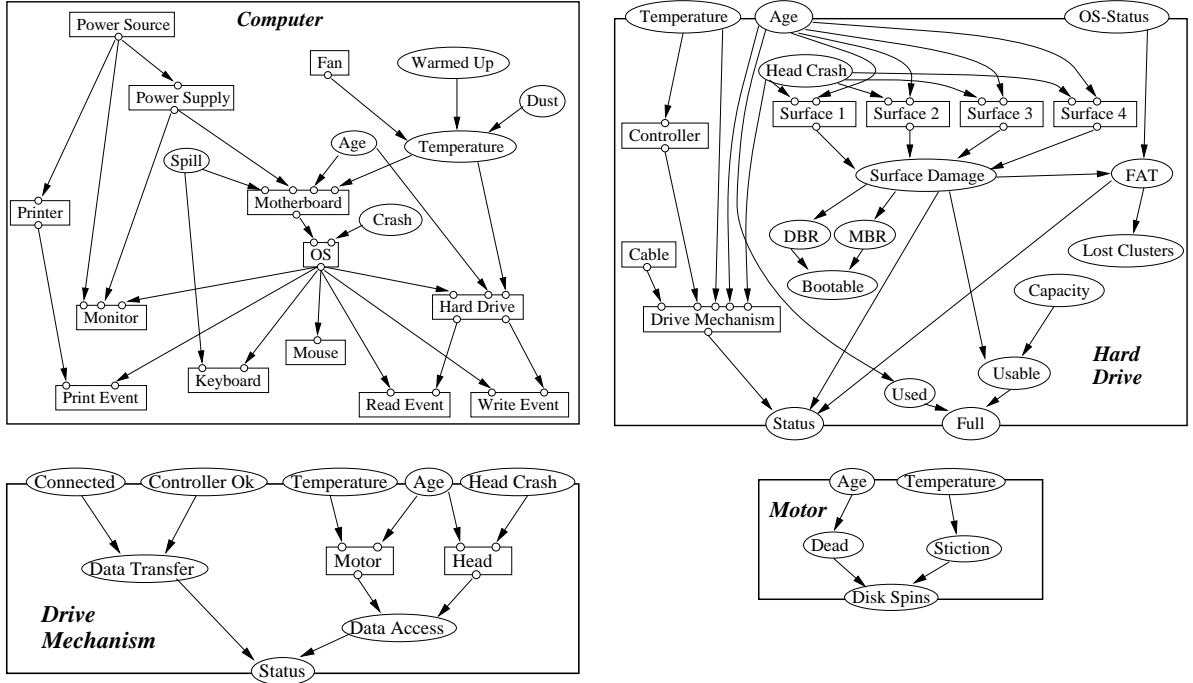


Figure 4.1: Four levels of hierarchy in an OOBN model of a computer system.

on their borders, showing that attributes are passed into and out of those components.

■

4.4 Semantics

As always, we present the semantics of our probabilistic modeling language in terms of a probability distribution over possible worlds. For an OOBN $\mathcal{O} = \langle \mathcal{L}, \mathcal{I}, \mathcal{P} \rangle$, the space of possible worlds is $\Omega_{\mathcal{L}, \mathcal{I}}$, as described in definition 4.2.13. We present the semantics first in terms of a generative process that randomly generates possible worlds, and then in terms of an equivalent probability distribution over the attribute-based equivalent of \mathcal{L}, \mathcal{I} , specified as a BN over the basic variables. Finally, we present semantics for the individual class probability models, in terms of a conditional probability function from values of the input attributes to values of the value attributes.

4.4.1 Generative Process Semantics

We describe the semantics of OOBNS using a natural generative process. The process operates recursively, following the structure of the part-of hierarchy. As it visits each object in the hierarchy, it processes each of the value attributes of the object. Simple attributes are processed by choosing a value for the attribute, according to the CPF of the attribute and the previously generated values of its parents. Processing a complex attribute results in a recursive call on the component object. The values of the inputs of the component are set according to their bindings.

The generative process is presented below, as the function **GenerateWorld**. **GenerateWorld** takes an OOBNS \mathcal{O} as input and returns a possible world ω for \mathcal{O} . The values of $\Delta\omega$, $[C]^\omega$ for each C and $[f]^\omega$ for component functions are set according to Definition 4.2.13. Most of the work in creating the world is done by the recursive function **GenerateEntity**, which takes a domain entity c and its class C . The notation $\Delta^\omega \leftarrow c : C$ means that the domain Δ^ω is augmented by inserting c into it, and setting the type of c to be C (i.e., c is added to $[C']^\omega$ for every supertype C' of C). Similarly, the notation $[A]^\omega(c) \leftarrow v$ means that $[A]^\omega$ is augmented by setting $[A]^\omega(c)$ to v .

Procedure GenerateWorld(\mathcal{O})

Let ω be an empty world.
 Let I_T be the top-level instance of \mathcal{O} .
 Let C_T be the type of I_T .
 $\Delta^\omega \leftarrow I_T : C_T$.
GenerateEntity(I_T , C_T).
 Return ω .

Procedure GenerateEntity(c , C)

Let A_1, \dots, A_n be an ordering of the value attributes of C ,
 such that for no $j < i$ is A_i a predecessor of A_j in $\mathcal{G}[C]$.
 For $i = 1$ to n do:
 If A_i is simple

Let \mathbf{v}^i be the set of parents of A_i in \mathcal{P}_C .
 Choose $w \in \text{Val}[A_i]$, according to $\text{CPF}_{A_i}(w \mid [\mathbf{v}^i]^\omega(c))$.
 $[A]^\omega(c) \leftarrow w$.
 Else
 $\Delta^\omega \leftarrow c.A$.
 Let C' be the range type of A .
 $[A]^\omega(c) \leftarrow c.A : C'$.
 For each input H of C' :
 $[H]^\omega(c.A) \leftarrow [\Theta[A.H]]^\omega(c)$.
GenerateEntity($c.A, C'$).

Let us show that the generative process so defined is coherent. Specifically, we show that the process terminates, and whenever an attribute is processed, the value of any attribute chain needed by that attribute (i.e., a parent of a simple attribute, or a binding of an input of a complex attribute) has already been generated. We rely on the fact that the dependency graph is acyclic, so that we can process the attributes in an order consistent with the graph, in which an attribute is processed only after all of its parents have been processed.

Lemma 4.4.1: *If **GenerateEntity** is called on arguments c , C , and, for any σ that begins with an input H of C , the value of $[\sigma]^\omega(c)$ exists before **GenerateEntity** begins, then:*

1. *For each value attribute A of C , and each attribute chain ρ on C that begins with a predecessor of A in $\mathcal{G}[C]$, the value of $[\rho]^\omega$ exists before A is processed.*
2. **GenerateEntity** terminates.
3. *When **GenerateEntity** terminates, the value of $[\sigma]^\omega$ exists for every attribute chain on C . ■*

Proof: By induction on the rank of C . In the base case, C has no complex attributes. Condition 1 holds because if σ is a chain on C that begins with an attribute that precedes A in $\mathcal{G}[C]$, σ must be a simple attribute B that precedes A in $\mathcal{G}[C]$. B is

either an input, so that $[B]^\omega$ exists by hypothesis when **GenerateEntity** begins, or it is a value attribute that is processed before A , and the value of $[B]^\omega$ is generated as B is processed. Either way the value of $[\sigma]^\omega$ exists before A is processed. Condition 2 holds because there are no recursive calls for complex attributes. Condition 3 holds because the values of all simple attributes are generated directly when they are processed.

To prove condition 1 for the inductive step, suppose the theorem holds for all classes C' whose rank is less than that of C . We will use another induction on the order in which the attributes of C are processed to show that it holds for C . The first value attribute A_1 generated can be preceded only by input attributes in $\mathcal{G}[C]$, so the condition holds in that case. Suppose it holds for every attribute preceding some value attribute A_i . That is, if $k < j < i$, then for any σ beginning with A_k , the value of $[\sigma]^\omega(c)$ exist when A_j is processed. If A_j is simple, the value of $[A_j]^\omega(c)$ will of course be generated when it is processed. If it is complex, values for $[\sigma]^\omega(c)$ exists for all chains σ beginning with a binding $\Theta[A_j.H]$, because $\Theta[A_j.H]$ begins with an attribute A_k that precedes A_j . Processing of A_j will result in a recursive call to **GenerateEntity**($c.A_j, C'$), where C' is the range type of A_j , with $[H]^\omega(c.A_j)$ set to the value of $[\Theta[A_j.H]]^\omega(c)$, for each input H of C' . C' must have lower rank than C , and the hypothesis of the theorem is satisfied for the recursive call, so the recursive call terminates with a value assigned to $[\sigma]^\omega(c.A_j)$ for any chain on C' . It follows that for any chain σ beginning with A_j , the value of $[\sigma.A_j]^\omega$ exists after A_j has been processed. Since this is true for all $j < i$, Condition 1 holds for A_i , and so it holds for all attributes by induction.

Condition 2 is obvious, since every recursive call is on a class of lower rank, which terminates by the induction hypothesis. We have just shown that after processing any attribute A , the value of $[\sigma]^\omega(c)$ exists for any chain beginning with A , so condition 3 holds. ■

Lemma 4.4.2: *Let \mathcal{O} be an OOBN. For any call to **GenerateEntity**(c, C) resulting from a call to **GenerateWorld**(\mathcal{O}), and any chain σ beginning with an input of C , the value of $[\sigma]^\omega(c)$ exists before the call to **GenerateEntity**.*

Proof: By induction on the calls to **GenerateEntity** resulting from the call to **GenerateWorld**(\mathcal{O}). The first call is to **GenerateEntity**(I_T, C_T), and the statement is trivial since C_T has no inputs. If the result is true for a call to **GenerateEntity**(c, C), and A is a complex value attribute of C with range type C' , it is also true for the resulting recursive call to **GenerateEntity**($c.A, C'$), following the same reasoning as in the proof of Lemma 4.4.1. This accounts for all the calls to **GenerateEntity**, so the statement holds. ■

Theorem 4.4.3: **GenerateWorld** *terminates, with a value assigned to $[I_T.\sigma]^\omega$ for every attribute chain σ on C_T .*

Proof: Immediate from Lemmas 4.4.1 and 4.4.2. ■

What exactly does the world generated by **GenerateWorld** look like? First, I_T is inserted into Δ^ω , and **GenerateEntity**(I_T, C_T) is called. Other than I_T , an entity is inserted into Δ^ω iff it is equal to $c.A$ for some complex value attribute A of the class C of an entity c that is previously inserted into Δ^ω and for which **GenerateEntity**(c, C) is called. It is easy to see that in the end, Δ^ω will be exactly the set $\{I_T.\sigma : \sigma \text{ is a complex component chain on } C_T\}$, as required by Definition 4.2.13. Also, for a component function f with domain type C , $[f]^\omega$ is defined to map an element $I_T.\sigma \in [C]^\omega$ to $I_T.\sigma.f$, as also required by that definition. In addition, the values of input attributes are set according to their bindings, as described in the proof of Theorem 4.2.16. The generative process chooses a value for each of the simple value attributes of each of the entities for which **GenerateEntity** is called, so a value is chosen for each of the basic variables, and only those attributes. So the world ω generated by **GenerateWorld** is the unique possible world with the particular chosen values \mathbf{x} for the basic variables \mathbf{X} , as described by Theorem 4.2.16. Formally:

Theorem 4.4.4: *Let $\mathcal{O} = \langle \mathcal{L}, \mathcal{I}, \mathcal{P} \rangle$ be an OOBN. Any world generated by **GenerateWorld**(\mathcal{O}) is a possible world $\omega \in \Omega_{\mathcal{L}, \mathcal{I}}$.*

Based on this theorem, we can define the semantics of an OOBN as follows:

Definition 4.4.5: An OOBN $\mathcal{O} = \langle \mathcal{L}, \mathcal{I}, \mathcal{P} \rangle$ defines a probability distribution $P_{\mathcal{O}}$ over $\Omega_{\mathcal{L}, \mathcal{I}}$, with $P_{\mathcal{O}}(\omega)$ defined to be the probability that the world ω is produced by the **GenerateWorld**(\mathcal{O}) process. ■

4.4.2 Equivalent BN

We can also view an OOBN as defining a BN over the basic variables. As the generative process unfolds, it chooses values for $[A]^{\omega}(c)$, according to CPF_A , given the values $[v]^{\omega}(c)$ of the parents of A . We know that c is $I_T.\sigma$ for some complex component chain σ , so $[A]^{\omega}(c)$ is actually a basic variable $X = [I_T.\sigma.A]^{\omega}$. In addition, we know from Lemma 4.2.10 that for each parent v of A , $[I_T.\sigma.v_i]^{\omega}$ is equal to the value of the basic variable $U_i = [I_T.\theta(\sigma.v_i)]^{\omega}$. We can view A as being a formal variable of C , and each v_i as being a formal parent. For an entity $c = I_T.\sigma$ whose class is C , we will have an actual variable X corresponding to A and actual parents U_i for each of the v_i . The value of X is then chosen according to $CPF_X = CPF_A$, given the previously generated values of its parents. It is easy to see that the values of the U_i are actually generated before the value of X .

Theorem 4.4.6: *Let $X = [I_T.\sigma.A]^{\omega}$ be a basic variable, C be the class of $I_T.\sigma$, and v a parent of A in the probability model of C . Then the basic variable $U = [I_T.\theta(\sigma.A.v)]^{\omega}$ is processed by **GenerateWorld** before X .*

Proof: The proof uses a lexicographic order on attribute chain similar to that of the proof of Lemma 4.2.10, but now the order is based on the dependency graphs of the classes. First, for each class C of \mathcal{L} we define the partial order \leq_C over the attributes of C , by defining $A \leq_C B$ iff A is a predecessor of B in $\mathcal{G}[C]$. Next, we define the partial order \leq_C^* over attribute chains on C as follows:

- If $\sigma = A.\sigma'$ and $\rho = B.\rho'$ and $A \leq_C B$, then $\sigma \leq_C^* \rho$.
- If $\sigma = A.\sigma'$ and $\rho = A.\rho'$ and the range type of A is C' and $\sigma' \leq_{C'}^* \rho'$, then $\sigma \leq_C^* \rho$.

We simply write \leq^* for $\leq_{C_T}^*$.

Now we show that

1. $\theta(\sigma) \leq^* \sigma$
2. $\sigma.v \leq^* \sigma.A$
3. $\theta(\sigma.v) \leq^* \sigma.A$
4. If $\sigma <^* \rho.A$, a value for $[I_T.\sigma]^\omega$ is generated before A is processed in the call to **GenerateEntity** on $I_T.\rho$.

The desired result will then follow immediately from (3) and (4).

(1) holds because $\theta(\sigma) \preceq^* \sigma$ (as shown in the proof of Lemma 4.2.10), and \preceq^* is a suborder of \leq^* , since $\mathcal{G}[C]$ contains all edges in $G^{\mathcal{I}}[C]$.

(2) holds because $v = B.\rho$, where B precedes A in $\mathcal{G}[C]$, so $B \leq_C A$ (C is the range type of σ), $v \leq_C^* A$, and $\sigma.v \leq^* \sigma.A$.

(3) follows from (1) and (2) by transitivity of \leq^* .

To show (4), write $\sigma = \tau.B_1.\sigma'$, and $\rho.A = \tau.B_2.\rho'$, with τ , σ' and ρ' possibly empty, and B_1 and B_2 distinct. I.e., τ is the longest common prefix of σ and ρ . This decomposition must be possible since $\sigma \neq \rho.A$. Since $\sigma <^* \rho.A$, we must have $B_1 <_C B_2$, where C is the range type of τ . In the call to **GenerateEntity**($I_T.\tau, C$), B_1 is processed before B_2 , and by Lemmas 4.4.1 and 4.4.2, the value of $[B_1.\sigma']^\omega(I_T.\tau)$ exists before B_2 is processed. Since $[I_T.\rho.A]^\omega = [I_T.\tau.B_2.\rho']^\omega$ is generated during the processing of B_2 , the result follows. ■

As a result, the **GenerateWorld** process generates values for the basic variables in a manner exactly equivalent to that of the generative process for a BN over the basic variables. So we can view an OOBN as defining the same distribution over the basic variables as does the BN. We call this BN over the basic variables the *flat BN equivalent* of the OOBN. Formally:

Definition 4.4.7: Let $\mathcal{O} = \langle \mathcal{L}, \mathcal{I}, \mathcal{P} \rangle$ be an OOBN. The *flat BN equivalent* of \mathcal{O} , denoted $\mathcal{B}_{\mathcal{O}}^F$ is defined as follows:

- $\mathcal{B}_{\mathcal{O}}^F$ has an attribute $X^{\mathcal{M}}$ for each basic variable X of \mathcal{O} .

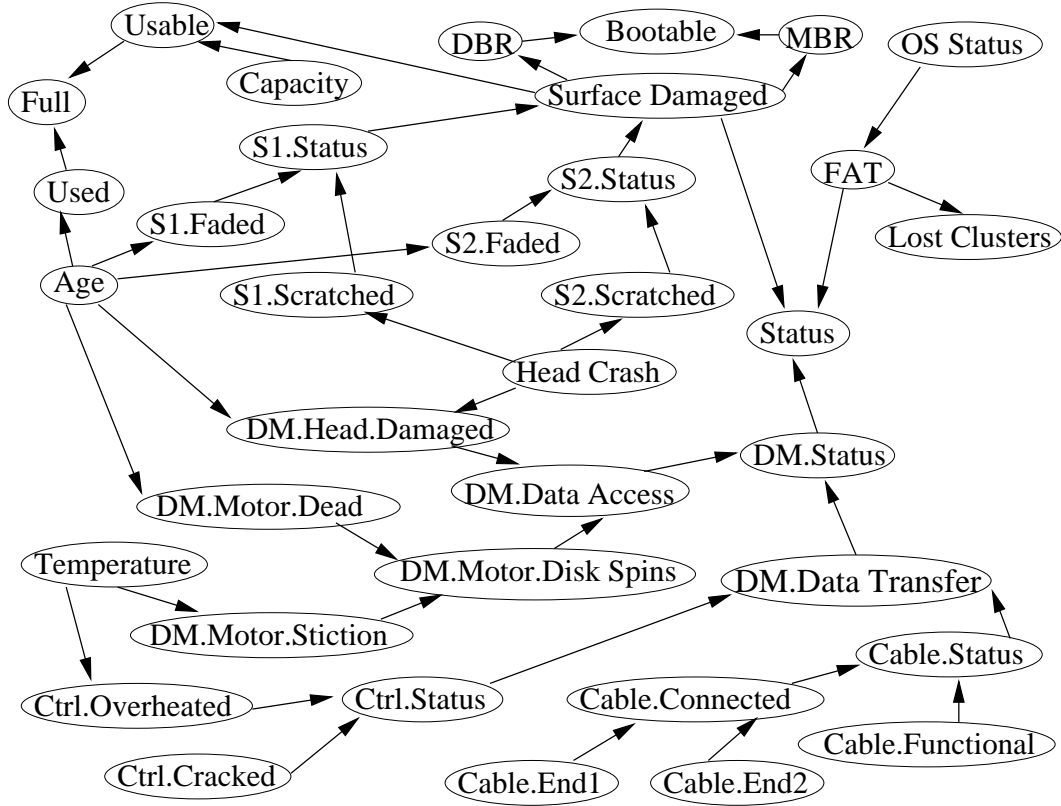


Figure 4.2: Flat BN equivalent of Hard Drive object

- For each basic variable $X = I_T.\sigma.A$, where A a simple attribute of class C , and each parent v of A in \mathcal{P}_C (where C is the range type of σ), let $U = I_T.\theta(\sigma.v)$. Then there is an edge from $U^{\mathcal{M}}$ to $X^{\mathcal{M}}$ in $\mathcal{G}[\mathcal{B}_{\mathcal{O}}^F]$.
- For each attribute $X = I_T.\sigma.A$, the CPF of $X^{\mathcal{M}}$ is CPF_A in \mathcal{P}_C .⁶ ■

The network $\mathcal{B}_{\mathcal{O}}^F$ is a Bayesian network over the attributes of \mathcal{M} , the attribute-based equivalent of \mathcal{L}, \mathcal{I} . It defines a probability distribution $P_{\mathcal{M}}$ over the worlds $\Omega_{\mathcal{M}}$. By Theorem 4.2.17, there is a one-to-one correspondence ψ between $\Omega_{\mathcal{M}}$ and $\Omega_{\mathcal{L}, \mathcal{I}}$, the set of possible worlds for \mathcal{O} . We can therefore use $\mathcal{B}_{\mathcal{O}}^F$ to define a probability distribution over $\Omega_{\mathcal{L}, \mathcal{I}}$. For any $\omega \in \Omega_{\mathcal{L}, \mathcal{I}}$, $P_{\mathcal{O}}(\omega) = P_{\mathcal{M}}(\psi^{-1}(\omega))$.

⁶We will write CPF_X instead of the more precise but ugly $CPF_{X^{\mathcal{M}}}$. This should not cause any confusion.

Example 4.4.8: For the sake of simplicity, let us consider an OOBN in which the **Hard-Drive** is the top-level class, rather than the **Computer** class. Since the **Hard-Drive** class in our running example has inputs, we need to change its model slightly by turning all of its inputs into value attributes. We also reduce the number of disk surfaces from four to two. Figure 4.2 shows the flat BN equivalent for this OOBN.⁷ Each node in the network is a simple attribute chain on $T_v[\text{Hard Drive}]$. Consider, for example, the node **DM.Motor.Dead**. In the **Motor** class, **Dead** has a single parent **Age**. The corresponding parent of **DM.Motor.Dead** is

$$\begin{aligned}
 \theta(\text{DM.Has-Motor.Age}) &= \\
 \theta(\text{DM}.\Theta[\text{Has-Motor.Age}]) &= \\
 \theta(\text{DM.Age}) &= \\
 \theta(\Theta[\text{DM.Age}]) &= \text{Age.} \blacksquare^8
 \end{aligned}$$

4.4.3 Semantics of a class probability model

We have seen how the generative process for an OOBN as a whole defines a probability distribution over possible worlds for the OOBN, and an equivalent distribution over the values of the basic variables. For the sake of modularity, we would also like to give meaning to each individual class probability model, in its own terms, not just in terms of the way all the class models fit together. In fact, if we look at the generative process, we see that there is a natural semantics for individual class probability model.

Consider an individual call to **GenerateEntity**(c, C). Lemma 4.4.2 shows that at the beginning of the call, values already exist for $[\sigma]^\omega$ for any σ beginning with an input H of C . Lemma 4.4.1 shows that at the end of the call, values exist for $[\sigma]^\omega$ for all σ on C . So we can view the behavior of **GenerateEntity** as producing values for

⁷The **Has-** prefix has been dropped from component functions. Also, **Drive Mechanism** has been abbreviated to **DM**, **Surface 1** and **Surface 2** to **S1** and **S2**, and **Controller** to **Ctrl**.

⁸Note that several different attributes in the OOBN all have the same name **Age**. In **DM.Motor.Age**, **Age** appears as an input of the **Motor** object. In **DM.Age**, it is an input of the **DM**. At the end, **Age** stands on its own, as a simple attribute of the **Hard Drive**, and it is also a basic variable. Of course, the same attribute name was used because they are all meant to represent the same value — the age of the computer system.

chains beginning with value attributes of C , given values for chains beginning with input attributes of C . Furthermore, for a particular set of values of the input chains, the behavior of **GenerateEntity** depends only on the probability model \mathcal{P}_C , and on the behavior for the complex attributes of C . Therefore, we can view \mathcal{P}_C as defining a function from values of chains beginning with inputs to probability distributions over the values of chains beginning with value attributes. In other words, \mathcal{P}_C defines a conditional probability function. In order to make this intuition precise, we need some terminology.

Definition 4.4.9: Let C be a class. A *value chain* on C is a simple attribute chain on C consisting only of value attributes. An *input chain* on C is a simple attribute chain on C beginning with an input attribute. If σ is the set of value chains on C , we write $Val[C]$ for $Val[\sigma]$. ■

We can prove an analogue of Lemma 4.2.10 for individual classes.

Lemma 4.4.10: *Let ω be a possible world, and c an entity in Δ^ω whose class is C . If σ is an attribute chain on C , there is a chain ρ that is either an input chain or value chain on C , such that $[\sigma]^\omega(c) = [\rho]^\omega(c)$.*

Proof: Let $c = I_T.\tau$. We know that $[I_T.\tau.\sigma]^\omega = [I_T.\theta(\tau.\sigma)]^\omega$, and the same is true for $I_T.\rho$ for any intermediate ρ produced in computing $\theta(\tau.\sigma)$. We will show that during the process of computing $\theta(\tau.\sigma)$, some intermediate result $\tau.\rho$ is produced in which ρ is either a value chain or an input chain on C . If $\theta(\tau.\sigma)$ begins with τ , $\theta(\tau.\sigma) = \tau.\rho$ with ρ a value chain, and we are done. If, on the contrary, $\theta(\tau.\sigma)$ does not begin with τ , there must be some point in the computation of $\theta(\tau.\sigma)$ where a chain $\tau.\rho$ is transformed into a chain not beginning with τ . But then, since τ consists of value attributes, ρ must begin with an input, so it is an input chain and again we are done.

■

As with Lemma 4.2.10, the proof of Lemma 4.4.10 gives us a way to compute the input or value chain ρ on C such that $[\sigma]^\omega(c)$ and $[\rho]^\omega(c)$ must be equal. We simply go through the process of computing $\theta(\tau.\sigma)$, stopping when an intermediate result

$\tau.\rho$, with ρ an input or value chain on C is produced. We will denote the chain ρ so produced by $\theta_C(\sigma)$.

Lemma 4.4.10 tells us that given particular values for all the input chains, the values of all attribute chains can be characterized by the value chains.⁹ We can therefore naturally view \mathcal{P}_C as defining a CPF from the input chains to the value chains. The definition of this CPF is inductive. For a class of lowest rank, all its attributes are simple, and the CPF for each of its attributes is given. For a class that contains complex attributes, we assume inductively that we have already obtained a CPF for each of its complex attributes. We can then define a BN over all the attributes of the class, assigning an arbitrary CPF to the inputs (we use a uniform CPF for the sake of definiteness). Any assignment of values to the input and value chains corresponds to an assignment of values to the nodes of the BN, so we can use the probability distribution defined by the BN to define the CPF of the class.

Definition 4.4.11: Let C be an OOBN class, with input attributes H_1, \dots, H_m , and value attributes A_1, \dots, A_n . We define the *hierarchical BN equivalent of C* , denoted \mathcal{B}_C^H , as follows:

- \mathcal{B}_C^H contains a node X_A for each (input or value) attribute A of C .
- If A is simple, $Val[X_A]$ is $Val[A]$. If A is complex with range type C' , $Val[X_A]$ is $Val[C']$. Note that in both cases $Val[X_A] = Val[\sigma]$, where σ is the set of simple attribute chains beginning with A and continuing with value attributes.

⁹There may be some redundancy amongst the input chains; some pairs of input chains may necessarily have the same value. However, there is no redundancy amongst the value chains, because distinct value chains correspond to distinct basic variables. We cannot normally determine, based on the model of C alone, which input chains must have the same value, because that depends on the bindings of the inputs, which are specified in some other class C' that has a component function whose range type is C . In fact, there may be more than one such C' , and the bindings of the inputs of C may be different in each of them, so in general the determination of which input chains necessarily have the same value may vary between different entities of C . In any case, this is not a concern. Since the values of all input chains have already been generated before **GenerateEntity** is called, we do not worry about the fact that some of them have the same value. It is enough that we can identify the exact set of attribute chains, namely the value chains, that fully characterize an entity of C .

- $\mathcal{G}[\mathcal{B}_C^H] = \mathcal{G}[C]$. The set of parents of X_A in $\mathcal{G}[C]$ will be denoted \mathbf{U}_A . Note that if A is an input, X_A has no parents.
- For each simple value attribute A , with attribute chain parents v_1, \dots, v_ℓ in \mathcal{P}_C , the CPF of X_A in \mathcal{B}_C^H is a function $F_A : \text{Val}[\mathbf{U}_A] \times \text{Val}[X_A] \rightarrow [0, 1]$, defined by

$$F_A(\mathbf{u}, x) = \text{CPF}_A(v_1(\mathbf{u}), \dots, v_\ell(\mathbf{u}), x),$$

where CPF_A is the CPF associated with A_i in \mathcal{P}_C . The notation $v_j(\mathbf{u})$ denotes the projection of \mathbf{u} onto the attribute chain v_j .

- For each complex value attribute A of C with range type C' , let the inputs of C' be H_1, \dots, H_ℓ . The CPF of X_A in \mathcal{B}_C^H is a function $G_A : \text{Val}[\mathbf{U}_A] \times \text{Val}[X_A] \rightarrow [0, 1]$, defined by

$$G_A(\mathbf{u}, x) = \text{CPF}_{C'}^H(\Theta[A.H_1](\mathbf{u}), \dots, \Theta[A.H_\ell](\mathbf{u}), x),$$

where $\text{CPF}_{C'}^H$ is as defined below. Again, the notation $\Theta[A.H_j](\mathbf{u})$ denotes the projection of \mathbf{u} onto the chain $\Theta[A.H_j]$.

- If H is an input attribute, the CPF for X_H in \mathcal{B}_C^H is the uniform distribution over $\text{Val}[X_H]$.

For a particular value \mathbf{h} for the input chains of C and \mathbf{a} for the value chains of C , and an attribute A of C , let $x_A(\mathbf{h}, \mathbf{a})$ denote the projection of \mathbf{h}, \mathbf{a} onto A . Note that $x_A \in \text{Val}[X_A]$. We define the function CPF_C^H from $\text{Val}[\boldsymbol{\sigma}] \times \text{Val}[C]$ to $[0, 1]$, (where $\boldsymbol{\sigma}$ is the set of input chains on C), by

$$\begin{aligned} \text{CPF}_C^H(\mathbf{h}, \mathbf{a}) = P_{\mathcal{B}_C^H} (& X_{A_1} = x_{A_1}(\mathbf{h}, \mathbf{a}), \dots, X_{A_n} = x_{A_n}(\mathbf{h}, \mathbf{a}) \mid \\ & X_{H_1} = x_{H_1}(\mathbf{h}, \mathbf{a}), \dots, X_{H_m} = x_{H_m}(\mathbf{h}, \mathbf{a})) \blacksquare \end{aligned}$$

Note the inductive nature of the definition: \mathcal{B}_C^H is defined in terms of $\text{CPF}_{C'}^H$ for various C' of lower rank than C , while $\text{CPF}_{C'}^H$ is in turn defined in terms of $\mathcal{B}_{C'}^H$. Note also that since the top-level class C_T has no inputs, $\text{CPF}_{C_T}^H$ defines a probability

distribution over the value chains on C_T , i.e., over the basic variables. One would hope that this distribution is the same as the one defined by the flat BN equivalent. Indeed, the following theorem shows that this is the case.

Theorem 4.4.12: *Let \mathcal{O} be an OOBN with top-level class C_T . For any assignment \mathbf{x} of values for all the basic variables, $CPF_{C_T}^H(\mathbf{x}) = P_{\mathcal{B}_{\mathcal{O}}^F}(\mathbf{x})$.*

Proof: The idea behind the proof is as follows: $P_{\mathcal{B}_{\mathcal{O}}^F}(x)$ is the product of CPF entries, one for each basic variable. $CPF_{C_T}^H(x)$ is also the product of CPF entries, one for each attribute of C_T . If we can show that the entry for attribute A in $CPF_{C_T}^H(x)$ is the product of the entries in $\mathcal{B}_{\mathcal{O}}^F$ for attribute chains beginning with A , we will be done. We show this by proving the following statement.

Let σ be a complex component chain on C_T , with range type C . By Theorem 4.2.16 there is a unique possible world corresponding to the assigned values \mathbf{x} to the basic variables, so for any value chain or input chain ρ on C , $[\rho]^\omega(I_T.\sigma)$ is determined. Let \mathbf{h} and \mathbf{a} be the values of the input chains and value chains respectively. The following statement holds:

$$CPF_C^H(\mathbf{a} \mid \mathbf{h}) = \prod_{X=I_T.\sigma.\rho} CPF_X(x \mid u_1, \dots, u_m),$$

where u_i is the value in \mathbf{x} assigned to parent U_i of X . In words, the probability according to CPF_C^H of the values of the value chains given the input chains is equal to the product, over basic variables beginning with σ , of the CPF entry for the value of that basic variables given its parents in $\mathcal{B}_{\mathcal{O}}^F$.

The proof of this statement is by induction on the rank of C . If C is of lowest rank, it contains only simple attributes. Let A be a value attribute of C . There is a corresponding basic variable $X = I_T.\sigma.A$. For any parent v of A , the value of v determined by \mathbf{h}, \mathbf{a} must be the same as the value of $\theta(I_T.\sigma.v)$, which is the corresponding parent U of X . So the CPF entry for A is the same as the CPF entry for the corresponding X . So

$$CPF_C^H(\mathbf{a} \mid \mathbf{h}) = \prod_A CPF_A(a \mid \mathbf{v}(\mathbf{a}, \mathbf{h})) = \prod_{X=I_T.\sigma.A} CPF_X(x \mid \mathbf{u})$$

as required.

For the induction step, an attribute of C is either simple, or it is complex with range type of lower rank than C . If A is a simple value attribute of C , the CPF entry for A is the same as the entry for the corresponding basic variable X , as before. Since X is the only basic variable of the form $I_T.\sigma.A.\rho$, the CPF entry for A is the product of entries of basic variables of form $I_T.\sigma.A.\rho$.

Meanwhile, if A is complex with range type C' , let \mathbf{h}_A and \mathbf{a}_A be the values of the input chains and value chains of C' , as determined by \mathbf{x} . By the inductive hypothesis, $CPF_{C'}^H(\mathbf{a}_A \mid \mathbf{h}_A) = \prod_{X=I_T.\sigma.A.\rho} CPF_X(x \mid u_1, \dots, u_m)$. Since the value of $\Theta[A.H]$ must agree with the value assigned to H in \mathbf{h}_A , the value of the function G_A in Definition 4.4.11 must be equal to $CPF_{C'}^H(\mathbf{a}_A \mid \mathbf{h}_A)$. So the CPF entry for A is equal to the product of CPF entries for basic variables X that begin with $\sigma.A$.

Since for all value attributes A of C , the CPF entry for A is equal to the product of entries for basic variable beginning with $\sigma.A$, we have

$$CPF_C^H(\mathbf{a} \mid \mathbf{h}) = \prod_A \prod_{X=I_T.\sigma.A.\rho} CPF_X(x \mid \mathbf{u}) = \prod_{X=I_T.\sigma.\tau} CPF_X(x \mid \mathbf{u})$$

as required. ■

In this subsection we have defined the semantics of a class C in a natural way, in terms of a conditional probability distribution over the values of the value attributes of C given the inputs of C . Defining the semantics this way ties it in very nicely with the hierarchical nature of the representation language. In addition, the semantics is modular: the same class can be reused in different models, and has the same meaning in the different models. As Theorem 4.4.12 shows, defining things this way results in the same semantics that we obtained using the flat BN equivalent.

4.5 Structured Inference

4.5.1 Interfaces and Encapsulation

Since a possible world is fully characterized by the values of the basic variables, we can phrase any query in an OOBN as a query about values of basic variables. As in BNs, a query seeks a probability distribution over a certain set \mathbf{Q} of query variables, given values \mathbf{e} for a set \mathbf{E} of evidence variables. The inference task is to compute $P_O(\mathbf{Q} \mid \mathbf{E} = \mathbf{e})$.

The flat BN equivalent of an OOBN discussed in the last section provides us with a way of answering any query on the OOBN. We simply construct the flat BN equivalent, and answer any query using the standard **Variable Elimination** algorithm described in Section 3.7. However, when we do inference in this way, we lose all of the structure of the OOBN by converting it into a flat BN. As discussed in the introduction, we want to be able to exploit the OOBN structure to perform efficient inference. In particular, we want to exploit both the hierarchical structure, and the fact that many of the instances share the same class model.

In developing a structured inference algorithm, we note that the hierarchical structure of an OOBN gives us conditional independence information, in addition to the standard information present in the graphs of the OOBN objects.

Definition 4.5.1: Let $I = I_T.\sigma$ be an instance in an OOBN, and let X be a basic variable. We say that X is *inside* I if $X = I.\rho$. X is *importable* by I if $X = I_T.\theta(\sigma.H.\rho)$, where H is some input of the range type of σ . X is *outside* I if it is neither inside nor importable by I . X is *imported* by I if it is importable by I and it is a parent of some basic variable inside I . X is *exported* by I if it is inside I and it is the parent of some basic variable outside I . X is *encapsulated* by I if it is inside I and is not exported by I . X is in the *interface* of I if it is either imported or exported by I . ■

Example 4.5.2: Figure 4.3 shows the interface of `Hard-Drive.Has-Drive-Mechanism` in the flat BN equivalent of the `Hard-Drive`. The interface consists of the variables

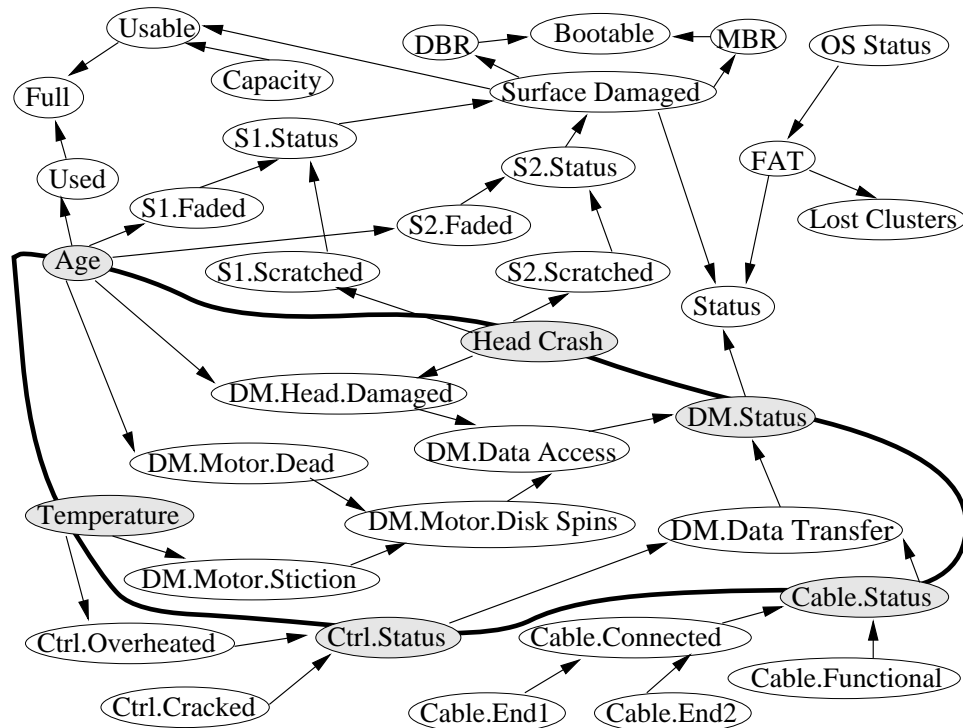


Figure 4.3: Interface of Drive-Mechanism object within Hard-Drive

Age, Temperature, Has-Controller.Status, Has-Cable.Status and Head-Crash imported by Has-Drive-Mechanism, and the single exported variable Has-Drive-Mechanism.Status. The black curve encloses that part of the network consisting of variables inside or imported by Has-Drive-Mechanism. The variables strictly inside the curve are encapsulated within Has-Drive-Mechanism. ■

As we now show, the interface of an object renders the variables inside the object conditionally independent of those outside the object. We can therefore use the object interfaces as separators when performing probabilistic inference.

Lemma 4.5.3: *Let $I = I_T.\sigma$ be an instance in an OOBN, let $I_T.\sigma.\rho$ be a basic variable inside I , and let $I_T.\tau$ be a child of $I_T.\sigma.\rho$ that is not inside I . Then $I_T.\tau$ is outside I .*

Proof: We must show that $I_T.\tau$ is not importable by I . Suppose the contrary. Then $\tau = \theta(\sigma.H.\tau')$ for some input H of I . Let $\sigma = \sigma'.A$. Using the partial order \leq^* from

the proof of Theorem 4.4.6,

$$\tau = \theta(\sigma'.A.H.\tau') = \theta(\sigma'.\Theta[A.H].\tau') \leq^* \sigma'.\Theta[A.H].\tau' <^* \sigma'.A = \sigma,$$

using Statement (1) from that proof. Since \leq^* is lexicographic, it follows that $\tau <^* \sigma.\rho$. But, writing $\tau = \tau'.B$, we must have $\sigma.\rho = \theta(\tau'.v)$, where v is a parent of B in the probability model for the range type of τ' , since $I_T.\tau$ is a child of $I_T.\sigma.\rho$. By Statement (3) of the proof of Theorem 4.4.6, $\sigma.\rho \leq^* \tau$. This is impossible since \leq^* is a partial order. ■

Theorem 4.5.4: *Let $I = I_T.\sigma$ be an instance in \mathcal{O} , and \mathbf{X} , \mathbf{Y} and \mathbf{Z} be the set of variables inside I , outside I , and in the interface of I respectively. Then $I(\mathbf{X}, \mathbf{Y} \mid \mathbf{Z})$.*

Proof: We show that \mathbf{X} is d-separated from \mathbf{Y} by \mathbf{Z} in $\mathcal{B}_{\mathcal{O}}^F$. The result will then follow from Theorem 3.5.5.

Consider any path π between variables $X \in \mathbf{X}$ and $Y \in \mathbf{Y}$. Since X is inside I and Y is not, π must contain an edge in which one end is inside I and the other is not. There are two possibilities. In the first case, the child is inside I , while the parent is imported by I . In the second case, the parent is inside I . By Lemma 4.5.3, the child must be outside of I . Therefore the parent is exported by I . In either of the two cases, then, the parent is a variable Z in the interface of I . So π passes through a variable $Z \in \mathbf{Z}$ and does not have converging arrows at Z . Therefore the path is blocked by \mathbf{Z} . Since all paths between \mathbf{X} and \mathbf{Y} must be blocked by \mathbf{Z} , \mathbf{X} is d-separated from \mathbf{Y} by \mathbf{Z} . ■

Example 4.5.5: We can see this theorem demonstrated pictorially in Figure 4.3. First of all, all the variables encapsulated within **Has-Drive-Mechanism** are enclosed in the black curve that passes through the interface variables. This is not enough, on its own, to guarantee d-separation. What is needed in addition is the fact that no interface variable is the child of variables that are both inside and outside the curve, which can readily be seen from the figure. ■

4.5.2 Structured Variable Elimination

Theorem 4.5.4 captures our intuition that the effect of any variable encapsulated inside an instance on the variables outside the instance can be completely captured by the variables in the interface. This suggests a way to perform inference hierarchically. In order to solve a query within a given instance I , we eliminate each contained instance J , replacing it with a factor over the interface of J . Eliminating J , of course, results in a recursive query on J . Once all the contained instances within I have been eliminated, we can use standard **Variable Elimination** to solve the query on I . The hierarchical inference algorithm, called **Structured Variable Elimination (SVE)**, takes four arguments: an instance I in an OOBN, a set σ of query variables, a set ρ of evidence variables, and the evidence, which is a value $e \in \text{Val}[\rho]$. **SVE** returns a factor over the query variables. The query and evidence variables are all value chains on the type of I . The algorithm is as follows:

Algorithm StructuredVariableElimination(I, σ, ρ, e)

- 1 Let C be the type of I .
- 2 For each simple value attribute A of C do:
- 3 If A is a chain $\rho \in \rho$
- 4 Let a be the value assigned to A in e .
- 5 $g_A = \text{CPF}_A[A = a]$.
- 6 Else
- 7 $g_A = \text{CPF}_A$.
- 8 For each complex value attribute A of C do:
- 9 Let C' be the class of $I.A$.
- 10 Let $\sigma' = \text{Imports}(C') \cup \text{Exports}(C') \cup \{\sigma' : A.\sigma' \in \sigma\}$.
- 11 Let $\rho' = \{\rho' : A.\rho' \in \rho\}$.
- 12 Let e' be the value in $\text{Val}[\rho']$,
- 13 such that for each $\rho' \in \rho'$,
- 14 the value assigned to ρ' in e'
- 15 is the same as the value assigned to $A.\rho'$ in e .
- 16 $f_A = \text{StructuredVariableElimination}(I.A, \sigma', \rho', e')$.

```

17   $g_A = \mathbf{Rename}(f_A, \psi)$  where
18   $\psi(\tau) = \theta_C(A.\tau)$ .

19   $\mathbf{f} = \{g_A : A \text{ is a value attribute of } C\}$ .
20  Let  $\tau = \{\tau : \tau \text{ is mentioned by some } f \in \mathbf{f}\} - \sigma$ .
21  For each  $\tau \in \tau$  do
22    Let  $\mathbf{g}$  be  $\{g \in \mathbf{f} : g \text{ mentions } \tau\}$ .
23     $h_\tau = \prod \mathbf{g}$ .
24     $k_\tau = \sum_\tau h_\tau$ .
25     $\mathbf{f} = \mathbf{f} - \mathbf{g} \cup \{k\}$ .
26   $\mathbf{f} = \prod \mathbf{f}$ .
27  Return  $\mathbf{f}$ .

```

Let us examine the algorithm line by line. Its general structure is similar to that of **VE**. It consists of two phases. In the first phase (lines 2-18), a factor g_A is prepared for each value attribute A of C . Each of these factors will be a factor over attribute chains on C . In the second phase (lines 19-27), all attribute chains except the query chains are eliminated from the set of factors. These two phases are similar to those of **VE**, the main difference being that now much more work is done in preparing the factors.

In the first phase, the preparation of the factor for an attribute depends on whether the attribute is simple or complex. For a simple attribute A (lines 2-7), if A is actually one of the chains for which we have evidence, g_A is prepared by conditioning the CPF of A on its observed value. Otherwise g_A is just its CPF in \mathcal{P}_A .

If A is complex (lines 8-18), a recursive call is required to compute a factor over the interface of $I.A$. Lines 10-14 compute the arguments for the recursive call. The set of query chains (line 10) consists of the interface of $I.A$, together with any query variables in σ that are actually inside of $I.A$. If a query variable σ begins with A , it is rewritten as $A.\sigma'$, and passed as the query variable σ' to the recursive call. Thus the query variables for the recursive call will be chains on C' .

The functions **Imports**(C') and **Exports**(C') are defined as follows. **Imports**(C') is the set of chains on C' beginning with an input attribute and followed by value

attributes. Such a chain is called an *input-value chain* on C' . $\mathbf{Exports}(C')$ is the set of chains on C' that use only output attributes.¹⁰ All variables imported by $I.A$ can be captured by the input-value chains, as the following lemma shows.

Lemma 4.5.6: *Let $I = I_T.\sigma$, let C be the type of $I.A$, let $X = I.A.\rho.B$ be a basic variable inside $I.A$, and let C' be the type of $I.A.\rho$. For any parent v of B in $\mathcal{P}_{C'}$, let $U = I_T.\theta(\sigma.A.\rho.v)$ be the corresponding parent of X . Suppose that U is imported by $I.A$. Then there is an input-value chain τ on C , such that in any possible world ω , $[U]^\omega = [\tau]^\omega(I.A)$. The chain τ is $\theta_C(\rho.v)$.*

Proof: By Definition 4.3.1, all but the first attribute in v must be an output attribute. Therefore, $\sigma.A.\rho.v$ contains at most one input attribute. Also, by Definition 4.2.3, $\Theta[A.H]$ can contain at most one input attribute. Each step of computing θ consists of replacing some $A.H$ with $\Theta[A.H]$, so it cannot increase the number of inputs. Now, by Lemma 4.4.10, the process of computing $\theta(\sigma.A.\rho.v)$ must at some point produce a chain $\sigma.A.\tau$, where τ is either an input chain or value chain on C . But τ cannot be a value chain, because then U would be $I_T.\sigma.A.\tau$, which is inside $I.A$ and not imported by it. So τ must begin with an input, and since it contains at most one input, it must be an input-value chain. The chain τ is $\theta_C(\rho.v)$ by definition of θ_C . ■

Lines 11-15 prepare the evidence passed to the recursive call to **SVE** on $I.A$. All evidence will be incorporated into the factor produced by the recursive call. The recursive call itself is performed in line 16. It returns a factor over the query chains on $I.A$, that is computed from the probability model for C' , and the values of evidence chains inside $I.A$. The returned factor is over chains on C' , but we need g_A to be a factor over chains on C . We therefore process f_A by renaming the variables it mentions into chains on C (lines 17-18). The **Rename** operator takes as an argument a *renaming function*, which maps variable names to variable names. In our case, the renaming function ψ maps a chain σ on C' to $\theta_C(A.\sigma)$, its corresponding chain on C .

¹⁰ $\mathbf{Imports}(C')$ and $\mathbf{Exports}(C')$ are actually supersets of the chains on C' of the variables imported and exported by $I.A$. Theorem 4.5.4 still holds using this set of variables as the interface set. This definition is simple, easy to compute, and is the same for all instances of a class, but it may result in larger interfaces than are actually necessary. In the next chapter, we will see a way to compute the actual interface during the process of solving a query.

Let us examine exactly what this does. Every chain σ mentioned by f_A is either a value chain or an input-value chain on C' . If σ is a value chain, $A.\sigma$ is a value chain on C , so $\psi(\sigma) = A.\sigma$, exactly as we would expect. If σ is an input-value chain $H.\sigma'$, $\theta_C(A.\sigma) = \theta_C(\Theta[A.H].\sigma')$. Since all but the first attribute of $\Theta[A.H]$ is an output, $\Theta[A.H].\sigma'$ is either a value chain or input-value chain on C , and $\psi(\sigma) = \Theta[A.H].\sigma'$.

One might think that the definition of the **Rename** operator would be trivial: **Rename**(f, ψ) would be exactly the same as f except that the names of the variables have changed. In fact, it is a little more subtle. The reason is that the renaming function ψ may not be one-to-one, in which case the resulting factor will mention fewer variables. This situation can arise for our renaming function. For example, suppose C' has two simple inputs H_1 and H_2 , and that $\Theta[A.H_1] = \Theta[A.H_2]$.¹¹ Then, if both H_1 and H_2 are query chains on $I.A$, they will both be mentioned by f_A , while $\psi(H_1) = \psi(H_2)$. We must therefore be a little careful in the definition. If f is a factor over the variables \mathbf{X} , **Rename**(f, ψ) is a factor over the image of \mathbf{X} under ψ , as follows.

Definition 4.5.7: Let f be a factor over variables \mathbf{X} , and let ψ be a renaming function from \mathbf{X} to some other set of variables \mathbf{Y} , such that for each X , $\text{Val}[\psi(X)] = \text{Val}[X]$. Then **Rename**(f)(ψ) is the factor g defined as follows: For any set of values $\mathbf{y} \in \text{Val}[\mathbf{Y}]$, let $\psi^{-1}(\mathbf{y})$ be the set of values $\mathbf{x} \in \text{Val}[\mathbf{X}]$ such that for each X, Y where $Y = \psi(X)$, $x = y$. Then $g(\mathbf{y}) = f(\psi^{-1}(\mathbf{y}))$. ■

By the end of the first phase of the algorithm, a factor g_A has been computed for each value attribute A of C . The variables mentioned by these factors are value chains and input-value chains on C . Specifically, they are the query variables σ , the simple attributes of C , and the interface variables of the complex attributes of C . The remainder of the algorithm performs standard **VE** over these variables, eliminating all but the query variables σ .

Every value chain on C , except for those that are query variables in σ , will be eliminated at some point during the **SVE** computation. A chain τ that is encapsulated inside $I.A$ will be eliminated during the recursive call on $I.A$. The chains τ

¹¹More generally, if H_1 and H_2 are complex, we could have $\Theta[A_i.H_1^i] = \sigma$, and $\Theta[A_i.H_2^i] = \sigma.\rho$. Then $H_1^i.\rho.\tau$ and $H_2^i.\tau$ will both be renamed to $\sigma.\rho.\tau$.

eliminated during the **VE** phase of the **SVE** computation on I consist of those value chains on C that are not query chains and are not encapsulated inside $I.A$ for some attribute A of C .

What is the meaning of the factor f returned by **SVE**? Note that it is unnormalized. It is equal to $\sum_{\tau} \prod_i g_i$. The g_i incorporate the CPFs for all the value chains on C , and are also conditioned on the evidence \mathbf{e} . Let us divide the query variables into value chains σ_K and input-value chains σ_H . The factor f is in fact equal to $P(\sigma_K, \rho = \mathbf{e} \mid \sigma_H)$. In words, the **SVE** call on instance I computes, for any assignment of values \mathbf{x}_K to the query variables inside I and the variables exported by I , and \mathbf{x}_H to the variables imported by I , the conditional probability that the query and exported variables have the values \mathbf{x}_K and the evidence variables have the values \mathbf{e} , given that the imported variables have the values \mathbf{x}_H . Since the top-level instance has no imports or exports, the top-level call returns $P(\sigma, \rho = \mathbf{e})$, where σ and ρ are the top-level query and evidence variables. The answer to a query that we want to return is actually $P(\sigma \mid \rho = \mathbf{e})$. We can compute it by normalizing $P(\sigma, \rho = \mathbf{e})$, dividing it by the normalizing factor $\sum_{\mathbf{x}} P(\sigma = \mathbf{x}, \rho = \mathbf{e}) = P(\rho = \mathbf{e})$.

4.5.3 Reuse of Inference

In order to answer any query in an OOBN, we just have to make a call to **SVE** on the top-level instance I_T . This will result in a recursive call for each of the instances in the OOBN. In so doing, the algorithm will take advantage of the hierarchical structure of the OOBN, but it will fail to exploit the fact that many instances share the same class model. In fact, all instances of a class have the same probability model. If we examine the behavior of the **SVE** algorithm, we see that it does not depend on specific details about the instance I on which it is called, but only on the probability model of the class C of I . We can reuse computation from one **SVE** call for a class to another call for the same class; as long as the other arguments are the same, the result will be the same.

The degree to which the reuse of computation can be exploited in this way depends on how many query variables and evidence variables there are in a query. If there are

few query variables and little evidence, the scope for reuse is very high. In particular, if there are no query or evidence variables in the class, the result of calling **SVE** will just be a conditional distribution over the exported variables given the imported variables. We call this distribution the *iconized distribution* of the class. If we have no particular information or interest in the internals of an instance of a class, we can represent that instance as an “icon”, and the iconized distribution of the class is all we need to compute the effect of that instance on the probability model.

We achieve the desired effect by changing the first argument of **SVE** from an instance to a class object, and by maintaining a cache with the answers to queries performed on a class. The modified algorithm is as follows:

Algorithm StructuredVariableElimination(C, σ, ρ, e)

```

If  $\langle C, \sigma, \rho, e \rangle$  is in Cache
  Return Cache[ $\langle C, \sigma, \rho, e \rangle$ ].
Else
   $f = \text{UncachedSVE}(C, \sigma, \rho, e)$ .
  Cache[ $\langle C, \sigma, \rho, e \rangle$ ] =  $f$ .
  Return  $f$ .
```

Procedure UncachedSVE(C, σ, ρ, e)

```

For each simple value attribute  $A$  of  $C$  do
  If  $A$  is a chain  $\rho \in \rho$ 
    Let  $a$  be the value assigned to  $A$  in  $e$ 
     $g_A = CPF_A[A = a]$ .
  Else
     $g_A = CPF_A$ .
For each complex value attribute  $A$  of  $C$  do
  Let  $C'$  be the range type of  $A$ .
  Let  $\sigma' = \text{Imports}(C') \cup \text{Exports}(C') \cup \{\sigma' : A.\sigma' \in \sigma\}$ .
  Let  $\rho' = \{\rho' : A.\rho' \in \rho\}$ .
  Let  $e'$  be the value in  $Val[\rho']$ ,
```

such that for each $\rho' \in \rho'$,
 the value assigned to ρ' in e'
 is the same as the value assigned to $A.\rho'$ in e
 $f_A = \text{StructuredVariableElimination}(C', \sigma', \rho', e')$.
 $g_A = \text{Rename}(f_A, \psi)$ where
 $\psi(\tau) = \theta_C(A.\tau)$.

$f = \{g_A : A \text{ is a value attribute of } C\}$.
 Let $\tau = \{\tau : \tau \text{ is mentioned by some } f \in f\} - \sigma$.
 For each $\tau \in \tau$ do
 Let g be $\{g \in f : g \text{ mentions } \tau\}$.
 $h_\tau = \prod g$.
 $k_\tau = \sum_\tau h_\tau$.
 $f = f - g \cup \{k\}$.
 $f = \prod f$.
 Return f .

By maintaining a cache, we can reuse computations not only within one query, but also across different queries for the same model. We can also reuse computation between different models in which the same class appears. The cache needs to be purged of all solutions for queries on a class whenever the class model changes. Also, if class C is defined in terms of another class C' , the cache needs to be purged of solutions for queries on C when the model for C' changes. However, solutions obtained for C can be kept when new subclasses of C are created, or when C is used in defining another class. The class C can be envisioned as providing a library of answers to queries. Answers to the most common queries are provided immediately, while less common queries require computation to solve.

Example 4.5.8: We illustrate the **SVE** algorithm by showing what happens when we compute $P(\text{DM.Motor.Stiction} \mid \text{Temperature} = \text{Cold}, \text{Status} = \text{Unreadable})$. We

start out with a call to

$$\mathbf{SVE}(\text{Hard-Drive}, \{\text{DM.Motor.Stiction}\}, \{\text{Temperature}, \text{Status}\}, \\ \langle \text{Temperature} : \text{Cold}, \text{Status} : \text{Unreadable} \rangle)$$

Processing the simple attributes is straightforward, just as in BNs. We will condition the CPFs of the two evidence variables on the values given. Then we process the complex attributes, each one resulting in a recursive call. When we process the attribute **DM**, σ^{DM} is set to consist of the imports and exports of the class **Drive-Mechanism**, as well as the query variable **Motor.Stiction**. Since none of the evidence variables are inside **DM**, ρ^{DM} and e^{DM} are empty.¹² This results in a recursive call to

$$\mathbf{SVE}(\text{Drive-Mechanism}, \\ \{\text{Status}, \text{Motor.Stiction}, \text{Connected}, \text{Controller-Ok}, \text{Temperature}, \text{Age}, \text{Head-Crash}\}, \\ \emptyset, \langle \rangle).$$

The result of this recursive call will be a factor over the variables **Status**, **Motor.Stiction**, **Connected**, **Controller-Ok**, **Temperature**, **Age**, and **Head-Crash**, expressing

$$P(\text{Status}, \text{Motor.Stiction} \mid \text{Connected}, \text{Controller-Ok}, \text{Temperature}, \text{Age}, \text{Head-Crash}).$$

The renaming operation will change the names of the variables in this factor to **DM.Status**, **DM.Motor.Stiction**, **Cable.Status**, **Ctrl.Status**, **Temperature**, **Age**, and **Head-Crash**.

The **SVE** algorithm will continue processing all the other complex attributes.

¹²Even though the evidence variable **Temperature** is also imported by **DM**, we do not need to pass it to the recursive call. The recursive call will return a factor over the ρ^{DM} that considers all possible values of **Temperature** to be possible, but we have already conditioned the CPF for **Temperature** to assign probability 0 to all values other than *Cold*.

We could, on the other hand, have designed the algorithm so that the values of imported variables are also passed as evidence to the recursive calls. There is an interesting tradeoff here. On the one hand, passing the evidence to the recursive call reduces the size of the resulting factor, since it is already conditioned on the evidence. In addition, in the presence of context specific independence, it may actually simplify the inference [12, 101]. On the other hand, since the evidence is part of the signature of the **SVE** call, passing evidence variables when it is not necessary to do so reduces the possibility of caching and reusing computation.

Note that only one recursive call

$$\mathbf{SVE}(\text{Disk-Surface}, \{\text{Age}, \text{Head-Crash}, \text{Status}\}, \emptyset, \langle \rangle)$$

is needed for the different disk surfaces. Also note that since there are no query or evidence variables inside any of the complex attributes other than **DM**, the queries on those attributes will just ask for the iconized distributions of their classes.

After eliminating all the complex attributes, the algorithm will be left with a set of attribute chains. The chains in this set consist of the simple attributes of **Hard Drive**, the chains exported by the complex attributes of **Hard Drive**, and the chain **DM.Motor.Stiction**. Standard **VE** is now used to eliminate all the chains other than **DM.Motor.Stiction**. The eliminated variables are **Age**, **Temperature**, **Head-Crash**, **S1.Status**, **S2.Status**, **Surface-Damaged**, **Usable**, **Capacity**, **Used**, **Full**, **DBR**, **MBR**, **Bootable**, **Ctrl.Status**, **Cable.Status**, **DM.Status**, **OS-Status**, **FAT**, **Lost-Clusters** and **Status**. ■

4.5.4 Complexity

How expensive is the cost of solving a query in an OOBN? Well, if we ignore for the moment the reuse of computation between different instances of the same class, there is a call to **SVE** for each instance in the OOBN. Thus the total cost is the sum of the costs performed directly within each call.

In the analysis that follows, we use notation similar to that of the previous chapter: b is an upper bound on the number of values in the domain of a simple attribute in the OOBN, and the number of variables mentioned by a factor f is denoted by $|f|$. The size of f is therefore $b^{|f|}$. As in the algorithm, σ will denote the set of query chains (the second argument to **SVE**), while τ will denote the set of eliminated chains. σ_A and ρ_A will denote the set of query and evidence chains respectively for the recursive call on complex attribute A . We will let n denote the number of variables in $\sigma \cup \tau$, while v will denote the total number of attribute chains mentioned during the course of the **SVE** computation, including evidence chains, and the chains on the types of the complex attributes passed as arguments to the recursive calls.

Theorem 4.5.9: *The time and space cost of computation performed directly within a*

single call to **SVE** for an object X is $O(nb^M + v)$, where M is $\max(\max_A(|\sigma_A|), M_0)$, M_0 being the maximal clique size of the induced graph of X for the variable elimination phase.

Ordinarily, M will be equal to M_0 , since there is a clique containing the variables named in g_A for each A . However, because of the **Rename** operator, it is possible that f_A mentions more variables than g_A . To take this possibility into account, we have to ensure that $M \geq |f_A| = |\sigma_A|$.

Proof: The work done directly within a single call to **SVE** can be divided into two phases:

1. A preprocessing phase, which involves:
 - (a) Preparing the factor g_A for each simple value attribute A_i (cost for each A_i is $O(b^{|CPF_A|})$, which is $O(b^M)$ since $|CPF_A| \leq M_0 \leq M$;
 - (b) Computing the arguments to the recursive call for each complex value attribute A_i . Using the given definition of **Imports** and **Exports**, the argument σ_A can be produced in time and space $O(|\sigma_A|)$, and $|\sigma_A| \leq v$. The arguments ρ_A and e_A can be produced in time and space $O(|\rho_A|)$, and $|\rho_A| \leq v$. So the cost for this step is $O(v)$.
 - (c) Renaming and storing the result f_A of the recursive call for each complex A . Cost for each is $O(b^{|f_A|})$ which is $O(b^M)$.

Total cost for the preprocessing phase is therefore $O(nb^M + v)$.

2. The variable elimination phase. By Theorem 3.7.9, the cost of this phase is $O(nb^{M_0})$.

Putting all these together, we see that the work done in a single call to **SVE**, not counting the recursive computations, is $O(nb^M + v)$. ■

Just as in BNs, we see that the critical factor is M . The crucial point is that OOBNs provide us with a way of keeping M small using only *local* considerations. The basic idea is simple. We stipulate a bound k on the number of variables in the

interface of an object, and m on the width of the dependency graph of a class, relative to some elimination order. If we take a dependency graph $\mathcal{G}[C]$, and replace each complex attribute A with a clique over the variables mentioned by g_A , we essentially get a graph \hat{G} of the **VE** computation. Furthermore, the size of the largest clique in this graph is at most $(m + 1)k$.¹³ However, this graph is not quite good enough. Recall from Section 3.7 that when we perform a **VE** computation with more than one query variable, the query variables must be connected to each other in the induced graph. In our case, we wish to return a factor over all the query variables σ , so we have to connect each pair of query variables in \hat{G} . After doing this, \hat{G} may no longer be triangulated; to make sure that it is triangulated without actually searching for a triangulation, we need to connect all the query variables to all other variables. The resulting graph will certainly be triangulated, but as we now prove its width will now be less than $(m + 2)k + q$, q being the number of query variables not in the interface of C . First we need the following definition, relating an elimination ordering used in the **VE** phase of **SVE** to an ordering over the attributes of C .

Definition 4.5.10: Let d be an ordering over the attributes of a class C , and σ a set of chains on C . An ordering D over σ is consistent with d , if, for any two chains σ_1 and σ_2 in σ , if $\sigma_1 = A_1.\rho_1$ and $\sigma_2 = A_2.\rho_2$, and A_1 precedes A_2 in d , σ_1 precedes σ_2 in D . ■

Theorem 4.5.11: *Let I be an OOBN instance, of class C , and let σ be the set of query chains for an **SVE** call on I . Let k be an upper bound on the number of variables in the interface of a complex attribute of C , or in the interface of C . Let q be the number of additional query variables in the query Q not in the interface of C . Suppose that there is an ordering d over the attributes of C such that the induced width of $(\mathcal{G}[C, d])$ is m . Let D be an ordering over the chains τ eliminated during the **VE** phase of the **SVE** computation that is consistent with d . Then the induced width M for the **VE** phase of the **SVE** computation is less than $k(m + 2) + q$.*

¹³The $m + 1$ factor arises from the fact that the width of a graph is one less than the size of its maximal clique.

Proof: Let \tilde{G} be the induced graph for $(\mathcal{G}[C], d)$. By hypothesis, the width of (\tilde{G}, d) is m . Let \hat{G} be the graph over $\sigma \cup \tau$ such that there is an edge between σ_1 and σ_2 if either is a query variable in σ , or if they both begin with the same attribute, or they begin with attributes A_1 and A_2 that are connected in \tilde{G} . Let \hat{D} be an ordering over $\sigma \cup \tau$ formed by prepending any ordering over the query variables σ to D . We claim that the width of (\hat{G}, \hat{D}) is less than $k(m+2) + q$. We show that the width of every variable σ is less than $k(m+2) + q$. If σ is a query variable, it has fewer than $k+q$ predecessors in \hat{D} , so its width is less than $k+q < k(m+2) + q$. Now let σ be an eliminated variable beginning with A , and suppose there is an edge between ρ and σ in \hat{G} . By definition of \hat{G} , ρ must be a query variable, or it must begin with A , or an attribute that precedes A in d and is connected to A in \tilde{G} . Since there are at most $k+q$ query variables, at most $k-1$ other variables beginning with the same attribute as σ , at most m attributes preceding A in \tilde{G} and at most k chains beginning with each such attribute, σ can be connected to at most $k(m+2) + q - 1$ of its predecessors in \hat{G} .

Next, we show that \hat{G} is the induced graph of (\hat{G}, \hat{D}) . Suppose not. Then in the course of eliminating the variables in τ in reverse order of D , edges are added to \hat{G} . Let σ be the first such node. Then σ must be connected to two predecessors ρ_1 and ρ_2 such that ρ_1 and ρ_2 are not connected in \hat{G} . We will show that this is impossible. First, if either ρ_1 or ρ_2 is a query variable, they will be connected to each other by the definition of \hat{G} . So they must both be eliminated variables. If ρ_1 and ρ_2 begin with the same attribute, they will be connected in \hat{G} by definition, so they must begin with different attributes. Now, suppose (wlog) that ρ_1 begins with the same attribute A as σ . Then ρ_2 must begin with an attribute B that is connected to A in \tilde{G} , so again ρ_2 is connected to ρ_1 in \hat{G} . We are left with the possibility that σ , ρ_1 and ρ_2 begin with attributes A , B_1 and B_2 respectively, where A is connected to both B_1 and B_2 in \tilde{G} . Furthermore, since ρ_1 and ρ_2 are predecessors of σ in D , and all are eliminated variables, B_1 and B_2 must be predecessors of A in d . But then, in the process of constructing \tilde{G} , B_1 and B_2 must have been joined together when A was processed (if they were not already connected), so B_1 and B_2 are connected in \tilde{G} . Therefore again ρ_1 and ρ_2 must be connected in \hat{G} .

Now consider the set of factors $\mathbf{f} = \{g_A : A \text{ is a value attribute of } C\}$ appearing at the beginning of the variable elimination phase. Two variables σ and ρ will appear in the same factor if either they are both in the interface of a complex attribute, or they both participate in the CPT of some simple attribute. In either case, either σ and ρ begin with the same attribute, or they begin with two attributes A and B that are connected in the moral graph for X . Either way, σ and ρ are connected in \hat{G} . Therefore the graph for \mathbf{f} is a subgraph of \hat{G} . It follows that the induced graph for \mathbf{f} under the ordering D is also a subgraph of the induced graph of (\hat{G}, d) , which is itself \hat{G} . Therefore the tree-width of the induced graph of \mathbf{f} under D is at most the tree-width of \hat{G} , which is less than $k(m+1)$. ■

An alternative approach is to connect attributes of C that begin query variables to each other in $\mathcal{G}[C]$, and still guarantee that the resulting graph has induced width $\leq m$. If we form \hat{G} from this augmented graph, the width of \hat{G} will indeed be less than $(m+1)k$.

Theorem 4.5.12: *Let I be an OOBN instance, of class C , and let σ be the set of query chains for an SVE call on I . Let \mathbf{Q} be the set of attributes of C that begin a chain $\sigma \in \sigma$. Let G be the graph over attributes of C formed by augmenting $\mathcal{G}[C]$ by connecting each pair of attributes in \mathbf{Q} together. Let k be an upper bound on the number of variables in the interface of a complex attribute of C , and the number of query variables in σ . Suppose that there is an ordering d over the attributes of C such that the induced width of (G, d) is m . Let D be an ordering over the chains τ eliminated during the VE phase of the SVE computation that is consistent with d . Then the induced width M for the VE phase of the SVE computation is less than $k(m+1)$.*

Proof: Similar to Theorem 4.5.11. ■

The value of these results lies in the fact that both k and m are values that can be controlled locally by the engineer of a class model. To control k , she only needs to make sure that the classes for all the complex attributes have interfaces of size at most k , and that the interface of the class itself has at most k variables. As for

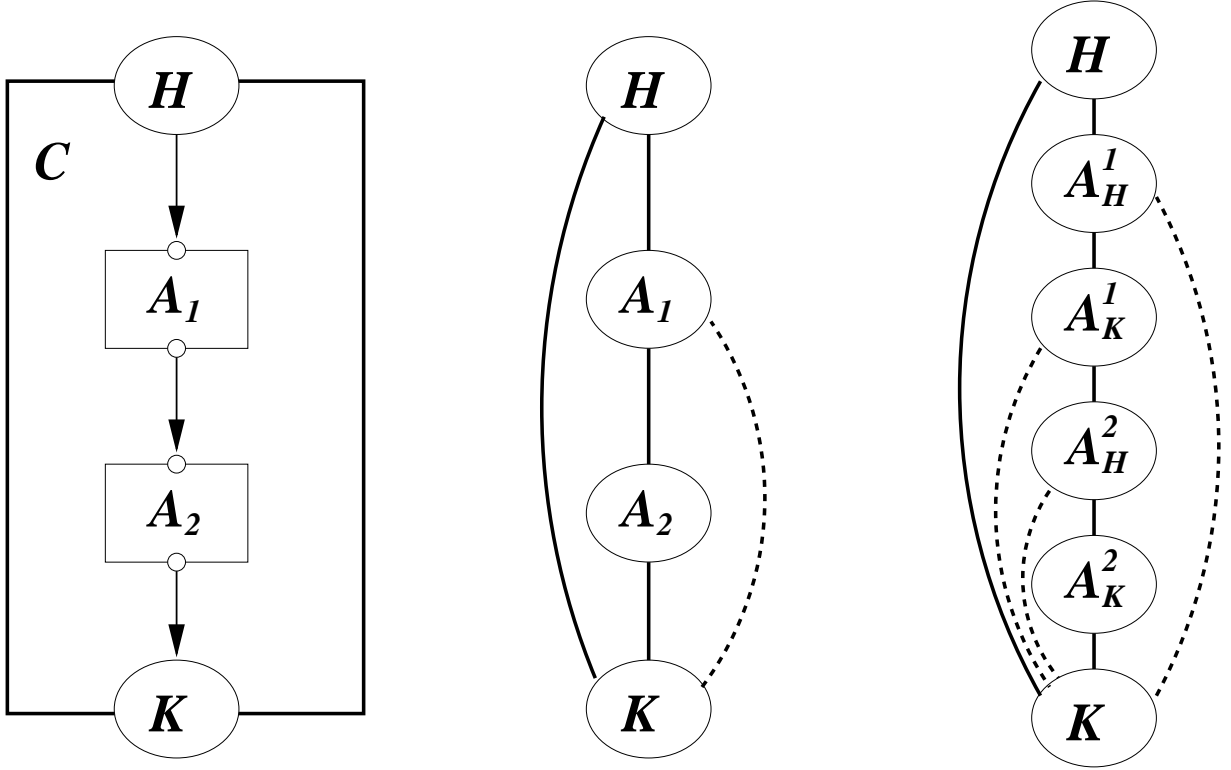


Figure 4.4: Bounds from Theorems 4.5.11 and 4.5.12 are not tight.

controlling m , the number of attributes of a class will typically be very small, so it can be quickly checked whether or not the dependency graph of the class, with all inputs and outputs connected, has tree width $\leq m$. If the bound $k(m + 2) + q$ from Theorem 4.5.11 is too large, Theorem 4.5.12 can be used to try to obtain a better bound for certain specific queries on the class. In particular, it may be useful to obtain a better bound on the cost of computing the iconized distribution for the class, which will also serve as a bound on the cost of computing any query on the class for which there are no additional query variables that are not in the interface.

Note that the bounds from Theorems 4.5.11 and 4.5.12 are not tight. The graph \hat{G} may contain far more edges than necessary. In particular, following the methods of both theorems, we connect all pairs of chains in \hat{G} that begin with attributes that are connected in \tilde{G} . Many of these pairs may actually not be connected in the induced graph for the **SVE** algorithm. Furthermore, following the method of

Theorem 4.5.11, the query variables are connected to all other variables to ensure that \hat{G} is triangulated. Again, many of these connections may not actually be necessary.

Example 4.5.13: The situation is illustrated by the example in Figure 4.4. There is a single class, with simple input H , simple output K , and two encapsulated complex attributes A_1 and A_2 , as shown in the left of the figure. Each of the A_i has an input A_H^i and an output A_K^i . Consider a query on C , in which the set of query variables σ is its interface $\{H, K\}$. The set of eliminated variables ρ is $\{A_H^1, A_K^1, A_H^2, A_K^2\}$. The dependency graph of C has tree-width 1, while k is 2 and q is 0. Following the method of Theorem 4.5.11, we replace each complex attribute A_i with a clique over A_H^i and A_K^i . Since A_1 and A_2 are connected in $\mathcal{G}[C]$, we connected each of the newly introduced variables to each other. In addition, we connect the query variables H and K to all other variables. As a result \hat{G} is a clique over the six variables in σ and ρ . Its tree width is 5, which is indeed less than $(m+2)k+q$ as guaranteed by the theorem, but barely.

Following the method of Theorem 4.5.12, we begin by connecting H and K in $\mathcal{G}[C]$, as shown in the center of Figure 4.4. The dashed edge is added in order to triangulate this graph. The width of the triangulated graph is 2, so m is now 2. \hat{G} is then produced by replacing each A_i with a clique over A_H^i and A_K^i as before. Again, the newly introduced variables are connected to each other. In addition K is connected to all newly introduced variables, while H is connected to A_H^1 and A_K^1 . The graph \hat{G} (not shown) has tree width 4, which is slightly better than before.

However, the true situation is in fact much better. The figure on the right shows the actual induced graph for the **VE** computation in this case. The solid edges show the graph of the set of factors \mathbf{f} , while the dashed edges are edges added in order to triangulate this graph. The width of the triangulated graph is only 2. ■

Theorem 4.5.14: *If the maximum number of variables in the interface of any object in an OOBN \mathcal{O} is k , and the tree width of each object is at most m , then **SVE** can be used to solve a query on q basic variables of \mathcal{O} in time and space $O(Nkb^{k(m+2)}b^q)$, where N is the total number of attributes of all instances in \mathcal{O} .*

Proof: Note that the number of variables in $\sigma_I \cup \tau_I$ is at most $N_I k$, where N_I is the number of attributes of instance I . This is because there are at most k variables in that set beginning with any attribute of I . Also, the total number of chains v_I mentioned during the **SVE** computation for I is at most N , the total number of attributes of all instances in the OOBN.

Next we show that the number of query variables passed to any instance I not in the interface of I is at most q . If a query variable σ not in the interface of $I.A$ is passed in the recursive **SVE** call on $I.A$, then $A.\sigma$ must have been a query variable in the **SVE** call on I . Furthermore, if $A.\sigma$ is in the interface of I , then σ must consist of output attributes of $I.A$. It follows that if σ is not in the interface of $I.A$, $A.\sigma$ is not in the interface of I . Therefore, the number of query variables passed to $I.A$ not in the interface of $I.A$ is at most the number of query variables passed to I not in the interface of I . Since the top-level query has q variables, the claim follows by induction.

By Theorem 4.5.9 **SVE** can be performed for each instance I in time $O(N_I b^M + v)$, where M is the max of M_0 and the largest $|f_A|$ for a complex attribute A of I . $|f_A|$ is clearly less than $k + q < (m + 2)k + q$, while by Theorem 4.5.11, $M_0 < (m + 2)k + q$. Therefore $M < (m + 2)k + q$, and the total cost of inference is is $O(\sum_{I_T.\sigma} (N_\sigma k b^{(m+2)k} b^q + v_I)) = O(N k b^{(m+2)k} b^q)$. ■

Corollary 4.5.15: *If the dependency graph for each object in \mathcal{O} is a polytree, and p is a bound on the number of parents of any attribute, **SVE** can be used to solve a query on \mathcal{O} in space and time $O(N k b^{(p+2)k} b^q)$.*

Proof: Theorems 3.7.10 and 4.5.14. ■

An important case is a query involving no evidence variables. In this case, we can use the iconized distributions for each of the classes, so that only one recursive call needs to be made for each class, rather than one for every instance.

Corollary 4.5.16: *For a query Q on q simple attributes of the top-level instance of \mathcal{O} , and no evidence, **SVE** can be used to solve Q in space and time $O(N k b^{(m+2)k} b^q)$, where N is now the number of attributes appearing in all the classes of \mathcal{O} .*

Proof: Since there are no query variables on attributes of embedded objects, and no evidence, the call to **SVE** will be the same for all embedded instances of the same class. Therefore a separate **SVE** call will not be made more than once for any class in \mathcal{O} . The result follows from Theorem 4.5.14. ■

4.5.5 Discussion

A project manager can stipulate target values for m and k for all the classes used in the project. Values of 2 for m and 4 for k would not be unreasonable, for example. Given such values, the $b^{(m+2)k}$ term can be treated as a (rather large) constant factor. The cost of inference would then grow linearly with the size of the OOBN, and exponentially with the number of query variables. The former is very good news, while the latter is completely unavoidable, since we have to return a probability distribution over all the query variables anyway.

Since the interface variables tend to dominate the inference cost, a good design guideline is to try to keep the number of values of an interface variable to a minimum. Fortunately, this seems a reasonable thing to do from a modeling point of view, since interface variables are used to summarize detailed information about an object. For example, a **Hard-Drive** object may have a **Remaining-Capacity** attribute with many different values. In a good design, this attribute will be encapsulated within **Hard-Drive**. A binary output attribute **Full** can be used to summarize the capacity of the **Hard-Drive** inasmuch as it affects the rest of the system. If we want to know whether or not a file of a certain size can be stored, we have to check the size of the file against the **Remaining-Capacity** of the drive. For such a situation, **Hard-Drive.Remaining-Capacity** can be provided as a query variable.

It is important to note that the inference cost obtained for the **SVE** algorithm on some OOBN query can always be obtained by the standard **VE** algorithm on the flat BN equivalent, for *some* elimination order of the variables. After all, even for the **SVE** algorithm, all the non-query basic variables do have to be eliminated at some point. The structured inference procedure of **SVE** does impose some structure on the elimination order, namely, that all the variables inside one object are eliminated

before any of the variables inside another object. Nevertheless, any elimination order used by the structured algorithm can be emulated in the flat BN. It is quite possible, in fact, that there exist elimination orders for the flat BN that are better than any structured elimination order.

What, then, have we gained from the **SVE** algorithm? For one thing, we now have a way of designing large probability models with known, controlled inference costs, and a way of designing class probability models with compositional performance guarantees. Once we know that **SVE** can be performed in an OOBN in a certain amount of time, we know that **VE** could be performed on the flat BN equivalent in a similar amount of time, but without the **SVE** analysis, we would have no way of knowing that. In addition, the **SVE** analysis provides the model designer with three usable design criteria for keeping the cost of inference down: keep the interfaces small, keep the local graph for each class simple, and keep the domains of the interface variables small. In building a large flat BN from scratch, a model designer would have a hard time focusing her design efforts and understanding what compromises are important to keep the inference costs under control.

Also, the **SVE** algorithm not only provides an analysis that shows that the inference costs can be kept down, but also a means of using that analysis, by exploiting the fact that local interfaces are small and local dependency models are simple. A structured elimination order will exploit these facts; an unstructured order may not. As we discussed in the last chapter, finding a good elimination order for a BN is an NP-hard problem. While fairly efficient algorithms exist for finding optimal or close to optimal elimination orders (e.g. [90], [8]), they are still quite expensive. Most practical implementations of BNs have not found it worth their while to employ them, and instead use a greedy heuristic algorithm to determine the elimination order. Our experimental results, discussed in Chapter 8, show that using a structured elimination order can result in orders of magnitude savings compared to inference in the flat BN using the standard greedy minimum discrepancy heuristic to compute elimination order. In other words, the structure of an OOBN provides a strong hint as to the existence of a good elimination order, and **SVE** provides a way to exploit that hint.

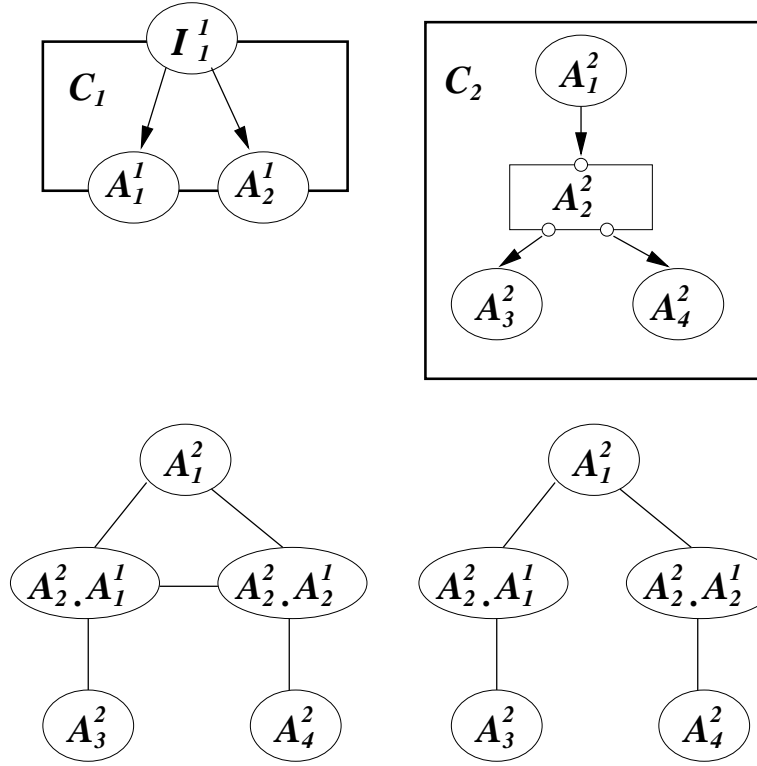


Figure 4.5: Decomposable interfaces.

Finally, **SVE** provides a way to exploit the reuse of computation between different instances of the same class, which cannot be done by **VE** in the flat BN. Our experimental results show that exploiting reuse of computation can yield a further order of magnitude speedup or more.

We close this section by discussing an enhancement to the **SVE** algorithm. After the variable elimination phase, the algorithm multiplies together all the remaining factors to produce a single term over all the query variables. This may not, in fact, be necessary. Consider the following example:

Example 4.5.17: OOBN \mathcal{O} has two classes, C_1 and C_2 , shown in Figure 4.5. C_2 is the top-level class. All attributes of C_1 are simple boolean attributes. It has a single input I_1^1 and two outputs A_1^1 and A_2^1 . Each of the outputs has I_1^1 as its only parent.

C_2 has four attributes: A_1^2 , A_3^2 and A_4^2 which are simple Boolean, and A_2^2 which is complex of class C_1 . The dependency model is as follows:

- A_1^2 is a root;
- the parent of A_2^2 is A_1^2 , and $\Theta[A_2^2.I_1^1]$ is A_1^2 ;
- A_3^2 has $A_2^2.A_1^1$ as its only parent;
- A_4^2 has $A_2^2.A_2^1$ as its only parent.

Consider what happens when we run **SVE** on C_2 . The first thing is a recursive call on C_1 . Since C_1 has no encapsulated attributes, $CPF_{A_1^1}$ and $CPF_{A_2^1}$ are multiplied together to produce a factor over the interface of C_1 . These are then introduced as a factor over $A_1^2, A_2^2.A_1^1$ and $A^2.A_2^1$ in the computation for C_2 . There is thus an edge between $A_2^2.A_1^1$ and $A^2.A_2^1$ in the induced graph (bottom left of the figure) for the C_2 computation, and the width of this graph is 2.

Inference could be performed more cheaply in this example, if, instead of multiplying together $CPF_{A_1^1}$ and $CPF_{A_2^1}$ and returning a factor over the interface, the **SVE** call for C_1 would simply return the pair of factors. These would then be introduced into the C_2 computation as two separate factors, over A_1^2 and $A_2^2.A_1^1$, and A_1^2 and $A_2^2.A_2^1$, respectively. There would then be no edge between $A_2^2.A_1^1$ and $A_2^2.A_2^1$ in the induced graph for the C_2 computation (bottom right of the figure). It is, in fact, a polytree, and its width is one. ■

From the example we see that multiplying together the leftover factors at the end of the **SVE** computation to produce a factor over the interface is actually unnecessary, and may be costly. Instead, we can just return a set of factors over the interface variables. The leftover factors only need to be multiplied together at the very end of the computation, after returning from the top-level **SVE** call, to compute the solution to a query. A similar idea was used in the *multiply-sectioned Bayesian networks (MSBNs)* of Xiang, Poole and Beddoes [100]. The improved **SVE** algorithm is as follows (the lines that have changed are indicated with a *):¹⁴

Algorithm StructuredVariableElimination(I, σ, ρ, e)

¹⁴We show the uncached instance-level version. Obviously the cached version of the algorithm can be modified in the same way.

Let C be the type of I .
 For each simple value attribute A of C do
 If A is a chain $\rho \in \rho$
 Let a be the value assigned to A in e
 $g_A = \{CPF_A[A = a]\}$.
 Else
 $g_A = \{CPF_A\}$.
 For each complex value attribute A of C do
 Let C' be the class of $I.A$.
 Let $\sigma' = \text{Imports}(C') \cup \text{Exports}(C') \cup \{\sigma' : A.\sigma' \in \sigma\}$.
 Let $\rho' = \{\rho' : A.\rho' \in \rho\}$.
 Let e' be the value in $Val[\rho']$,
 such that for each $\rho' \in \rho'$,
 the value assigned to ρ' in e'
 is the same as the value assigned to $A.\rho'$ in e
 * $f_A = \text{StructuredVariableElimination}(I.A, \sigma', \rho', e')$.
 * $g_A = \{\text{Rename}(f_A, \psi) : f_A \in f_A\}$ where
 $\psi(\tau) = \theta_C(A.\tau)$.

 * $f = \cup_A g_A$.
 Let $\tau = \{\tau : \tau \text{ is mentioned by some } f \in f\} - \sigma$.
 For each $\tau \in \tau$ do
 Let g be $\{g \in f : g \text{ mentions } \tau\}$.
 $h_\tau = \prod g$.
 $k_\tau = \sum_\tau h_\tau$.
 $f = f - g \cup \{k\}$.
 * // We no longer need to compute $f = \prod f$.
 * Return f .

4.6 Working with OOBNs

Unlike a BN, which provides a single, standalone model of a domain, an OOBN provides a set of class probability models. Such models can be used as a library, that can be applied to many different specific situations. For example, one may provide a library of many different PC components that can be used to diagnose a wide variety of PC configurations.

In order to use a library of classes in a specific situation, the modeler needs to construct a particular top-level object to describe the specific situation. We call such an object the *scenario* object. There are two basic methods for constructing a scenario object. One is to create it from scratch, putting together a bunch of attributes representing the different objects in the scenario, and connecting them to each other. For example, one might create a scenario to describe a particular PC configuration by putting together the components, e.g. a **Motherboard** attribute of type 200-MHz-Intel-Pentium-II, a **Monitor** attribute of type 15in-SVGA, two **Hard-Drive** attributes, and so on. Each particular PC would have a slightly different set of components, and their types would vary from PC to PC, so the scenario object would be created for each PC.

An alternative approach is to take an existing object model, and modifying it to contain the specific features required by the scenario. The **Computer** object from Figure 4.1 could have been developed using this approach. Example 4.2.5 could It could have been constructed by taking a generic model of a PC with its components, and then adding event objects to it for the particular events that happened in the scenario — in this case, a print event, a read event, and a write event. One could also specify particular subclasses for the different components of the particular computer being examined. However, this does not have to be done all at once, but can be part of a gradual process.

4.6.1 Defining Class and Subclass Models

An important feature of OOBNs that helps make them flexible and capable of modelling many different situations is the ability to define a hierarchy of classes with

similar interfaces but different probability models. Although our relational language contains a class hierarchy, we have not really taken advantage of that feature up to this point. We simply assumed that the model for each class in the hierarchy was defined somehow, with no relationship between the model for a subclass and the model for its superclass, except that they share some of the same attributes. Now, however, we describe how class models may actually be defined, and how a new class can be defined from existing classes.

A class may be defined from scratch, by completely specifying the object structure and the dependency and probability models. The process of creating a class model proceeds in four stages. The first stage is to specify the object structure: the set of attributes, their partition into input, output and encapsulated attributes. For a simple attribute A , $Val[A]$ is specified by explicitly enumerating the possible values; for a complex attribute A , a class name $Class[A]$ is provided to specify its range type. The second stage is to draw a DAG over the attributes, with the inputs as roots. The third stage is to choose the parents of simple attributes and bind the inputs of complex attributes, making sure that all the parents and bindings are visible. The final stage is to assign a CPF to each of the simple attributes. The probability model for complex attributes does not have to be provided, since it is drawn from the associated class.

Alternatively, a subclass may be defined from an existing class (its *base class*) using the standard mechanism of inheritance. As a default, the subclass will inherit all aspects of the model of its base class. The definition of the subclass may override any of the features of the base class, subject to the condition that it should always be possible to use the subclass when the base class is expected. To formalize this condition, we define the notion of type-compatibility between classes.¹⁵

Definition 4.6.1: A class C is *type-compatible* with class C' , if

- for each output attribute K of C' , C has output attribute K , and $Class[K]$ in C is type-compatible with $Class[K]$ in C' .

¹⁵The purpose of this definition is not to say what it means for one class to be a subclass of another in a typed relational language (we did that in Chapter 2); rather, it is meant to help show how a set of class and subclass specifications can work together coherently to form a set of classes in a typed relational language, with a valid information-passing structure.

- for each input attribute H of C , C' has input H , and $Class[H]$ in C' is type-compatible with $Class[H]$ in C . ■

The condition on outputs ensures that if anything uses a value of C' , that value will be available in C . The condition on inputs ensures that if an instance I of C is used where an instance of C' is expected, all the inputs required by I should be available to it.

The definition of a subclass may modify the base class following the same four stages as in the definition of a class from scratch. In the first stage, the object structure is modified. The following changes are allowed:

- Adding an output or encapsulated attribute.
- Removing an encapsulated attribute or input.
- Replacing the range type of a complex output attribute with one that is type-compatible with it.
- If the range type of a complex input attribute is C , replacing the range type with a class C' such that C is type-compatible with C' .
- Changing the type of an encapsulated attribute.

The general rule is that inputs can be removed or made simpler, outputs can be added or made richer, while anything can be done with an encapsulated attribute.

The second stage in defining a subclass is to update the dependency graph. Any new attributes must be added to the dependency graph, and any removed attributes must of course be removed. The graph may be further modified arbitrarily, as long as it remains acyclic, and inputs are always roots.

The third stage is to supply the local dependency models of the new attributes, and to update the models of existing attributes. If A is a new simple attribute, or one whose predecessors in the dependency graph have changed, a new set of parents \mathbf{v} of A must be supplied. The set of parents may optionally be changed for other simple attributes. Any new parent of A must of course be visible to it.

If A is a new complex attribute, a visible attribute chain of the appropriate type must be supplied as the binding for each of its inputs. A new binding must also be supplied for any input of an existing attribute whose previous binding is no longer visible to it. The bindings of other inputs may also optionally be changed.

The final stage in specifying the model of a subclass is to provide a local conditional probability function for each new simple attribute, and for any simple attribute whose set of parents has changed. The CPF may also optionally be changed for other simple attributes.

Example 4.6.2: Let us try to develop a class hierarchy of mouse objects. We want the base **Mouse** class to describe the kinds of things that can go wrong with a mouse, without making any assumptions about how the mouse actually works. It has complex attribute **Has-Cable** of class **Cable**, a **Clicker** attribute of class **Clicking-Mechanism**, and a **Tracker** attribute of class **Tracking-Mechanism**. It has simple attributes for various possible faults. It also has an input, **OS-Status**, because mouse faults can often be due to problems in the OS.

We now develop a hierarchy for the clicking and tracking mechanisms. The generic **Clicking-Mechanism** class has just a single output attribute **Stuck**. The **Clicking-Mechanism** class has three subclasses: **One-Button**, **Two-Buttons**, and **Three-Buttons**. The **Three-Buttons** class changes the model of **Clicking-Mechanism**, by adding three new attributes for each of the buttons, and making **Stuck** depend on them. The **Three-Buttons** class itself has the subclass **Three-Button-Emulator**, which describes the mechanism where the third button is emulated by clicking the two physical buttons simultaneously. The model for **Three-Button-Emulator** has the same attributes as **Three-Buttons**, but **Button-3** is now made to depend on **Button-1** and **Button-2**.

The family of tracking mechanisms has more variety. Again, the generic **Tracking-Mechanism** class consists of just the single output **Status**, whose value ranges over *Responsive*, *Sluggish*, *Stuck* and *No-Response*. (*Stuck* differs from *No-Response* in that it indicates a situation where the response is stuck at a particular value.) One subclass of **Tracking-Mechanism** is the standard **Ball-And-Mousepad**. It has attributes representing possible causes of problems, such as **Fuzzy-Ball**, **Rough-Pad** and **Gunky-Wheels**, all of which influence the status. Another subclass of **Tracking-Mechanism** is the **Trackpoint**

(the little ball found on some laptop PCs). The `Trackpoint` class may change the `Tracking-Mechanism` model only by changing the CPF for `Status` to give higher probability to *Stuck*.¹⁶

Now that we have subclasses for particular types of clicking and tracking mechanisms, we can create subclasses for different mouse devices simply by specifying which particular subclass to use for the `Clicker` and `Tracker` attributes. For example, we can create the `Standard-PC-Mouse` subclass of `Mouse`, and change the `Mouse` model by specifying only that the class of `Clicker` is `Two-Buttons`, and the class of `Tracker` is `Ball-and-Mousepad`. This example illustrates the fact that class hierarchies can be developed in parallel. The major design decision is which families of classes to use. ■

4.6.2 Abstraction and Refinement

OOBNS provide a natural context for working with models iteratively, beginning with an abstract description of a situation, and gradually refining it as necessary. The user can begin with a high-level, abstract scenario object, that can be constructed very quickly, or may be part of a library. The classes of complex attributes may be abstract, *iconized* versions of the types of object they represent. They may have no internal attributes at all, but only a simple conditional probability distribution over the outputs given its inputs. The word “iconized” is indicative of the fact that only the interfaces of the objects appear in the model, but that they are available for expansion at any time. The iconized distribution of an iconized class, as defined in the previous section, is in fact the same as the entire distribution defined by the iconized class model.

If the user decides that the details of a particular component are important, she can replace the iconized class with a particular concrete subclass of it. For example, the iconized `Motherboard` class may be replaced with the `Pentium-Motherboard` subclass of `Motherboard`, which has specific attributes for the `CPU`, `RAM` and so on. The new subclass may themselves be somewhat abstract, and some of its components may itself be iconized. The process of focusing on a particular part of the model and

¹⁶Based on a sample of size 1, this is a recurring problem for trackpoint mechanisms.

refining it is an iterative one.

The advantages of this iterative focusing and refinement process are that both the user's attention and the computational energy are spent where needed. The user only needs to state specific information about the parts of the models she considers important. As for computation, nothing needs to be done for the iconized attributes, since their class model is exactly the conditional probability distribution that needs to be returned by **SVE**. For each refinement step, computation needs to be performed only for the branch of the part-of hierarchy that gets changed, i.e., the object that is refined, and each of its containing objects, up to the top-level object. Since the models of objects in other parts of the hierarchy do not change, computation from previous iterations can be reused. The caching performed by the **SVE** algorithm will ensure that this happens.

4.7 Conclusion

We have presented Object-Oriented Bayesian Networks, a language for building structured probability models for complex, hierarchical domains. OOBNs provide modular, composable models of objects. They provide reusable class probability models, and an inheritance mechanism for creating subclasses. They have coherent probabilistic semantics, both in terms of a natural generative model, and in terms of the hierarchical structure of the system.

We also presented the **SVE** algorithm, which provides a way to exploit the hierarchy and redundancy for probabilistic reasoning in OOBNs. In particular, they allow the designer of a class model to state compositional performance guarantees on the cost of using the class in an OOBN. With their support for reusable classes, inheritance, and reuse of computation, OOBNs provide a natural environment for iterative refinement and focusing of a model as more information is gained about a particular situation.

In the next three chapters, we will discuss extensions of the OOBN framework that provide an even more powerful and flexible representation language. In Chapter 8, we will present an implementation of the SPOOK system for building OOBNs and

working with them, together with experimental results on the performance gains of the **SVE** algorithm, and some example applications.

Chapter 5

Relational Probability Models

5.1 Introduction

In the previous chapter we presented Object-Oriented Bayesian Networks, which allow us to capture much of the structure of a domain in a probability model. In particular, they allow us to express the hierarchical structure of a domain, and the fact that many of the same types of objects reappear over and over again in a model. We showed that OOBNs provide a convenient, modular language for constructing complex probability models. In addition, we showed that both the hierarchical structure and the reuse of class models can be exploited to support efficient inference.

Unfortunately, OOBNs are still not quite adequate for modeling some of the complex domains we are interested in. The problem is that the hierarchical structure they employ is too rigid. All of the objects in the model are forced into the part-of hierarchy, whether it is natural or not. No relationships other than the whole-part relationship can be expressed. A strictly hierarchical structure may be adequate for describing certain kinds of physical systems, such as an airplane or a computer system. However, once we move to more general domains, such as modeling a university or a battlespace situation, the hierarchy is inappropriate. What is the hierarchical relationship, for example, between a student and a course she is taking? Even when modeling a computer system for troubleshooting purposes, we may want to describe different events that happen in a particular situation — events such as a file failing

to load correctly, or an application crashing. The relationship between an event and the components involved in it is not hierarchical. In addition, if we want to model a situation with multiple computers connected over a network, we want to describe the relationship between interconnected computers, which cannot be modeled adequately in a hierarchical manner.

This last example illustrates a related limitation of OOBNs, namely, that they do not allow multiple interconnections between the objects in a system. We want to be able to express the fact, for example, that two different computers in a network happen to use the same printer. We need a language that allows us to talk not only about class models, but also to describe the ways that instances are connected to each other.

In this chapter we address these issues with *relational probability models (RPMs)*. In RPMs, probability models are still associated with classes of objects, but now the instances can be connected to each other by any relationship, not just the part-whole relationship. Objects are no longer organized hierarchically. Given two interrelated objects, neither one is considered to contain the other in an absolute sense. However, from a relative point of view, each object can be said to contain the other as part of its model. In this sense the language is *multi-centered*: any object can be viewed as the “center” of a model, with all other objects spreading out from it.

In addition to allowing non-hierarchical relationships between objects, RPMs allow the class probability models to be augmented with a description of the connections between instances in the system. Logically, connections between instances are relations between domain entities. The relations can be represented using a standard relational database system or a knowledge-based system. Thus, RPMs integrate probabilistic representations with more traditional logic-based representations.

This approach of decomposing the representation language into the probabilistic class models and statements about instances is very flexible, and works well in modeling an environment in which the system configuration changes frequently. The class probability models will remain fixed as the system configuration changes; only the relational model will change. For example, in a computer network model, new

machines will often be added to the network, and the interconnections between machines will often change. This type of change can be handled very easily by an RPM, without the need to specify new object models.

Another restriction of OOBNs that is relaxed in RPMs is the requirement that classes be non-recursive. So, for example, we may now have a **Woman** class with a **Mother** attribute, and the class of **Woman.Mother** is also **Woman**. Even though we allow recursive class definitions, we do not allow the dependency models themselves to be recursive — that will have to wait until Chapter 7. So, for example, if the class **Woman** has a **Happy** attribute, we would not allow **Happy** to depend on **Mother.Happy**. This compromise allows RPMs to integrate with many existing knowledge bases that use recursively defined classes, while still maintaining the relative simplicity of non-recursive probability models.

The fact that objects are no longer organized hierarchically does not come without a cost. We can no longer use input and output attributes to describe the information that flows into and out of an object, since those imply the existence of some higher level calling object that passes information into the object. Instead, an object receives its information from many related objects, none of which are hierarchically superior to it. The way an object depends on other objects is now completely specified within the dependent object. As a result, an object no longer knows all the ways that other objects depend on it, and we cannot separate the internals of an object from the outside world by a simple interface.

One might think to limit the ways other objects can depend on an object by restricting the set of values exported by the object, in the same way that we differentiated between output and encapsulated attributes in OOBNs. However, the presence of recursive class definitions makes this approach fail. For example, if we want to allow a woman's mother to influence objects that are related to the woman, **Mother** must be an output attribute of **Woman**. But then there is no way to stop other objects depending on the great-great-grandmother of a woman. An interface consisting of all information that an instance of **Woman** potentially passes to the outside world would have to be infinite. In short, the lack of a strict hierarchy and the presence of recursive class definitions make it more difficult to exploit encapsulation

of entire portions of a model within an object. Nevertheless, as we shall see, we can still design our inference algorithm to exploit encapsulation wherever possible.

The remainder of this chapter is organized as follows. In the next section we describe the basic relational structure of our models, and in Section 5.3, we describe how the probability model is specified. We also discuss how to make sure the probability model is acyclic and non-recursive. In Section 5.4 we present the semantics of RPMs, giving both intuitive semantics based on a generative process and formal measure-theoretic semantics. In Section 5.5 we modify the **Structured Variable Elimination** algorithm from the previous chapter to account for the complications introduced by RPMs. We conclude in Section 5.6 with a discussion of various aspects of RPMs.

5.2 Basic Language Definition

In the previous chapter, we had a very restricted relational language appropriate for describing hierarchical systems. In order to describe more general relational models, we now relax all the restrictions on the typed relational language, except for the fact that all attributes are single-valued, i.e., that the set \mathbf{R} of relations (multi-valued attributes) is empty. We will call a typed relational language with no multi-valued relations a *single-valued relational language*.

Given a single-valued relational language, we will allow a knowledge base to contain two types of statements. The first type of statement relates named instances to each other via complex attributes. If the KB contains the statement $I.A = J$, then we would expect the value of $I.A$ in any possible world to be equal to the value of J . The second type of statement relates different complex attributes to each other, by declaring one to be the inverse of the other. If the KB contains the statement that B is an inverse of A , then we would expect that in any possible world, if $c.A = d$, then $d.B = c$ for any entities c and d . Formally,

Definition 5.2.1: A *single-valued relational knowledge base* consists of the following:

- A *single-valued relational language*, i.e., a typed relational language $\mathcal{L} = \langle \mathbf{C}, \sqsubseteq, \mathbf{A}, \mathbf{f}, \mathbf{R}, \mathbf{I} \rangle$, where $\mathbf{R} = \emptyset$.

- A set of *instance statements* on \mathcal{L} , of the form $I.A = J$, where $I, J \in \mathbf{I}$, $A \in \mathbf{f}$, the type of I is a subtype of the domain type of A , and the type of J is a subtype of the range type of A .
- A set of *inverse statements* on \mathcal{L} , of the form $A \text{ inverse-of } B$, where the domain type of A is a supertype of the range type of B , and the domain type of B is a supertype of the range type of A . ■

Instance and inverse statements can together imply other statements about attributes of named instances. For example, if a KB contains the statements $I.A = J$, and $B \text{ inverse-of } A$, then it is implied that the value of $J.B$ is I . To make our life easier, we shall always assume that all statements about attributes of named instances that are implied by other statements will be contained explicitly in the KB. If that is not the case, we can always generate such statements in a simple preprocessing phase. We also make the assumption that there is at most one instance statement for any $I.A$.

Example 5.2.2: As a simple illustrative example for this chapter, we will consider a KB \mathcal{K} with a **Person** class, with subclasses **Man** and **Woman**. The **Person** class has complex attributes **Mother** and **Father**, with range type **Woman** and **Man** respectively. The **Person** class also has simple attributes **Healthy**, **Wealthy** and **Happy**, all of which are boolean. The **Woman** and **Man** classes have subclasses **Married-Woman** and **Married-Man** respectively; each has an additional attribute **Wife-Of** or **Husband-Of** with the appropriate range type. \mathcal{K} contains the inverse statements **Wife-Of** *inverse-of* **Husband-Of** and **Husband-Of** *inverse-of* **Wife-Of**.

\mathcal{K} also contains a set of named instances and instance statements describing a family. The named instances are *Liz* of class **Married-Woman**, *Phil* of class **Married-Man**,

Chas of class **Married-Man**, and *Bill* of class **Man**. The instance statements are

Bill.Father = *Chas*
Chas.Mother = *Liz*
Chas.Father = *Phil*
Liz.Wife-Of = *Phil*
Phil.Husband-Of = *Liz*

■

As we did with hierarchical relational languages, we will make some restrictions on the possible worlds to make sure that they have the structure we want. Of course we want the possible worlds to satisfy the instance and inverse statements. In addition, we want the structure of a possible world to be fully known, so that there is no uncertainty as to the values of complex attributes. (We will deal with the case where there is uncertainty as to the relational structure of a possible world in the next chapter.) For hierarchical relational languages, we achieved our goal by stipulating that the entities in the world must be organized in a tree structure, and that there are no extraneous entities. Eliminating extraneous entities was achieved by stipulating that a possible world could not have a proper subworld that was also a possible world, and we use the same technique here. Organizing the entities into a tree was achieved through conditions 1 and 2 of Definition 4.2.7, which stipulated that different complex attributes must have disjoint ranges, and that complex attributes must be one-to-one.¹ Here the situation is a little more complex. We cannot state either of these conditions, because they may be violated by instance and inverse statements, which can force different attributes of different entities to have the same values. We use a more complex condition, which seeks to limit the number of *identities* that hold between different attribute chains. Conditions 1 and 2 of Definition 4.2.7 are special

¹To be precise, in the previous chapter we distinguished between input attributes and value attributes, and only required the complex value attributes to satisfy these conditions. The conditions were necessarily violated for complex input attributes, just as they are here for instance and inverse statements. However, the situation was still simpler in the previous chapter, because the specification of which attributes violated the uniqueness assumption was a part of the language definition, whereas here they are part of the knowledge base.

cases of the more complex condition.

Definition 5.2.3: Let \mathcal{L} be a typed relational language, and ω an interpretation for \mathcal{L} . An *identity* in ω is a quadruple $\langle I, \sigma, J, \rho \rangle$, such that $[I.\sigma]^\omega = [J.\rho]^\omega$. ■

Definition 5.2.4: Let \mathcal{K} be a single-valued relational knowledge base with language \mathcal{L} . A *legal world* for \mathcal{K} is an interpretation ω of \mathcal{L} such that:

1. For every instance statement $I.A = J$ in \mathcal{K} , $[I.A]^\omega = [J]^\omega$.
2. For every inverse statement $A \text{ inverse-of } B$ in \mathcal{K} , and every pair of entities $c, d \in \Delta^\omega$, $c = [B]^\omega(d)$ implies $d = [A]^\omega(c)$.
3. There is no interpretation ω' of \mathcal{K} satisfying conditions 1 and 2 such that the set of identities in ω' is a proper subset of the set of identities in ω . A world satisfying this condition is said to satisfy the *unique names assumption*.
4. There is no proper subworld of ω satisfying conditions 1, 2 and 3. ■

Note that the notion of an inverse presented here one-sided. It is possible for a KB to contain a statement $A \text{ inverse-of } B$ but not $B \text{ inverse-of } A$. A legal world can satisfy the former and not the latter. Also note that an attribute can have multiple inverses. While this may not be particularly useful in practice, it does not present technical difficulties to allow it.

We can now show that these definitions force legal worlds to have the properties we want. We will prove analogues of Lemmas 4.2.9 and 4.2.10, and Theorem 4.2.11, showing that there is a set of attribute chains, called *standard chains*, that play the same role as component chains in hierarchical possible worlds.² That is, the different standard chains have different interpretations, but every chain is the same as some standard chain. Therefore, the set of entities in every possible world corresponds to the set of complex standard chains, while the values of the simple standard chains fully characterize the possible worlds.

²We avoid using the name *component chain* here, since the term *component* implies a hierarchical relationship between a container and its contained object, which we do not wish to assume here.

Definition 5.2.5: Let \mathcal{K} be a single-valued relational KB. A standard chain for \mathcal{K} has the form $I.\sigma$, where I is a named instance, and σ is a (possibly empty) attribute chain on the type of I , such that

1. σ does not begin with an attribute A , such that \mathcal{K} contains an instance statement of the form $I.A = B$, and
2. σ does not contain any occurrence of $A.B$, where \mathcal{K} contains an inverse statement B *inverse-of* A . ■

Note that the definition of standard chain here includes the instance I , whereas it did not in the definition of component chain in the previous chapter. The reason is that now there may be more than one named instance in the KB, and we need to indicate to which one the chain is attached.

Lemma 5.2.6: *Let \mathcal{K} be a single-valued relational KB. For every named instance I , and every attribute chain σ on the type of I , there exists a standard chain $J.\rho$ for \mathcal{K} , such that in every legal world ω , $[I.\sigma]^\omega = [J.\rho]^\omega$.*

Proof: Suppose not. There must be a chain σ of minimal length such that $I.\sigma$ violates the statement of theorem. It cannot be a standard chain. If $\sigma = A.\sigma'$, and \mathcal{K} contains the statement $I.A = I'$, then $[I.\sigma]^\omega = [I'.\sigma']^\omega$. But σ' is shorter than σ , so by assumption there is some standard chain $J.\rho$ such that $[I'.\sigma']^\omega = [J.\rho]^\omega$. But then, $[I.\sigma]^\omega = [J.\rho]^\omega$, contrary to our assumption that $I.\sigma$ violates the statement of the theorem. We are left with the possibility that σ has the form $\tau.A.B.\tau'$, where \mathcal{K} contains the statement B *inverse-of* A . But then $[B]^\omega([I.\tau.A]^\omega) = [I.\tau]^\omega$, so $[I.\sigma]^\omega = [I.\tau.\tau']^\omega$. Since $\tau.\tau'$ is shorter than σ , the same reasoning as above shows that this case is also impossible. We therefore conclude that the statement of the theorem is satisfied for every $I.\sigma$. ■

As in the previous chapter, there is a simple way to compute the standard chain to which an attribute chain is equal. Again, we use the notation $\theta(I.\sigma)$ to denote the

standard chain to which $I.\sigma$ is equal in every possible world. $\theta(I.\sigma)$ is computed as follows:³

While σ has the form $\tau.A.B.\tau'$, where B inverse-of A do:

$\sigma = \tau.\tau'$.

While $\sigma = A.\sigma'$, and \mathcal{K} contains a statement $I.A = J$ do:

$I = J$.

$\sigma = \sigma'$.

Return $\sigma.I$.

Lemma 5.2.7: *Let \mathcal{K} be a single-valued relational KB, ω a legal world for \mathcal{K} , and $I.\sigma$ and $J.\rho$ distinct complex standard chains. Then $[I.\sigma]^\omega \neq [J.\rho]^\omega$.*

Proof: We need to show that there are no identities $\langle I, \sigma, J, \rho \rangle$ in ω , where $I.\sigma$ and $J.\rho$ are distinct standard chains. Without loss of generality, we can assume that σ is at least as long as ρ . We will prove the statement by induction on the lengths of σ and ρ . For the base case, if both σ and ρ are empty, and $I.\sigma = J.\rho$, I must be the same as J by Definition 2.2.3. Therefore there are no identities $\langle I, \sigma, J, \rho \rangle$ with $I.\sigma$ and $J.\rho$ distinct and σ and ρ empty.

For the induction step, consider two standard chains $I.\sigma$ and $J.\rho$, and suppose that there are no identities $\langle I'.\sigma', J'.\rho' \rangle$, with $I'.\sigma'$ and $J'.\rho'$ standard, such that either σ' is shorter than σ , or σ' is the same length as σ and ρ' is shorter than ρ . We must show that $[I.\sigma]^\omega \neq [J.\rho]^\omega$.

Suppose not. We will show that ω violates the unique names assumption, i.e., that there is a world ω' satisfying conditions 1 and 2 of Definition 5.2.4 that has strictly fewer identities than ω . Since σ is at least as long as ρ and at least one is non-empty, σ is non-empty, and we can write $\sigma = \sigma'.A$. Let c be $[I.\sigma']^\omega$, and d be $[I.\sigma]^\omega = [J.\rho]^\omega$. We will construct a new possible world ω' that is identical to ω , except that $\Delta^{\omega'}$ contains an element e not in Δ^ω , the type of e is the same as the type of d , $[A]^{\omega'}(c) = e$, and for every attribute B on the type of e , $[B]^{\omega'}(e) = [B]^{\omega'}(d)$. Note

³We need to process inverse statements before instance statements, since processing inverse statements can cause instance statements to apply, but not vice versa. Consider for example $I.A.B.C$, where \mathcal{K} contains the statements B inverse-of A and $I.C = J$.

that for any instance K and chain τ , if $[K.\tau]^{\omega'} \neq e$, $[K.\tau]^{\omega'} = [K.\tau]^\omega$. Furthermore, if $[K.\tau]^{\omega'} = e$, then τ must be $\tau'.A$, $[K.\tau']^\omega = c$, and $[K.\tau]^\omega = d$. Therefore any identity that holds in ω' must hold in ω .

Now, suppose that ω' violates condition 1 of Definition 5.2.4. Then there must be an instance statement $K.B = K'$ that holds in ω but not in ω' . We must then have $[K.B]^{\omega'} \neq [K.B]^\omega$, so $[K.B]^{\omega'} = e$. By the comments above, this implies that $B = A$ and $[K]^\omega = c$. But, by induction hypothesis, the identity $\langle K, \epsilon, I, \sigma' \rangle$ cannot hold if $K.\epsilon$ and $I.\sigma'$ are distinct. So I must be K , and $\sigma' = \epsilon$. So \mathcal{K} contains the instance statement $I.A = K'$, contrary to assumption that $I.\sigma = I.A$ is standard. Therefore ω' must satisfy condition 1.

Suppose now that ω' violates condition 2 of Definition 5.2.4. Then \mathcal{K} must contain an inverse statement B' *inverse-of* B that holds in ω but not in ω' . So there must be some $K.\tau.B.B'$ such that $[K.\tau.B.B']^{\omega'} \neq [K.\tau.B.B']^\omega$, and again we must have $B' = A$, with $[K.\tau]^\omega = d$, $[K.\tau.B]^\omega = c$, $[K.\tau.B.A]^\omega = d$, but $[I.\tau.B.A]^{\omega'} = e$, violating the inverse statement. By induction hypothesis, the identity $\langle I, \sigma', J, \rho.B \rangle$ cannot hold if $J.\rho.B$ is standard, so ρ must be $\rho'.C$, where B *inverse-of* C . But then we must have

$$c = [B]^\omega(d) = [B]^\omega([J.\rho]^\omega) = [B]^\omega([J.\rho'.C]^\omega) = [J.\rho']^\omega,$$

so the identity $\langle I, \sigma', J, \rho' \rangle$ must hold, contrary to induction hypothesis. Therefore ω' must satisfy condition 2.

Finally, the identity $\langle I, \sigma, J, \rho \rangle$, which holds in ω , does not hold in ω' . Therefore ω' is a world satisfying conditions 1 and 2, such that the set of identities in ω' is a proper subset of the set of identities in ω . Therefore ω violates the unique names assumption, contrary to hypothesis that ω is a legal world. ■

Theorem 5.2.8: *Let \mathcal{K} be a single-valued relational KB, ω a legal world for \mathcal{K} , and let Σ denote the set of complex standard chains in \mathcal{K} . There exists a one-to-one correspondence ϕ from Σ to Δ^ω , such that for any standard chain $I.\sigma \in \Sigma$, $\phi(I.\sigma) = [I.\sigma]^\omega$, and the class of $\phi(I.\sigma)$ (Definition 2.2.4) is the range type of σ .*

Proof: The proof is the same as that of Theorem 4.2.11, using the no proper sub-worlds condition, except that now Lemma 5.2.7 is used instead of Lemma 4.2.9 to show that ϕ is one-to-one. ■

We can now proceed as in the previous chapter, to fix the set of domain elements in our possible worlds to be precisely the set of standard chains.

Definition 5.2.9: Let \mathcal{K} be a single-valued relational KB. The *set of possible worlds* for \mathcal{K} , written $\Omega_{\mathcal{K}}$, or simply ω when \mathcal{K} is clear, is the set of legal worlds for \mathcal{K} in which Δ is $\{I.\sigma : I.\sigma \text{ is a standard chain for } \mathcal{K}\}$, and the map ϕ from Theorem 5.2.8 is the identity map. ■

Example 5.2.10: Consider the KB \mathcal{K} of Example 5.2.2. A possible world for \mathcal{K} has the following structure. There are entities $[Liz]^\omega$, $[Phil]^\omega$, $[Chas]^\omega$, and $[Bill]^\omega$, corresponding to the four named instances. The instance statements stipulate that $[Chas.Mother]^\omega = [Liz]^\omega$, $[Chas.Father]^\omega = [Phil]^\omega$, $[Bill]^\omega.Father = [Chas]^\omega$, $[Liz]^\omega.Wife-Of = [Phil]^\omega$, and $[Phil]^\omega.Husband-Of = [Liz]^\omega$. Because the type of $Chas$ is **Married-Man**, there must be some entity c that is equal to $[Chas.Husband-Of]^\omega$. The type of c is **Married-Woman**, and $[Wife-Of]^\omega(c) = Chas$. Also, there is another entity d that is equal to $[Bill.Mother]^\omega$. By the unique names assumption, $c \neq d$ — if we want to force these to be equal, we need to introduce a new named instance and two instance statements. The parents of $[Liz]^\omega$, $[Phil]^\omega$, c and d , so their must be eight other domain entities representing their parents. Similarly, each of these parents has two other entities representing their parents, and so on. By the unique names assumption, all these newly introduced parents are distinct. Also, by the no proper subworlds condition, d and all the other entities introduced as parents must be of the class **Man** or **Woman**, and not married. ■

The possible worlds are fully characterized by the values of the simple attributes of the domain entities, i.e., by the values of the simple standard attribute chains. In the previous chapter we defined basic variables in terms of component chains. We no longer have components in the current framework, but standard chains serve the same purpose as component chains.

Definition 5.2.11: A *basic variable* is a simple standard attribute chain. ■

Example 5.2.12: Continuing our running example, the basic variables for \mathcal{K} are all chains that have one of the following forms, where A is one of the simple attributes Healthy, Wealthy or Happy, and σ is an attribute chain consisting of zero or more occurrences of Mother or Father:

$Liz.\sigma.A$
 $Phil.\sigma.A$
 $Chas.A$
 $Chas.Husband-Of.\sigma.A$
 $Bill.A$
 $Bill.Mother.\sigma.A$

Note that there are no basic variables beginning with $Chas.Mother$, $Chas.Father$, or $Bill.Father$, because of instance statements, or beginning with $Chas.Husband-Of.Wife-Of$, because of inverse statements. ■

Note that there may be infinitely many basic variables, as in this example. Therefore, we can no longer define an equivalent attribute-based model for our language as we did in the previous chapter. As a result, when we come to define the probabilistic semantics of our language, we will not be able to use a flat BN equivalent of the entire KB.

Theorem 5.2.13: Let \mathcal{K} be a single-valued relational KB. Let \mathbf{X} be the set of basic variables of \mathcal{K} , and let \mathbf{x} be an assignment of values to \mathbf{X} such that for each $X \in \mathbf{X}$, the assigned value x is in $Val[X]$. There exists a unique possible world $\omega \in \Omega_{\mathcal{K}}$ such that for each $X \in \mathbf{X}$, $[X]^\omega = x$.

Proof: Same as Theorem 4.2.16. ■

5.3 Probability Model and Acyclicity

5.3.1 Generative Semantics

As we did with hierarchical systems, we create a probability model for more general relational systems by supplying a local probability model for each class of object. The local probability model for a class is specified in the same way as in an OOBN. The dependency model is specified by listing a set of parents for each simple attribute, each parent being a simple attribute chain. The numeric component of the probability model is provided by specifying a CPF for each of the simple attributes.

Definition 5.3.1: Let A be a simple attribute of class C . A *local probability model* for A consists of:

- A set of parents $\mathbf{v} = v_1, \dots, v_m$, where each v_i is a simple attribute chain on C .
- A conditional probability function CPF_A from $Val[\mathbf{v}]$ to $Val[A]$.

A *class probability model* for a class C consists of a local probability model for each of the simple value attributes of C . ■

Example 5.3.2: Continuing our running example, we could define the parents of the Happy attribute of Person to be Mother.Father.Wealthy and Father.Healthy, with an appropriate CPF. For the Married-Man subclass of Person, we may specify that the Happy attribute has the single parent Husband-Of.Happy, while the Happy attribute of the Married-Woman class has the parent Wife-Of.Mother.Happy.

In our example KB, these specifications will induce dependencies between basic variables. For example, *Bill.Happy* depends on *Bill.Mother.Father.Wealthy* and on *Chas.Healthy*, while *Phil.Happy* depends on *Liz.Happy*. The class probability models combine together with the relational structure of the KB to specify how the basic variables in the KB actually influence each other. ■

Notice the flexibility of this approach to creating probability models. The probabilistic components of the representation language are associated with classes of

objects. The probability model associated with a class specifies, for each instance of the class, a conditional probability distribution over properties of the instance given properties of its related objects. It makes no assumptions about the identity of the instance, or the identities of the related instances. The class probability models remain fixed as the relational structure of the domain varies. The system modeller is therefore protected from having to redefine a probability model each time the system configuration changes. He needs only to design a set of class probability models once and for all; changes to the system configuration can be handled by updates to a traditional relational database or knowledge based system.

A set of probability models for all the classes in a single-valued relational KB \mathcal{K} defines a dependency model over all the basic variables of \mathcal{K} .

Definition 5.3.3: Let \mathcal{K} be a single-valued relational KB, and \mathcal{P} consist of a probability model \mathcal{P}_C for each class C of \mathcal{K} . We define the relation \leftarrow over basic variables of \mathcal{K} as follows. For any basic variables X and Y , let $X = I.\sigma.A$, and let C be the range type of σ . Then $X \leftarrow Y$ iff Y is equal to $\theta(I.\sigma.v)$ for some parent v of A in \mathcal{P}_C . If $X \leftarrow Y$, we say that X *depends on* Y . ■

In order for a set of class probability models to be coherent, we require two properties of the dependency relation \leftarrow . The first is obvious: the relation should be acyclic, as it is in BNs and OOBNs. The second requirement is that there should be no infinite chains of dependencies, i.e., no infinite sequence of basic variables $(X)_1^\infty$ such that $X_i \leftarrow X_{i+1}$. The reason is that we will define the probability model in terms of a generative process that can generate the value of any basic variable. In order to generate the value of a variable, the process must first generate the values of its parents. If there is an infinite dependency chain, the process will not terminate. For now, therefore, we disallow infinite dependency chains. We shall consider models with infinite chains in Chapter 7.

Obviously we cannot check that \leftarrow has no cycles or infinite chains by examining the relation directly, because there may be infinitely many variables. Rather, we need to examine the structure of the probability models. One approach to guaranteeing that \leftarrow is acyclic is similar to the one we used in OOBNs. We create a dependency

graph for each class, and make sure that it is acyclic. In OOBNS, the graph took into account the information-passing structure. Here we have no information-passing structure, so we have to account for the flow of information between objects in a different way.

Definition 5.3.4: Let \mathcal{P} be a set of probability models for the classes of a single-valued relational KB. For each complex attribute A , we define the *import set* of A , written $Imp[A]$ to be the set of attributes B of the range type of A , such that there is some class C and some attribute D of C such that a parent of D in \mathcal{P}_C has the form $\sigma.A.B.\tau$. The *export set* of A , written $Exp[A]$, contains all the attributes of C that are in the import set of an inverse of A , i.e., $Exp[A] = \cup_{\{B:B \text{ inverse-of } A\}} Imp[B]$.

The *dependency graph* of C , written $\mathcal{G}[C]$ has an edge from A to B if either:

- B is simple, and some parent of B in \mathcal{P}_C begins with A , or
- B is complex, and $A \in Exp[B]$. ■

The reasoning behind this definition is as follows. Suppose I and J are two objects,⁴ with $I.A = J$. If A does not have an inverse, then the only flow of information between the two objects is from J to I . This is because attributes of I can refer to J , but not vice versa. However, if A has an inverse B , then attributes of J can refer to I , and information can flow back and forth between the two objects. The import set of B lists those attributes of I that are referred to in the probability model for J . These attributes will be contained in the export set of A . By making sure that all attributes in the export set of A precede A in $\mathcal{G}[C]$, while attributes that refer to J follow A in $\mathcal{G}[C]$, we ensure that the interaction between I and J does not produce a cycle.

Example 5.3.5: Suppose we wanted to augment the dependencies specified in Example 5.3.2 by specifying that the **Happy** attribute of **Married-Woman** also depends on **Wife-Of.Happy**. This creates a cyclic dependency model, because the happiness of the husband and wife mutually influence each other. The cycle is detected in

⁴ I and J are general domain entities here, not necessarily named instances.

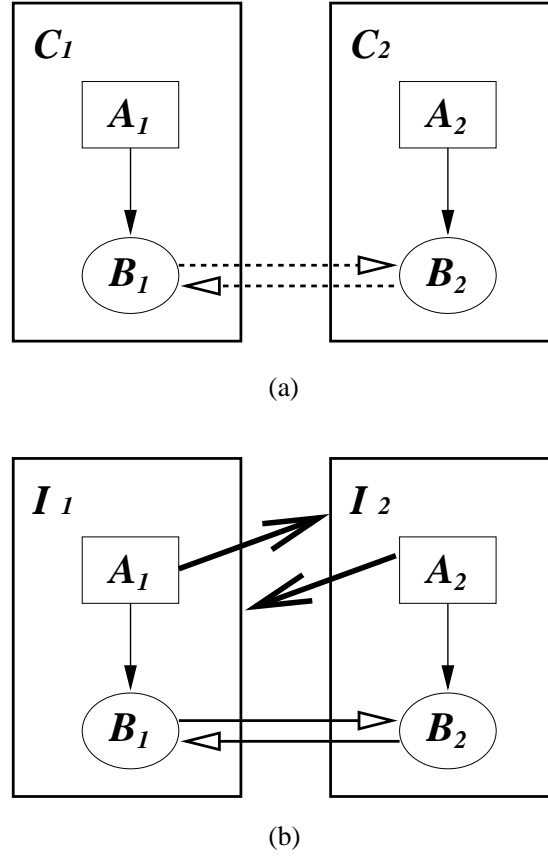


Figure 5.1: (a) Infinite and (b) cyclic dependency models.

$\mathcal{G}[\text{Married-Woman}]$, as follows: because **Happy** depends on **Wife-Of.Happy**, there is an edge from **Wife-Of** to **Happy**. However, because the **Happy** attribute of **Married-Man** depends on **Husband-Of.Happy**, **Happy** is in $\text{Imp}[\text{Husband-Of}]$. Since **Husband-Of** is an inverse of **Wife-Of**, **Happy** is therefore in $\text{Exp}[\text{Wife-Of}]$. Therefore there is an edge from **Happy** to **Wife-Of** in $\mathcal{G}[\text{Married-Woman}]$, and the cycle is detected. ■

In fact, for a KB that contains no instance statements, acyclicity of all the $\mathcal{G}[C]$ is sufficient to guarantee that \leftarrow is acyclic. However, this approach is deficient for several reasons. First, it does not rule out infinite dependency chains. In addition, in the presence of instance statements, it does not even rule out acyclic chains.

Example 5.3.6: Figure 5.1 (a) shows an example of an infinite dependency chain. There are two classes, and each class C_i has a complex attribute A_i and a simple

attribute B_i . The range type of A_1 is C_2 , while the range type of A_2 is C_1 . B_1 has the parent $A_1.B_2$, while B_2 has the parent $A_2.B_1$. Neither of the complex attributes has an inverse, so their export sets are both empty. Each of the class dependency graphs are as shown, with an edge from A_i to B_i . There are, in fact, no cyclic dependency chains in this model. However, if there is a basic variable $X = I.\sigma.B_1$, where the range type of σ is C_1 , X will depend on $I.\sigma.A_1.B_2$, which will in turn depend on $I.\sigma.A_1.A_2.B_1$, and so on.

Figure 5.1 (b) illustrates that in the presence of instance statements, there may be cyclic dependencies even when all the class dependency graphs are acyclic. In this example there are two instances: I_1 of type C_1 , and I_2 of type C_2 , with the models of C_1 and C_2 as before. The KB contains the instance statements $I_1.A_1 = I_2$ and $I_2.A_2 = I_1$. Now $I_1.B_1$ depends on $I_2.B_2$ and vice versa.

Examples of infinite and cyclic dependency chains not detected by the class dependency graphs can also be created in our running example. Adding **Mother.Happy** as a parent of **Happy** obviously creates an infinite dependency chain, but there is no edge going out of **Happy** in $\mathcal{G}[\text{Person}]$ so the graph remains acyclic. An example of a cyclic dependency chain is a little more contrived — if we stipulate that $\text{Bill.Father} = \text{Bill}$, then adding **Father.Happy** as a parent of **Happy** creates a cyclic dependency chain. ■

One can address these issues by introducing a global dependency graph, in which dependencies between attributes of different objects, both classes and instances, are modeled.⁵ This need for a global dependency graph introduces a slight loss of modularity, since we can no longer guarantee based solely on the class dependency graphs that a complete dependency model will be coherent. However, the global dependency graph is a data structure that can be computed by the system, and the user does not need to worry about it explicitly. Cycles can be presented to the user as they are detected; when that happens, the user will be alerted of the inter-object interactions that are causing a problem.

Definition 5.3.7: Let \mathcal{K} be a single-valued relational KB, with a set \mathcal{P} of probability models for the classes of \mathcal{K} . The *global dependency graph* for \mathcal{K} is the graph $\mathcal{G}[\mathcal{K}]$

⁵Even though we are dispensing with the class dependency models for eliminating cycles, they are still a useful concept, and will in fact be used in our inference algorithm.

defined as follows: $\mathcal{G}[\mathcal{K}]$ contains a node $C.A$, for each class C in \mathcal{K} and each attribute A of C , and a node $I.A$, for each named instance I in \mathcal{K} and each attribute A of the type of I . $\mathcal{G}[\mathcal{K}]$ contains the following edges:

1. If a parent v of A in \mathcal{P}_C begins with B , $\mathcal{G}[\mathcal{K}]$ contains
 - (a) an edge from $C.B$ to $C.A$;
 - (b) an edge from $I.B$ to $I.A$ for each instance I of type C .
2. If $A \in \text{Imp}[B]$, and C_1 and C_2 are the range type and domain type of A respectively, $\mathcal{G}[\mathcal{K}]$ contains
 - (a) an edge from $C_1.A$ to $C_2.B$;
 - (b) an edge from $C_1.A$ to $I.B$ for each instance I of type C_2 such that \mathcal{K} does not contain an instance statement assigning a value to $I.B$;
 - (c) an edge from $J.A$ to $I.B$ if I is an instance of type C_2 and \mathcal{K} contains an instance statement $I.B = J$. ■

If the global dependency graph is acyclic, then \leftarrow is acyclic and contains no infinite chains. In the examples of Figure reffig:cyclicity-example, the infinite dependency chain will be detected by the cycle $(C_1.B_1, C_1.A_1, C_2.B_1, C_2.A_2, C_1.B_1)$, while the cyclic instance-level dependency chain will be detected by the cycle $(I.B_1, I.A_1, J.B_1, J.A_2, I.B_1)$.

Theorem 5.3.8: *Let \mathcal{K} be a single-valued relational KB, and \mathcal{P} be a set of probability models for the classes of \mathcal{K} . If $\mathcal{G}[\mathcal{K}]$ is acyclic, then \leftarrow is acyclic and contains no infinite dependency chains.*

Proof: For a standard attribute chain $I.\sigma.A$, we will let $[I.\sigma.A]$ denote a *representative node* for $I.\sigma.A$ in $\mathcal{G}[\mathcal{K}]$ as follows: If σ is empty, $[I.\sigma.A] = I.A$. If σ is non-empty, $[I.\sigma.A] = C.A$, where C is the range type of σ .

Suppose X and Y are basic variables, and $X \leftarrow Y$. We claim that $[Y]$ is an ancestor of $[X]$ in $\mathcal{G}[\mathcal{K}]$.

Let X be $I.\sigma.A$. Y must be equal to $\theta(I.\sigma.v)$ for some parent v of A . Let A_0 be the first attribute of v , and let $Z_0 = I.\sigma.A_0$. By definition of $\mathcal{G}[\mathcal{K}]$, there is an edge

from $[Z_0]$ to $[X]$ in $\mathcal{G}[\mathcal{K}]$: clause 1 (a) applies if σ is non-empty, while clause 1 (b) applies if σ is empty.

Recall that, in the computation of θ , we first eliminate pairs $A.B$ such that B *inverse-of* A , and then process instance statements. Write $\sigma = \sigma'.B_1 \dots .B_n$, and $v = B_n^{-1} \dots .B_1^{-1}.v'$, where B_i^{-1} *inverse-of* B_i , and the first attribute of v' is not an inverse of the last attribute of σ' . We allow the cases where σ' is empty, or $n = 0$ (i.e., $v' = v$). Let $v' = A_1 \dots .A_m$, and $Z_1 = I.\sigma'.A_1$. If $n = 0$, $Z_1 = Z_0$, and $[Z_1]$ is an ancestor of $[X]$ in $\mathcal{G}[\mathcal{K}]$. Otherwise, for $i > 1$, since v contains $B_i^{-1}.B_{i-1}^{-1}$, $B_{i-1}^{-1} \in \text{Imp}[B_i^{-1}]$. By clause 2 of Definition 5.3.7, there is an edge from $[I.\sigma.B_n^{-1} \dots .B_i^{-1}.B_{i-1}^{-1}]$ to $[I.\sigma.B_n^{-1} \dots .B_i^{-1}]$ in $\mathcal{G}[\mathcal{K}]$: clause 2 (a) applies, unless $i = n$ and σ is empty, in which case clause 2 (b) applies. Similarly, since v contains $B_1^{-1}.A_1$, there is an edge from $[Z_1]$ to $[I.\sigma.B_n^{-1} \dots .B_1^{-1}]$. Since $[Z_0] = [I.\sigma.B_n^{-1}]$, $[Z_1]$ is an ancestor of $[Z_0]$ in $\mathcal{G}[\mathcal{K}]$, and therefore also an ancestor of $[X]$.

Now, if σ' is non-empty, no instance statement can apply to $I.\sigma'.v'$, otherwise $X = I.\sigma.v$ would not be a standard chain, since σ' is a prefix of σ . Therefore $Y = I.\sigma'.v' = I.\sigma'.A_1 \dots .A_m$. Also, for $i < m$, since v contains $A_i.A_{i+1}$, there is an edge from $[I.\sigma'.A_1 \dots .A_i.A_{i+1}]$ to $[I.\sigma'.A_1 \dots .A_i]$ in $\mathcal{G}[\mathcal{K}]$, by clause 2 (a) of Definition 5.3.7. Since $Z_1 = I.\sigma'.A_1$, and $[Z_1]$ is an ancestor of $[X]$, it follows that $[Y]$ is an ancestor of $[X]$ as claimed.

If on the other hand σ' is empty, we may need to apply instance statements to $I.v'$ in the process of computing $\theta(I.\sigma.v)$. Write $I_1 = I$, and let I_2, \dots, I_ℓ be named instances such that for $j < \ell$, \mathcal{K} contains the instance statement $I_j.A_j = I_{j+1}$, but \mathcal{K} contains no instance statement on $I_\ell.A_\ell$. Because A_m is simple, we must have $\ell \leq m$. Y is now equal to $I_\ell.A_\ell \dots .A_m$, and $[Y]$ is the same as or an ancestor of $I_\ell.A_\ell$ by the same reasoning as before. Also, for $j < \ell$, since $I_j.A_j = I_{j+1}$, and $A_{j+1} \in \text{Imp}[A_j]$, there is an edge from $I_{j+1}.A_{j+1}$ to $I_j.A_j$ in $\mathcal{G}[\mathcal{K}]$ by clause 2 (c) of Definition 5.3.7. Since $Z_1 = I_1.A_1$, it follows that $[Y]$ is either equal to or an ancestor of $[Z_1]$, and again $[Y]$ is an ancestor of $[X]$ as claimed.

As a result, every dependency chain $X_1 \leftarrow X_2 \leftarrow \dots$ corresponds to a sequence of nodes $[X_1], [X_2], \dots$ of $\mathcal{G}[\mathcal{K}]$, such that $[X_{i+1}]$ is an ancestor of $[X_i]$. Since $\mathcal{G}[\mathcal{K}]$ is finite and acyclic, there can be no cyclic or infinite dependency chains. ■

Definition 5.3.9: A *single-valued relational probability model* is a pair $\langle \mathcal{K}, \mathcal{P} \rangle$, in which \mathcal{P} consists of a probability model \mathcal{P}_C for each C in \mathcal{K} , such that $\mathcal{G}[\mathcal{K}]$ is acyclic. ■

The method we have presented here, that uses the global dependency graph $\mathcal{G}[\mathcal{K}]$ to prevent models with cyclic and infinite dependency chains, is sound by Theorem 5.3.8, but it is by no means complete. If a dependency model is cyclic or has infinite chains, that will be detected in the graph. However, it is quite possible for the dependency graph to be cyclic, but for the dependency relation \leftarrow to be safe.

The reason is that whenever $B \in \text{Imp}[A]$, we make $C'.B$ a parent of $C.A$ in $\mathcal{G}[\mathcal{K}]$, (C and C' being the domain types of A and B respectively). If D is a simple attribute of C that has a parent beginning with A , $C.D$ will be a child of $C.A$ in $\mathcal{G}[\mathcal{K}]$. This implies that $C'.B$ will precede $C.D$ in the dependency graph, even if D does not actually depend on $A.B$, but only on some other attribute of C' imported by A . In particular, the dependency graph prevents patterns of interaction where two objects pass information back and forth several times, in such a way that the total flow of information is acyclic.

If we want to allow these more sophisticated interaction patterns, we need a dependency graph with a finer granularity, that can model the flow of information in more detail. One way to do this is to make a simple attribute D of C depend directly on $C'.B$ if a parent of D begins with $A.B$ (and the range type of A is C'), rather than mediate the dependency through the imports set of A . We omit the details of how the new dependency graph is constructed, and how it prevents cyclic and infinite dependency chains, as they should be fairly clear. Some details can be found in [27].

5.4 Semantics

5.4.1 Generative Semantics

As usual, we first present intuitive semantics for our language in terms of a generative process. This process generates values for all simple attributes of all the objects in the model. It begins with a single top-level process for all the attributes of

named instances. The generation of these attributes needs to be interleaved, because attributes of different instances may mutually depend on each other, as shown in Example 5.3.6. Generating the value of a complex attribute that is not bound via some instance statement requires a recursive call to a process that generates an instance of the appropriate class.

In the top-level process, we use an ordering d over the attributes of the named instances that is consistent with $\mathcal{G}[\mathcal{K}]$. I.e., if $I.A$ precedes $J.B$ in $\mathcal{G}[\mathcal{K}]$, $I.A$ must precede $J.B$ in d . For the process that generates values for an instance of a class C , we use an ordering d_C over attributes of C that is consistent with $\mathcal{G}[\mathcal{K}]$. I.e., if $C.A$ precedes $C.B$ in $\mathcal{G}[\mathcal{K}]$, then A precedes B in d_C .

Before performing a recursive call to generate the value of a complex attribute, the process checks to see whether that attribute already has a value. That could happen due to either instance or inverse statements. We take care of the former by enforcing all instance statements at the very beginning of the process, while we take care of the latter by enforcing inverse statements as soon as they become applicable.

The full generative process is as follows. The top-level **GenerateWorld** call takes a single-valued relational probability model \mathcal{K} as argument, and returns a world $\omega \in \Omega_{\mathcal{K}}$.

Procedure GenerateWorld(\mathcal{K})

```

Let  $\omega$  be an empty world.
Let  $I$  be the named instances of  $\mathcal{K}$ .
For each instance  $I \in I$  do:
  Let  $C$  be the type of  $I$ .
   $\Delta^\omega \leftarrow I : C$ .
  For each attribute  $A$  of  $I$  do:
    If  $\mathcal{K}$  contains a statement  $I.A = J$ 
       $[A]^\omega(I) \leftarrow J$ .
  For each  $I.A$  in order  $d$  do:
    If  $A$  is simple
      GenerateSimple( $I, A$ ).
    Else
```

FindOrGenerateComplex(I, A) .

Procedure **GenerateSimple**(c, A)

Let C be the type of c .

Let \mathbf{v} be the set of parents of A in \mathcal{P}_C .

Choose $v \in \text{Val}[A]$, according to $\text{CPF}_A(v \mid [\mathbf{v}]^\omega(c))$.

$[A]^\omega(c) \leftarrow v$.

Procedure **FindOrGenerateComplex**(c, A)

If $[A]^\omega(c)$ exists

Return.

$\Delta^\omega \leftarrow c.A$.

Let C be the range type of A .

$[A]^\omega(c) \leftarrow c.A : C$.

If \mathcal{K} contains a statement B *inverse-of* A

$[B]^\omega(c.A) = c$.

GenerateEntity($c.A, C$) .

Procedure **GenerateEntity**(c, C)

For each attribute A of C in order d_C do:

If A is simple

GenerateSimple(c, A) .

Else

FindOrGenerateComplex(c, A) .

The **GenerateWorld** process presented here is superficially very similar to the process for OOBNS. However, there is a major difference. Because there may be infinitely many domain entities, the process as described here may never terminate. So we cannot actually envision running the process to generate a complete possible world. Instead, we can envision a *lazy evaluation* of the process, in which we specify a set of attributes of domain entities that we are interested in, and then only generate those aspects of the world that are actually needed to determine the values of the attributes

we are interested in. Because the dependency model contains no infinite dependency chains, only a finite portion of the world needs to be generated to determine the values of a finite number of attributes.

These intuitions suggest a way to formalize the probability model defined by a KB. Rather than constructing an infinite flat BN equivalent for the entire KB, we generate a flat BN relative to a finite set of attributes of domain entities. This restricted BN will be finite, and will define a probability distribution over the values of that particular set of attributes. We can then put all these local distributions together, to get a *probability measure* over all the attributes of all the domain entities.

5.4.2 Probability Measures

We now briefly present the basic concepts of measure theory needed to understand the semantics of our language. The reader who is interested in a more thorough treatment should consult one of the many textbooks on measure theory, such as [34].

Definition 5.4.1: Let Ω be a set of possible worlds. An *algebra* over Ω is a set of subsets of Ω that is closed under finite unions and complements. A σ -*algebra* over Ω is a set of subsets of Ω that is closed under countable unions and complements. ■

It is clear that an algebra is also closed under intersections and differences. It is also clear that \emptyset and Ω belong to any algebra over Ω .

A subset of Ω essentially defines a property of possible worlds, namely, the property of being in the subset. Rather than defining the probability of each possible world directly, we will define the probability that a possible world satisfies some property, i.e., the probability that it lies in some subset of Ω . We characterize the set of properties whose probabilities we can talk about by specifying a σ -algebra \mathcal{E} over Ω . Each set $E \in \mathcal{E}$ is called an event, representing the “event” that $\omega \in E$, and \mathcal{E} is called the *event space*. The requirement that the event space be an algebra makes sure that if we can speak of the probabilities of the events E and F , we can also speak of the probabilities of “not E ”, and of “ E or F ”, which is only reasonable. The further requirement that the event space be a σ -algebra allows us to speak of events

that can only be characterized as the countably infinite union of more basic events. For example, suppose Ω consists of English sentences, and the event E_n is the set of sentences of n words beginning with “Once upon a time...”. If we can specify the probability of E_n for each n , we ought to be able to talk about the probability of the event $\cup_n E_n$, which is the set of sentences of any length beginning with “Once upon a time...”.

Definition 5.4.2: Let \mathcal{E} be a set of subsets of Ω . A real-valued function f on \mathcal{E} is *additive* if, for every disjoint $E, F \in \mathcal{E}$ whose union is also in \mathcal{E} , $f(E \cup F) = f(E) + f(F)$. The function f is *countably additive* if for every disjoint sequence E_1, E_2, \dots of sets in \mathcal{E} whose union is also in \mathcal{E} , $f(\cup_{i=1}^{\infty} E_i) = \sum_{i=1}^{\infty} f(E_i)$.⁶ ■

Definition 5.4.3: Let \mathcal{E} be a σ -algebra over Ω . A probability measure P over (Ω, \mathcal{E}) is a countably additive function $P : \mathcal{E} \rightarrow [0,1]$ such that $P(\Omega) = 1$. ■

The requirement that P be countably additive is natural. It says that if we have a set of disjoint events, the probability that any of the events occur is the sum of the probabilities of each of the individual events. For example, the probability that a sentence begins with the words “Once upon a time...” is the sum over all lengths n of the probability that a sentence is of n words and begins with “Once upon a time...”. The second condition just says that the probability that a possible world is a possible world should be 1.

A probability distribution, as defined in Section 3.2, over a finite set of worlds Ω , is just a special case of probability measure defined here. The event space is 2^Ω , and for any subset E of Ω we define $P(E) = \sum_{\omega \in E} P(\omega)$. It is clear that P so defined satisfies the conditions of a probability measure.

5.4.3 Measure-Theoretic Semantics for RPMs

In order to define a probability measure over Ω_K , we must first choose an event space. Formally we proceed as follows:

⁶The meaning of the infinite sum is the limit of the partial sums $\sum_{i=1}^n f(E_i)$ if the limit exists, or ∞ if the sum grows without bound.

Definition 5.4.4: Let \mathbf{X} be a finite set of basic variables of \mathcal{K} , and let \mathbf{x} be an assignment of values to \mathbf{X} . The set $\{\omega \in \Omega_{\mathcal{K}} : [X_i]^\omega = x_i, \forall X_i \in \mathbf{X}\}$ is called a *basic event*, and is denoted $[\mathbf{X} = \mathbf{x}]$. The set of all basic events will be denoted \mathcal{F} . ■

The basic events are the fundamental elements in our event space. We would like our probability model to define the probability that a possible world will satisfy a set of assignments to basic variables, for any finite set of assignments. The set of basic events \mathcal{F} is not an algebra or σ -algebra, because it is not closed under unions. However, it is closed under intersections, and the set of finite or countable unions of basic events is a σ -algebra. We will take this set as our event space $\mathcal{E}_{\mathcal{K}}$. The set \mathcal{F} of basic events is called a *base* for $\mathcal{E}_{\mathcal{K}}$.

One property of this particular event space is that it distinguishes between every pair of possible worlds. By Theorem 5.2.13, no two possible worlds agree on the values of all basic variables, so for any pair of worlds ω and ω' , there is a basic event E such that $\omega \in E$, and $\omega' \notin E$, and vice versa. Thus the event space is actually quite a rich one.

We proceed to define a function f , that will specify the probability of any basic event.

Definition 5.4.5: Let \mathbf{X} be a finite set of basic variables of \mathcal{K} . The *flat BN relative to \mathbf{X}* , denoted $\mathcal{B}[\mathbf{X}]$, is a BN \mathcal{B} defined as follows:

- \mathcal{B} contains a node for every variable $X \in \mathbf{X}$, and for every ancestor Y of X in the \leftarrow relation.
- $\mathcal{G}[\mathcal{B}]$ contains an edge from Y to X if $X \leftarrow Y$.
- Let $X = I.\sigma.A$ be a node in \mathcal{B} . If σ is empty, let C be the type of I , otherwise let C be the range type of σ . The CPF associated with X in \mathcal{B} is derived from CPF_A in \mathcal{P}_C as follows. Let \mathbf{U} be the parents of X in \mathcal{B} . CPF_X is a function from $Val[\mathbf{U}] \times Val[X]$ to $[0, 1]$, defined by $CPF_X(\mathbf{u}, x) = CPF_A(x \mid v(\mathbf{u}))$, where $v(\mathbf{u})$ denotes the value assigned to $\theta(I.\sigma.v)$ in \mathbf{u} .

We define a function $f : \mathcal{F} \rightarrow [0, 1]$ by

$$f([\mathbf{X} = \mathbf{x}]) = P_{\mathcal{B}[\mathbf{X}]}(\mathbf{X} = \mathbf{x}). \quad \blacksquare$$

If $\mathcal{G}[\mathcal{K}]$ is acyclic, then by Theorem 5.3.8, and the fact that the number of direct parents of any basic variable is finite, a basic variable has a finite number of ancestors. So $\mathcal{B}[\mathbf{X}]$ contains a finite number of nodes. Also, if X is in $\mathcal{B}[\mathbf{X}]$ and $X \leftarrow Y$, then Y is also in $\mathcal{B}[\mathbf{X}]$, so $\theta(I.\sigma.v)$ is always a parent of $X = I.\sigma.A$ in \mathcal{B} for any parent v of A in the local probability model of A . Therefore CPF_X is well-defined, $\mathcal{B}[\mathbf{X}]$ is a well-defined BN, and the function f is well-defined.

Lemma 5.4.6: *If \mathbf{X} and \mathbf{Y} are finite sets of basic variables of \mathcal{K} , and $\mathcal{B}[\mathbf{Y}]$ contains every $X \in \mathbf{X}$, then for any value $\mathbf{x} \in \text{Val}[\mathbf{X}]$,*

$$P_{\mathcal{B}[\mathbf{Y}]}(\mathbf{X} = \mathbf{x}) = f([\mathbf{X} = \mathbf{x}]).$$

Proof: Since each $X \in \mathbf{X}$ is a node in $\mathcal{B}[\mathbf{Y}]$, X is either in \mathbf{Y} or the ancestor of some $Y \in \mathbf{Y}$. So every ancestor of X is in $\mathcal{B}[\mathbf{Y}]$. In addition, the parents and CPF of X in $\mathcal{B}[\mathbf{Y}]$ are the same as in $\mathcal{B}[\mathbf{X}]$. So the portion of $\mathcal{B}[\mathbf{Y}]$ consisting of \mathbf{X} and ancestors of \mathbf{X} is exactly the same as $\mathcal{B}[\mathbf{X}]$. Therefore

$$P_{\mathcal{B}[\mathbf{Y}]}(\mathbf{X} = \mathbf{x}) = P_{\mathcal{B}[\mathbf{X}]}(\mathbf{X} = \mathbf{x}) = f([\mathbf{X} = \mathbf{x}]). \quad \blacksquare$$

Lemma 5.4.7: *The function f is additive on \mathcal{F} .*

Proof: Let F be a basic event, and F_1, \dots, F_n be disjoint basic events, with $\cup_1^n F_i = F$. Write $F = [\mathbf{X} = \mathbf{x}]$, $F_i = [\mathbf{Y}_i = \mathbf{y}_i]$. Let \mathbf{Y} be $\cup_1^n \mathbf{Y}_i$, and consider $\mathcal{B}[\mathbf{Y}]$. Each of the F_i must mention all the \mathbf{X} , otherwise we would not have $F_i \subseteq F$. So $\mathbf{X} \subseteq \mathbf{Y}$, and, by Lemma 5.4.6,

$$P_{\mathcal{B}[\mathbf{Y}]}(\mathbf{X} = \mathbf{x}) = f([\mathbf{X} = \mathbf{x}]).$$

By the same lemma,

$$P_{\mathcal{B}[\mathbf{Y}]}(\mathbf{Y}_i = \mathbf{y}_i) = f([\mathbf{Y}_i = \mathbf{y}_i]).$$

But since the F_i are disjoint events whose union is F ,

$$f([\mathbf{X} = \mathbf{x}]) = P_{\mathcal{B}[\mathbf{Y}]}(\mathbf{X} = \mathbf{x}) = \sum_{i=1}^n P_{\mathcal{B}[\mathbf{Y}]}(\mathbf{Y}_i = \mathbf{y}_i) = \sum_{i=1}^n f([\mathbf{Y}_i = \mathbf{y}_i]). \quad \blacksquare$$

Lemma 5.4.8: *No basic event is the infinitely countable disjoint union of other basic events.*

Proof: Suppose not. Let $(F_i)_1^\infty$ be a sequence of disjoint basic events such that $\cup_1^\infty F_i$ is equal to basic event F . Each of the F_i must agree with F on the attributes mentioned by F . Let \mathbf{X}^i be the variables mentioned by F_i that are not mentioned by F , and let \mathbf{X} be $\cup_1^\infty \mathbf{X}^i$. Consider all possible assignments \mathbf{x} to \mathbf{X} . Since $F = \cup_{\{\mathbf{x} \in \text{Val}[\mathbf{X}]\}} (F \cap [\mathbf{X} = \mathbf{x}])$, every $F \cap [\mathbf{X} = \mathbf{x}]$ must be contained in some F_i . We will show by diagonalization that this is impossible.

\mathbf{X} must be infinite, because there are only a finite number of distinct basic events that mention only variables in a finite set, and it must be countable, since it is the countable union of finite sets. Let $(X)_1^\infty$ be an enumeration of \mathbf{X} . Consider an infinite tree, in which each node represents a basic event. The root is the event F , and each node G at depth m has a child $G \cap [X_i = x_i]$, for each value $x_i \in \text{Val}[X_i]$. Each infinite path through the tree represents an event $F \cap [\mathbf{X} = \mathbf{x}]$. We mark a node with an event F_i if every path going through the node is contained in F_i . We will say that a node in the tree is *blocked* if it is equal to or a descendant of some node marked by F_i , and we will call a node a *blocking node* if it is blocked but its parent is not blocked. The depth of a blocking node marked by F_i is equal to the largest index of the variables mentioned by F_i . Since the F_i are disjoint, no path can be marked by more than one F_i . Therefore there are blocking nodes marked by each of the F_i . Since all the \mathbf{X} are mentioned by some F_i , there are arbitrarily deep blocking nodes. Therefore, the subtree consisting of unblocked nodes is infinite, and by Königs lemma, it must contain an infinite path. This infinite path corresponds to a $F \cap [\mathbf{X} = \mathbf{x}]$

that is not contained in any F_i . ■

Lemma 5.4.9: *f is countably additive on \mathcal{F} .*

Proof: Immediate from Lemmas 5.4.7 and 5.4.8. ■

It is a basic fact of measure theory that any countably additive function f on a base for a σ -algebra can be uniquely extended to a measure μ on the σ -algebra. For any event $E \in \mathcal{E}$, if E is the finite union $\cup_1^n F_i$ of disjoint basic events, define $\mu(E) = \sum_1^n f(F_i)$. If E is the countable disjoint union $\cup_1^\infty F_i$, define $\mu(E) = \lim_{j \rightarrow \infty} \sum_1^j f(F_i)$. If the limit grows without bound, $\mu(E) = \infty$, though this cannot happen in our case. Countable additivity of f ensures that μ is also countably additive. Formally:

Theorem 5.4.10: *Let $\langle \mathcal{K}, \mathcal{P} \rangle$ be a single-valued relational probability model. There is a unique probability measure μ on $\langle \Omega_{\mathcal{K}}, \mathcal{E}_{\mathcal{K}} \rangle$, such that for any basic event $[\mathbf{X} = \mathbf{x}]$, $\mu([\mathbf{X} = \mathbf{x}]) = P_{\mathcal{B}[\mathbf{X}]}(\mathbf{X} = \mathbf{x})$.*

Proof: By Lemma 5.4.9 and the remarks following it, there is a unique measure μ on $\langle \Omega_{\mathcal{K}}, \mathcal{E}_{\mathcal{K}} \rangle$ such that $\mu([\mathbf{X} = \mathbf{x}]) = f([\mathbf{X} = \mathbf{x}]) = P_{\mathcal{B}[\mathbf{X}]}(\mathbf{X} = \mathbf{x})$. To show that μ is a probability measure, we must show that $\mu(\Omega_{\mathcal{K}}) = 1$. Let X be any basic variable. $\Omega_{\mathcal{K}} = \cup_{x \in \text{Val}[X]} [X = x]$, so

$$\mu(\Omega_{\mathcal{K}}) = \sum_{x \in \text{Val}[X]} \mu([X = x]) = \sum_{x \in \text{Val}[X]} P_{\mathcal{B}[\{X\}]}(X = x) = 1.$$

■

Definition 5.4.11: Let $\langle \mathcal{K}, \mathcal{P} \rangle$ be a single-valued RPM. We define the semantics of $\langle \mathcal{K}, \mathcal{P} \rangle$ to be the unique probability measure μ over $\Omega_{\mathcal{K}}$ such that $\mu([\mathbf{X} = \mathbf{x}]) = P_{\mathcal{B}[\mathbf{X}]}(\mathbf{X} = \mathbf{x})$. The probability measure μ will be denoted $P_{\mathcal{K}}$. ■

5.5 Inference

We want to compute the answer to a query of the form $P_{\mathcal{K}}(\mathbf{Q} \mid \mathbf{E} = \mathbf{e})$, for some set of query variables \mathbf{Q} , and some assignment \mathbf{e} to evidence variables \mathbf{E} . The query and

evidence variables are basic variables of \mathcal{K} .

One way to solve the query is to use the technique called *knowledge-based model construction (KBMC)* [98]. In this approach, a BN that is sufficient to answer the query is constructed, and the query is solved using standard inference algorithms in that BN. Let \mathbf{X} be the set of variables in $\mathbf{Q} \cup \mathbf{E}$. From the previous section, we know that $\mathcal{B}[\mathbf{X}]$ is sufficient to define a probability distribution over \mathbf{X} , and the answer to our query is equal to $P_{\mathcal{B}[\mathbf{X}]}(\mathbf{Q} \mid \mathbf{E} = \mathbf{e})$. We therefore need to construct $\mathcal{B}[\mathbf{X}]$, and solve the query in that BN.

As in the previous chapter, we are not satisfied with this approach, because it fails to exploit the object structure of the domain. Once again, we want a structured inference algorithm that can exploit the fact that the domain is described in terms of interacting objects, and that many of these objects have the same probability model.

The **SVE** algorithm from the previous chapter carries through in its essentials. Once again, the algorithm behaves recursively. To solve a query on an object, we eliminate each of the complex attributes of the object using a recursive call, to obtain a factor over the interface of the attribute. We then use standard **VE** to solve the local query. As before, we can exploit the fact that many objects have the same probability model by defining the **SVE** algorithm in terms of the class probability models rather than specific instances.

However, there are two issues that make the algorithm more complicated here. The first issue that needs to be resolved is that objects no longer have a single, clearly defined interface. The interface of an object depends on how attributes of other objects refer to it, and on the particular query asked, and it can be arbitrarily complex.

The second issue is that in the presence of instance statements, encapsulation does not hold in the same way that it did in hierarchical models. In an OOBN, all the objects in a model were organized in a tree structure. That is no longer the case here; we may have a network of inter-related instances whose structure forms a graph, not a tree. For example, suppose we have three named instances I , J and K , and suppose we have the instance statements $I.A = J$, $J.B = K$ and $I.C = K$. It is not the case that K is encapsulated from I by the interface between I and J .

First we describe how the recursive algorithm deals with the first issue. The basic idea is that when solving a query on class C , we need to compute the interface of each complex attribute of C on the fly, as we perform the computation. For this purpose, we maintain a *Needed* set for each attribute. The set $Needed[A]$ consists of attribute chains on the range type of A , that are used by other attributes. At the beginning of the **SVE** computation, $Needed[A]$ is initialized to contain the query and evidence chains beginning with A . When we process an attribute B , we discover what attribute chains it depends on. If B depends on the chain τ , and $\tau = A.\tau'$, then we add $A.\tau'$ to $Needed[A]$. We process attributes in a bottom-up fashion, so that an attribute is not processed until all attributes that may depend on it have been processed. The class dependency graph $\mathcal{G}[C]$ is used to determine the order in which attributes are processed. We make sure that no attribute is processed until all its children in $\mathcal{G}[C]$ have been processed.

For a simple attribute B , it is easy to determine what chains B depends on — it simply depends on all the parents v of B in \mathcal{P}_C . For a complex attribute A , if A has no inverse, A cannot depend on any other chains, so we can simply compute a probability distribution over $Needed[A]$ when eliminating A . However, suppose A does have an inverse B . Let c be an entity of the range type of A , and let $c.A$ be d , of type C' . Since B *inverse-of* A , $d.B$ is c . Processing the attribute A results in a recursive call to a query on C' , in which the set of query variables is $Needed[A]$. Suppose that during the processing of the recursive query, we find that some chain $B.\tau$ is needed. The value of $B.\tau$ on d is actually the value of τ on c . Therefore $B.\tau$ is not encapsulated from c by the interface between c and d . So the chain τ must be part of the interface between c and d .

The fact that τ is part of the interface between c and d is not known at the time that A is processed, during the computation on C . It is a result of the fact that the recursive computation on C' requires the chain $B.\tau$. This information is only available in the model for C' , and is determined during the recursive computation for the query on C' . For this reason, we have the recursive **SVE** function return two values. One is the set of such chains τ that are part of the interface between c and d , that are not in $Needed[A]$. This is the set of chains τ such that a chain $B.\tau$ appears

during the recursive computation for an inverse B of A . The second return value is a factor over the entire interface between c and d . I.e., it is a factor over $Needed[A] \cup \tau$. In fact, the $Needed[A]$ correspond to the outputs of d in OOBN terminology, and the τ behave like inputs, so the factor returned by **SVE** expresses $P(Needed[A], e \mid \tau)$.

To assist in bookkeeping, **SVE** maintains $Needed[A]$ also for simple attributes A . For these attributes $Needed[A]$ serves as a flag to indicate whether the attribute is needed at all during the computation. If $Needed[A] = \emptyset$ at the time A is processed, then A is ignored completely.

We are now ready to describe the recursive part of the **SVE** algorithm. It takes five arguments. The first four are the same as in OOBNs: a class C , a set of query chains σ , a set of evidence chains ρ , and an assignment $e \in Val[\rho]$. The fifth argument is the attribute D through which **SVE** was called; it will be used to detect when a complex attribute of C should not be processed recursively, but should be treated as an inverse attribute, as described above. The algorithm returns a set of input chains τ , and a factor over $\sigma \cup \tau$.

Algorithm StructuredVariableElimination(C, σ, ρ, e, D)

For each attribute A of C do

$Needed[A] = \emptyset$.

For each $\sigma \in \sigma \cup \rho$ do:

 If $\sigma = A.\sigma'$

$Needed[A] = Needed[A] \cup \{\sigma\}$.

For each attribute A of C ,

 in a bottom-up order consistent with $\mathcal{G}[C]$ do:

 If $Needed[A] \neq \emptyset$

 If A is simple

 If A is a chain $\rho \in \rho$

 Let a be the value assigned to A in e

$g_A = CPF_A[A = a]$.

 Else

$g_A = CPF_A$.

```

Else // A is complex
  If not A inverse-of D
    Let C' be the range type of A.
    Let  $\sigma' = \{\sigma' : A.\sigma' \in Needed[A]\}$ .
    Let  $\rho' = \{\rho' : A.\rho' \in \rho\}$ .
    Let e' be the value in Val[ $\rho'$ ],
      such that for each  $\rho' \in \rho'$ ,
      the value assigned to  $\rho'$  in e'
      is the same as the value assigned to A. $\rho'$  in e
     $\langle \tau', f_A \rangle = \text{StructuredVariableElimination}(C', \sigma', \rho', e', A)$ .
     $g_A = \text{Rename}(f_A, \psi)$  where
       $\psi(B.\tau) = \tau$  if B inverse-of A,
       $\psi(B.\tau) = A.B.\tau$  otherwise.
    For each B. $\sigma$  mentioned by  $f_A$ 
       $Needed[B] = Needed[B] \cup \{\sigma\}$ .

 $\tau = \cup_{\{B:B \text{ inverse-of } D\}} Needed[B]$ .
 $f = \{g_A : Needed[A] \neq \emptyset\}$ .
Let  $\phi = \{\phi : \phi \text{ is mentioned by some } f \in f\} - (\sigma \cup \tau)$ .
For each  $\phi \in \phi$  do
  Let g be  $\{g \in f : g \text{ mentions } \phi\}$ .
   $h_\phi = \prod g$ .
   $k_\phi = \sum_\phi h_\phi$ .
   $f = f - g \cup \{k\}$ .
 $f = \prod f$ .
Return  $\langle \tau, f \rangle$ .

```

Example 5.5.1: We use the dependency model from Example 5.3.2 to illustrate the use of the *Needed* sets. First, suppose there is a query on the *Person* class, asking for the probability distribution over the value of *Happy* given that *Mother.Healthy* is true. *Needed*[*Happy*] is initialized to {*Happy*}, while *Needed*[*Mother*] is initialized to {*Mother.Wealthy*}. The *Needed* sets for the other attributes are initialized to be

empty.

Let us suppose that the attribute of **Person** are processed in the order **Healthy**, **Happy**, **Mother**, **Father** and **Wealthy**, which is consistent with $\mathcal{G}[\text{Person}]$. When **Healthy** is processed, $\text{Needed}[\text{Healthy}]$ is empty, so it is ignored. Next, **Happy** is processed, and $\text{Needed}[\text{Happy}]$ is non-empty. One parent of **Happy** is **Mother.Father.Wealthy**, which is added to $\text{Needed}[\text{Mother}]$, while the other parent is **Father.Healthy**, which is added to $\text{Needed}[\text{Father}]$. Next, **Mother** is processed, resulting in a recursive call to **SVE**. $\text{Needed}[\text{Mother}]$ is $\{\text{Mother.Healthy}, \text{Mother.Father.Wealthy}\}$, so the query chains are **Healthy** and **Father.Wealthy**, while the evidence asserts that **Healthy** is true. Because **Mother** has no inverses, the first return value τ returned by this recursive call is empty. So no chains are added to the *Needed* sets of other attributes after the recursive call returns. The **Father** attribute is processed next, resulting in another recursive call, with the query variable **Healthy**. Again the return value τ is empty. Finally, the **Wealthy** attribute is ignored because NeededWealthy remains empty.

Now consider a query on the **Married-Man** class, asking for a distribution over the value of **Happy**. In this class, **Happy** has the parent **Husband-Of.Happy**, so processing **Happy** results in **Husband-Of.Happy** being placed in $\text{Needed}[\text{Husband-Of}]$. Processing **Husband-Of** then results in a recursive query on the **Married-Woman** class, with the query variable **Happy**. The fifth argument to the recursive call is the attribute **Husband-Of**. When the **Happy** attribute of the **Married-Woman** class is processed, **Wife-Of.Mother.Happy** is added to $\text{Needed}[\text{Wife-Of}]$. When **Wife-Of** is processed, the algorithm detects that it is an inverse of **Husband-Of**, so nothing is done. At the end of the recursive call, however, the return value τ contains the chain **Wife-Of.Mother.Happy**, which is then renamed to **Mother.Happy** in the original **SVE** computation on the **Married-Man** class. The chain **Mother.Happy** is therefore added to $\text{Needed}[\text{Mother}]$. Note that since **Mother** is in $\text{Exp}[\text{Husband-Of}]$, it precedes **Husband-Of** in $\mathcal{G}[\text{Married-Man}]$, and is therefore processed after **Husband-Of**. ■

This recursive algorithm is applied only to unnamed, generic instances of a class C , and not to named instances. As we said earlier, encapsulation does not always apply in the presence of instance statements. Since there can be no instance statements

about unnamed instances, it is safe to apply the recursive algorithm to them. However, we need to handle named instances separately. Because attributes of different named instances may be interleaved in $\mathcal{G}[\mathcal{K}]$, we must process all the named instances together. For this purpose, we create a special top-level instance that contains all attributes of all named instances.

Definition 5.5.2: Let $\langle \mathcal{K}, \mathcal{P} \rangle$ be a RPM. The *top-level object* of \mathcal{K} , denoted T , is an object⁷ defined as follows:

- T contains an attribute $[I.A]$ for each attribute A of a named instance I in \mathcal{K} . $[I.A]$ is simple or complex according to whether A is simple or complex, and its range type is the same as that of A .
- If $[I.A]$ is simple, let C be the class of I , and let \mathbf{v} be the parents of A in \mathcal{P}_C . For each $v \in \mathbf{v}$, let $J.B.\sigma$ be $\theta(I.v)$. Then $[I.A]$ has a parent $[J.B].\sigma$ in \mathcal{P}_T .⁸ The CPF of $[I.A]$ in \mathcal{P}_T is the same as the CPF of A in \mathcal{P}_C . ■

A query on a RPM can now be solved via a call to **SVE** on the top-level object. The final argument D to this call will be null, since the query on the top-level object is not invoked from within another query. **SolveQuery** takes a set of query variables \mathbf{Q} , a set of evidence variables \mathbf{E} and a value $e \in \text{Val}[\mathbf{E}]$, and returns $P(\mathbf{Q} \mid \mathbf{E} = e)$. First the query and evidence variables are turned into attribute chains on the top-level object. **SVE** is called on the top-level object. The first returned argument will always be empty, since the last argument to the **SVE** call is empty. The second returned argument is equal to $P(\mathbf{Q}, \mathbf{E} = e)$, but expressed in terms of chains on the top-level object. It is renamed, and then normalized to obtain the desired answer to the query.

Algorithm SolveQuery($\mathbf{Q}, \mathbf{E}, e$)

Let σ be $\{[I.A].\sigma' : I.A.\sigma' \in \mathbf{Q}\}$.

⁷ T can be viewed as an instance object or as a class object with a single instance. Since the distinction is not important, we just call T an object.

⁸The notation $[J.B].\sigma$ indicates the fact that $[J.B]$ is actually a single attribute of T , as opposed to $J.B.\sigma$, which begins with the instance J followed by the attribute B .

Let ρ be $\{[I.A].\rho' : I.A.\rho' \in \mathbf{E}\}$.
 Let e' be the value in $Val[\rho]$,
 such that for each $[I.A].\rho' \in \rho$,
 the value assigned to $[I.A].\rho'$ in e'
 is the same as the value assigned to $I.A.\rho'$ in e .
 $\langle \emptyset, g \rangle = \mathbf{SVE}(T, \sigma, \rho, e', \emptyset)$.
 $f = \mathbf{Rename}(g, \psi)$, where
 $\psi([I.A].\sigma) = I.A.\sigma$.
 Return $\mathbf{Normalize}(f)$.

As we did with OOBNS, we can take advantage of the fact that many objects have the same probability model by caching solutions returned by calls to **SVE**. We now need to use all five arguments as indices to the cache. The cache will mainly be useful for computations on generic unnamed instances of a class, since there is only one call per query to the top-level object, and this call will vary from query to query. We can also implement the improvement discussed at the end of Section 4.5.5, in which **SVE** returns a set of factors rather than a single factor.

Most of the complexity analysis from the previous chapter carries over to here. The cost of inference within a single **SVE** call is dominated by the **VE** phase at the end, and is exponential in the induced width M of the graph describing the **VE** computation. Once again, M depends on the induced width of $\mathcal{G}[C]$ and on the sizes of the interfaces. However, since the size of the interface of an object is not an inherent property of the class of the object, but is dependent on other object models and on the query, it is harder to bound the cost of inference based solely on local considerations. Furthermore, for the named instances, one needs to consider the induced width of the top-level object, which cannot be determined locally based on the models of the individual instances. We could potentially utilize some encapsulation even at the top level, but we choose not to for simplicity.

As a result, one cannot provide compositional performance guarantees for general RPMs in the way that one can for OOBNS. This is the price paid for the more general representation language. It is our intuition that the decomposition of a model in terms of interacting entities will naturally lead to small interfaces in most cases, so the fact

that the sizes of interfaces are unbounded in principle will often not be an issue in practice. If the interfaces are in fact small, our algorithm will exploit that fact.

The need to perform inference for all the named instances in a single top-level object can also lead to performance difficulties. The cost of inference will depend on the tree-width of the top-level project, and we believe the presence of many interconnected named instances will in some cases lead to a high tree-width for the top-level object in practice. As a result, exact inference may not be feasible for the top-level object in these cases. We believe that this may be a more serious issue than the unbounded sizes of interfaces, and shall discuss this matter further in Section 6.3.

5.6 Discussion

5.6.1 Integration with OOBNs

While relational probability models are more flexible than OOBNs, because they allow general relations and interconnected instances, there are some things that can be expressed in the OOBN framework but not in RPMs. For one thing, if the relationship between two objects really is hierarchical, this can be exploited in the OOBN framework, while in the RPM framework the two objects are treated symmetrically. A RPM can be used to model a hierarchical relationship, but it is harder to do. Suppose C and C' are two RPM classes, and we want a hierarchical relationship between them. We give C an attribute A of range type C' , and C' an attribute B of range type C , and specify that B *inverse-of* A . As a result, an object of type C' can depend on attributes of its containing object via B , and the containing object can in turn depend on the contained object via A . All attributes of the containing object on which the contained object depends are in $Exp[A]$, so they must precede all attributes of the containing object which depend on the contained object in $\mathcal{G}[C]$. In implementing a hierarchical relationship in this way, it is critical that only one inverse statement is used, in the specified direction. If a statement A *inverse-of* B is added, then $\mathcal{G}[C']$ will be cyclic.

Another difference between OOBNs and RMPs is that the OOBN framework uses

the binding mechanism to pass information between objects. The binding mechanism corresponds to a “push” model of information passing, because the information is passed to the contained object from its container, whereas the RPM framework uses a “pull” model of information passing, because the passing of information is controlled by the object that uses the information. There are situations in which the push model is advantageous. For example, suppose a class C has simple attributes A_1 and A_2 with the same range, and complex attributes B_1 and B_2 of class C' . We may want our model to specify that B_1 depends on A_1 , while B_2 depends on A_2 , but this cannot be done in a pull model — the model for C' would have to specify a dependence on one or the other of A_1 and A_2 , which would then be used for both B_1 and B_2 . The only way to achieve the desired effect in a pull model is to create two subclasses of C' , which is unwieldy. A push model, on the other hand, can easily handle this situation, because the model for C can state explicitly that A_1 is passed to B_1 and A_2 is passed to B_2 .

As we have discussed, OOBNs also have an advantage over more general RPMs because OOBN components have clearly defined interfaces, and we can provide compositional performance guarantees using these interfaces. Even though hierarchical models may be insufficient for completely representing a system, it is clear that most complex systems are at least partly hierarchical. We therefore want to preserve the advantages of hierarchical models, without losing the flexibility provided by general relational models. We can easily combine the two representation languages to obtain the advantages of both. Some of the relations in a relational model can be designated as component functions, implying a hierarchical relationship between a container and contained object. The container can define bindings for the contained object, as in OOBNs. Other relations would behave just as in relational models.

In order for the container to bind the value of an attribute of the contained object, that attribute should not have parents in the contained object’s model. Otherwise, we would want to condition the parents of the bound attribute on its value. It would be difficult to give semantics to the resulting model in terms of a generative process. Therefore, we only allow roots of the class dependency graph of the contained object to be bound by its container. All such roots are potential inputs of the object.

The development of the syntax and semantics for the combined representation language should be clear, so we do not provide any more details. One issue that needs to be dealt with in inference is combining the on-the-fly computation of interfaces with the OOBN style of passing information between objects. This can be achieved by replacing the last argument to **SVE**, which is currently the attribute through which **SVE** is called, with a set of attributes whose values are to be obtained from the calling object. This set will include both inverses of the attribute through which **SVE** is called and attributes whose values are bound by the calling object. An attribute in this set is to be treated in the same way that an inverse of the entry-point attribute is currently treated — no recursive call is made for the attribute, and chains beginning with the attribute are not eliminated during the variable elimination phase. The modified **SVE** algorithm is as follows (only the changed parts of the algorithm are shown):

Algorithm StructuredVariableElimination(C, σ, ρ, e, D)

```
// Initialization and processing of simple attributes are same as before
// Processing of complex attributes is now as follows
Else // A is complex
  If not  $A \in D$ 
    Let  $C'$  be the range type of  $A$ .
    Let  $\sigma' = \{\sigma' : A.\sigma' \in \text{Needed}[A]\}$ .
    Let  $\rho' = \{\rho' : A.\rho' \in \rho\}$ .
    Let  $e'$  be the value in  $\text{Val}[\rho']$ ,
      such that for each  $\rho' \in \rho'$ ,
      the value assigned to  $\rho'$  in  $e'$ 
      is the same as the value assigned to  $A.\rho'$  in  $e$ 
    Let  $D' = \{B : B \text{ inverse-of } A\} \cup \{B : \text{there exists a binding } \Theta[A.B]\}$ .
     $\langle \tau', f_A \rangle = \text{StructuredVariableElimination}(C', \sigma', \rho', e', D')$ .
     $g_A = \text{Rename}(f_A, \psi)$  where
       $\psi(B.\tau) = \tau$  if  $B$  inverse-of  $A$ ,
       $\psi(B.\tau) = \Theta[A.B].\tau$  if there is a binding  $\Theta[A.B]$ ,
       $\psi(B.\tau) = A.B.\tau$  otherwise.
```

For each $B.\sigma$ mentioned by f_A

$$Needed[B] = Needed[B] \cup \{\sigma\}.$$

$$\tau = \cup_{D \in \mathcal{D}} Needed[D].$$

// Remainder of variable elimination phase is same as before.

5.6.2 Isomorphic Worlds

Recall that in Section 4.2.3, we addressed the issue of isomorphic worlds, that differ only in the names of the domain entities. As we said there, we do not want to consider isomorphic worlds as really being different as far as the probability model is concerned. We resolved the issue there by explicitly specifying the set of domain entities, and we used the same solution in this chapter. However, we mentioned that an elegant solution to the problem uses measure theory, and we describe that solution here.

The solution is in fact very simple. Rather than restricting the set of possible worlds in our probability space to contain only those worlds containing a particular set of domain entities, we allow the set of possible worlds to include all worlds that have the right kind of structure, regardless of the identity of the domain entities. However, we continue to specify the event space in terms of basic events, which are *formulas* on possible worlds. In particular, the basic event $[\mathbf{X} = \mathbf{x}]$ holds in a possible world ω if for all $X_i \in \mathbf{X}$, $[X_i]^\omega = x_i$. For a basic variable X_i , $[X_i]^\omega$ is a well-constructed term even if the names of the domain elements are different from those specified in Definition 5.2.9. Furthermore, the value of $[X_i]^\omega$ does not depend on the names of the domain elements, and is the same in all isomorphic worlds. Therefore, if ω and ω' are isomorphic, they will belong to exactly the same events in the event space, and will be indistinguishable as far as the probability measure is concerned.

5.6.3 Conclusion

In this chapter, we extended our probabilistic representation language beyond hierarchical models to more general relational probability models. After presenting the

basic language definitions and describing the relational structure of the models for this language, we discussed the problem of making sure that the probability model is acyclic and non-recursive. We showed that the problem is more complex than for hierarchical models, and requires a global dependency graph over the attributes of all objects in the model. We then presented semantics for the language. Intuitively, the semantics is defined through a generative process, from which values of particular attributes can be generated using lazy evaluation. We showed that this intuition can be formalized by defining a probability measure over the set of possible worlds. Next, we presented a structured inference algorithm for relational probability models, extending the **SVE** algorithm for OOBNs to deal with the more general case. In particular, we showed how object interfaces could be computed on the fly during the process of solving a query, and how to deal with interconnected instances.

Even though inference in RPMs is more difficult than in OOBNs, we feel that the added expressive power is well worth the cost. When modeling complex situations, it is crucial to be able to talk about general, non-hierarchical relationships between the object, and to be able to talk about interconnected instances. In OOBNs, we blindly assumed that all the entities in a domain are connected in a tree structure. This assumption loses a lot of important information, and the properties of different objects may be assumed to be independent or conditionally independent when in fact they are not. By allowing interconnected instances in RPMs, we allow richer and more accurate models to be represented. Importantly, this allows us to derive more inferences from evidence about one object to properties of other objects.

By separating the domain model into fixed class probability models and a flexible relational model, we obtain a probabilistic representation language that is both powerful and flexible. Furthermore, a model designer has the ability to control the degree to which the interconnectivity structure of the domain objects is modeled in detail. A model in which the full interconnectivity structure is made explicit is more accurate and allows richer inferences, but comes at the cost of more expensive inference. The modeler needs to represent only those interconnections between objects that are absolutely crucial. By making the unique names assumption about the objects that are not made explicitly, we allow the inference algorithm to take advantage

of encapsulation for all the unnamed objects.

In the models discussed in this chapter, we still maintained the restriction that all complex attributes are single-valued. This restriction will be relaxed in the next chapter, when we consider multi-valued attributes. In defining the set of possible worlds, we made certain assumptions, in particular the unique-names assumption, that guaranteed that the relational structure of the possible worlds are fully known, for a particular KB. In the next chapter, we will describe extensions to the language that allow us to express uncertainty about the relational structure of the possible worlds. We also made the restriction in the current chapter that the dependency model should not contain infinite dependency chains. In Chapter 7, we consider recursive probability models that do contain such infinite chains.

Chapter 6

Structural Uncertainty

6.1 Introduction

In the previous chapter, we allowed entities in the domain to be inter-related, but we made strong assumptions about the relational model. We assumed that all relationships were functional, that is, for every object X with complex attribute A there is a single object Y such that $X.A = Y$. Clearly, this assumption severely limits the expressive power of our models. For example, there may be many students registered for a course, and each student may take a number of courses. The attributes `Student.Taking` and `Course.Registered` are *multi-valued*.

In this chapter, we gradually relax the restrictions concerning multi-valued attributes. First, in Section 6.2, we allow multi-valued attributes where the number of values is fixed in advance. Semantically, this case is a very simple extension of single-valued relational models, but the introduction of multi-valued attributes even in this highly restricted manner requires us to introduce new tools for representation and inference.

Next, in Section 6.3, we consider models in which the number of values of a multi-valued attribute is allowed to vary, but the set of values is always fully specified. These types of models are well suited to augmenting a relational database with a probabilistic model. The semantics of these models is quite different from that of the previous chapter. We make the closed world assumption, so that the entities in the

domain and the relationships between them are only those specified explicitly in the database, and there is no need to consider generic, unnamed instances of a class.

Once we allow the number of values of a multi-valued attribute to vary, and consider models in which the set of values is not known, we open the door to a whole new kind of uncertainty, which we call *structural uncertainty*. This is uncertainty about the relational structure of the model itself, in this case about the number of values of a multi-valued attribute. This type of structural uncertainty is called *number uncertainty*, and is the topic of Section 6.4. By introducing *structural attributes* into the language, we allow the structural uncertainty to be incorporated directly into the class probability models. The result is a rich and expressive language, in which we can apply the techniques of Bayesian network inference to reasoning about the relational configuration of a system.

There are other kinds of structural uncertainty besides uncertainty over the number of values of a multi-valued attribute. One kind is uncertainty over the type of a complex attribute. For example, we may not know whether a course taken by a student is a math course or a humanities course. Another kind is uncertainty over the value of a complex attribute, i.e., uncertainty over which other entities an entity is related to. For example, we may not know whether Jane Studios is taking CS121 or CS221. Again, we model these kinds of structural uncertainty by introducing explicit structural attributes. Section 6.5 discusses these kinds of structural uncertainty, and Section 6.6 discusses possible extensions to the language to model even richer flavors of structural uncertainty.

6.2 Multi-Valued Attributes

6.2.1 Language

We begin this chapter by considering a relational language that allows multi-valued attributes, but in which the number of values of a multi-valued attribute is fixed. Formally, we allow any typed relational language $\langle \mathbf{C}, \sqsubseteq, \mathbf{A}, \mathbf{f}, \mathbf{R}, \mathbf{I} \rangle$, and no longer

require \mathbf{R} to be empty, but we associate a value $\#[R]$ with each relation R , determining the number of values of R . If the domain type of R is the class C , and $\#[R] = m$, then for any entity c of C , there will be exactly m entities d_1, \dots, d_m such that $R(c, d_i)$. We will call a language of this form a *fixed-number relational language*.

We allow a knowledge base to contain instance and inverse statements concerning multi-valued attributes. A KB may contain the statement $J \in I.A$, where I and J are named instances, and A is a multi-valued attribute. The meaning of this statement is that one of the values of $I.A$ is J . The KB may in fact contain up to $\#[A]$ such statements. We assume that different instance statements on the same $I.A$ are numbered, so we can speak of the first value of $I.A$, the second value, and so on.

Several different types of inverse statements are possible with multi-valued attributes. All have the form B *inverse-of* A , where A and B are complex attributes. An inverse statement involving two single-valued attributes is called a *one-one statement*. A *one-many statement* is one in which A is multi-valued and B is single-valued. A single entity I is related to multiple entities J_1, \dots, J_m via A , which are all related to I via B . For example, a professor may teach a number of courses, but a course normally only has a single instructor — A is the **Teaches** attribute of **Professor**, while B is the **Instructor** attribute of **Course**. A *many-one statement* is one in which A is single-valued while B is multi-valued. In this case, if I is related to J via A , then I must be one of the values of $J.B$. For example, A may now be **Course.Instructor** while B is **Professor.Teaches**. A *many-many statement*, of course, is one in which both A and B are multi-valued. In this case, an entity I is related to multiple entities J_1, \dots, J_m via A , and for each of the J_i , I is one of the values of the $J_i.A$. An example of this case is the relationship between a course and the students taking the course.

An interpretation ω for a typed relational language that has multi-valued attributes must specify $[R]^\omega$ for each relation $R \in \mathbf{R}$, in addition to all the other things specified for single-valued relational languages. In addition, since the language specifies $\#[R]$, we force ω to respect that specification by adding the constraint that for every entity c of the domain type of R , the number of values in $[R]^\omega(c)$ must be $\#[R]$. Also, the instance and inverse statements involving multi-valued attributes must all be respected. Formally:

Definition 6.2.1: Let \mathcal{K} be a knowledge base for a fixed-number relational language \mathcal{L} . A *possible world* for \mathcal{K} is an interpretation ω of \mathcal{L} such that:

1. For every relation R with domain type C , and every $c \in [C]^\omega$, $|\{d : [R]^\omega(c, d)\}| = \#[R]$.
2. For every instance statement $I.A = J$ in \mathcal{K} , where A is single-valued, $[I.A]^\omega = [J]^\omega$.
3. For every instance statement $J \in I.A$ in \mathcal{K} , where A is multi-valued, $[A]^\omega([I]^\omega, [J]^\omega)$.
4. For every inverse statement B *inverse-of* A in \mathcal{K} , and every pair of entities $c, d \in \Delta^\omega$, $[A]^\omega(c, d)$ implies $[B]^\omega(d, c)$.¹
5. ω satisfies the *unique names assumption*. I.e, there is no interpretation ω' of \mathcal{K} satisfying conditions 1–4 such that the set of identities in ω' is a proper subset of the set of identities in ω .
6. There is no proper subworld of ω satisfying conditions 1–4. ■

To illustrate what the interconnectivity structure looks like with the unique names assumption and many-many inverses, consider a KB \mathcal{K} with **Student** and **Course** classes, where **Student.Taking** and **Course.Enrolled-In** are inverses. It is of course unrealistic to assume that the number of courses taken by a student and the number of students in a course are fixed, but for now, assume that $\#[\text{Taking}]$ is 4, while $\#[\text{Enrolled-In}]$ is 30. Suppose that \mathcal{K} contains the instance *Jane-Student* of type **Student** and *CS221* of type **Course**, and the statements *Jane-Student* \in *CS221.Enrolled-In*, and *CS221* \in *Jane-Student.Taking*.

A possible world for \mathcal{K} has the following structure. There are two entities corresponding to the named instances, which of course are related to each other. In addition, there are 29 other **Student** entities related to *CS221*. Each of those students, as well as *Jane-Student*, is related to three other **Course** entities besides *CS221*. Because of the unique-names assumption, these other course entities are all distinct.

¹This clause applies to all the different kinds of inverse statements.

Each of them is related to 29 other students, which are all distinct, and all different from the students mentioned so far. Each of these new students is related to 3 other courses, and so on.

At this point we could proceed with the program of the previous two chapters, showing that the structure of possible worlds is fixed, and that there is a one-to-one correspondence between some set of attribute chains and domain entities. When a multi-valued attribute appears in a chain, we would have to include an index to distinguish between the different values of the attribute, but other than that, the argument is much the same as before, and we omit the details.

We could also follow the approach in previous chapters of restricting the possible worlds to be those containing a certain specific set of domain entities. However, from now on we shall use the approach described at the end of the previous chapter, in which we use the event space to render indistinguishable worlds that differ only in the names of the entities that are in them. We therefore allow all worlds satisfying Definition 6.2.1 into the set Ω_K of possible worlds.

In defining the class probability models, we need to specify a way for the properties of an object to depend on properties of other objects that are related to it by a multi-valued attribute. Rather than depending on each of the individual related objects directly, we allow the first object to depend on the other objects via some *aggregate* property of the set of related objects. Such aggregate properties are expressed through *quantifiers*.

Definition 6.2.2: A *quantifier* on class C has the form $\#[A.\sigma = v]$, where A is a multi-valued attribute of C , σ is a simple single-valued attribute chain on the range type of A , and v is a value in $Val[\sigma]$.² We assume for convenience that a quantifier $A.\sigma$

²Quantifiers as we define them here are closely related to the traditional existential and universal quantifiers in first-order logic. For example, the universally quantified expression $\forall c : I.A = c \implies c.\sigma = v$ can be expressed as $\#[A.\sigma = v] = \#[A]$. Our use of the word “quantifier” is actually very close to its literal meaning of “counter”.

Other types of aggregation operators, such as average or sum, are used in database query languages, and may sometimes be meaningful for our models. For example, the evaluation of a school may depend on the average number of students in a class. In the language presented here we shall not deal with aggregates other than quantifiers, but they can be added to the language without difficulty. On the other hand, performing probabilistic inference with these more sophisticated aggregates may be extremely expensive.

will never contain the sequence $B.D$ where D *inverse-of* B . A quantifier is interpreted in a possible world ω as a function from $[C]^\omega$ to the integers $\{0, 1, \dots, \#[A]\}$ as follows:

$$[\#[A.\sigma = v]]^\omega(c) = |\{d : d \in [A]^\omega(c) \text{ and } [\sigma]^\omega(d) = v\}|. \quad \blacksquare$$

An example of a quantifier is the expression $\#[\text{Taking.Workload} = \text{heavy}]$, defined on the **Student** class, which in our example takes on values between 0 and 4. Quantifiers can appear as parents of simple attributes in a class probability model. Thus, a dependence on the values of a multi-valued attribute can be modeled by introducing a quantifier, and making the quantifier a parent of simple attribute. For example, the quantifier $\#[\text{Taking.Workload} = \text{heavy}]$ may be a parent of the simple attribute **Tired** of **Student**. Of course, we also allow the parent of an attribute to be a simple single-valued attribute chain, as before.

Multi-valued attributes and quantifiers introduce no new complications when it comes to making sure that a dependency model is acyclic and non-recursive. If a simple attribute A of C has a quantifier $\#[B.\sigma = v]$ as a parent, we simply add an edge from B to A in $\mathcal{G}[C]$. In the global dependency graph $\mathcal{G}[\mathcal{K}]$, we add an edge from $C.B$ to $C.A$, and also from $I.B$ to $I.A$ for an instance I of type C . We must also modify the imports sets of attributes to account for quantifiers. If a quantifier $\#[\sigma = v]$ appears as the parent of some attribute, and σ contains the sequence $A.B$, then B is added to $\text{Imp}[A]$.

6.2.2 Semantics

The generative process semantics for fixed-number RPMs is similar to those for single-valued RPMs. In the generative process, whenever the values of a multi-valued attribute A need to be generated, $\#[A]$ different entities of the appropriate class are created rather than a single entity. If some of the values of A are already bound due to instance and inverse statements, only enough new entities are generated to make up $\#[A]$. The individual values of A are then generated independently of each other, according to the probability model for the range type of A .

In defining the formal measure-theoretic semantics, we follow the same process

as before, defining a set of basic variables, and defining basic events to be formulas consisting of assignments of values to a finite number of basic variables. We define a function f that assigns a probability to each basic event, using a flat BN sufficient to computing the probability distribution over the variables mentioned by that event. We define the event space to consist of finite and countable unions of basic events, and extend f to a probability measure over the event space.

Our approach here does differ from that of the previous chapter in one respect. The set of basic variables no longer consists of all simple attributes of all domain entities. The reason is that we do not distinguish between different values of the same multi-valued attribute. For example, suppose instance I has a multi-valued attribute A with two values, which we will call $I.A[1]$, and $I.A[2]$. Suppose each of the $I.A[i]$ have a simple attribute B . In one possible world, $I.A[1].B$ has the value *True*, while $I.A[2].B$ is *False*, and in another possible world the opposite holds. As far as we are concerned, these two possible worlds should be indistinguishable. They both satisfy the property $I.\#[A.B = \text{True}] = 1$. It is $I.\#[A.B = \text{True}]$ that we wish to be a basic variable, not the various $I.A[i].B$. We therefore distinguish between *variables*, which denote chains that have a specific simple value in a possible world, and *basic variables*, which are variables with which we describe events.

From a measure-theoretic point of view, two possible worlds ω and ω' are indistinguishable if for every event E in the event space, $\omega \in E \iff \omega' \in E$. By defining the event space in terms of basic variables, and stipulating that $I.\#[A[1].B = \text{True}]$ is a basic variable, but $I.A[1].B$ and $I.A[2].B$ are not, we achieve the desired effect. If ω is a possible world in which $I.A[1].B$ has value *True* and $I.A[2].B$ has value *False*, while ω' is a world that agrees with ω in every respect except that the values of $I.A[i].B$ are switched, then indeed there is no event in the event space that can distinguish between ω and ω' , so they are indistinguishable. Formally, we make the following definitions.

Definition 6.2.3: Let \mathcal{K} be a fixed-number relational language. An *indexed chain* of \mathcal{K} is a chain $I.A_1 \dots A_n$, where, for $i < n$, A_i is either a single-valued complex attribute, or has the form $B[j]$, where B is a multi-valued complex attribute and $1 \leq j \leq \#[B]$, and A_n is one of the above or a simple attribute or a quantifier.

A *standard indexed chain* is an indexed chain σ that satisfies the following properties:

1. σ does not begin with $I.A$, with A single-valued and an instance statement $I.A = J$ in \mathcal{K} .
2. σ does not begin with $I.A[i]$, with A multi-valued and i less than or equal to the number of instance statements on $I.A$ in \mathcal{K} .
3. σ does not contain $A.B$ or $A[j].B$ with B single-valued and B *inverse-of* A .
4. σ does not contain $A.B[1]$ or $A[j].B[1]$ with B multi-valued and B *inverse-of* A .

A *variable* is a standard indexed chain that ends in a simple attribute or a quantifier. A *basic variable* is a variable with no multi-valued attributes in it. ■

Note that the assumption that the instance statements on a multi-valued attribute are numbered allows us to specify exactly how the $I.A[i]$ are bound. Also, if B is a multi-valued attribute that is an inverse of A , and some entity c is the value of the complex standard chain $I.\sigma.A$ (where A is single-valued), there must be some i such that the value of $I.\sigma.A.B[i]$ is equal to $I.\sigma$. We stipulate that the indexed chains $I.\sigma.A.B[i]$ are numbered in such a way that it is always $I.\sigma.A.B[1]$ which is equal to $I.\sigma$. This is the reason behind condition 4: $I.\sigma.A.B[1]$ is not a standard chain, because its value is always equal to that of $I.\sigma$. Enforcing this numbering can always be done safely because only one value of $I.\sigma.A.B[i]$ can be equal to $I.\sigma$, by the unique names assumption. In addition, inverse and instance statements cannot apply simultaneously to the same entity, because of the assumption made in the previous chapter that all instance statements that are implied by inverse statements are made explicit.

Analogues of Lemmas 5.2.6 and 5.2.7 hold here: for every indexed chain X there is a unique standard indexed chain Y such that $[X]^\omega = [Y]^\omega$ in every possible world. The proofs are essentially the same as in the previous chapter. We can define the function θ that maps an indexed chain $X = I.\sigma$ to the standard indexed chain to which it is equal, as follows:

While σ has the form $\tau.A.B.\tau'$ or $\tau.A[i].B.\tau'$ or $\tau.A.B[1].\tau'$ or $\tau.A[i].B[1].\tau'$,

where B inverse-of A do:

$$\sigma = \tau.\tau'.$$

While $\sigma = A.\sigma'$, and \mathcal{K} contains a statement $I.A = J$

or $\sigma = A[i].\sigma'$, and the i -th instance statement on $I.A$ in \mathcal{K} is $J \in I.A$ do:

$$I = J.$$

$$\sigma = \sigma'.$$

Return $\sigma.I$.

The flat BN constructed relative to a set of basic variables may contain both basic variables and other variables. The reason is that the variables on which a quantifier depends are not basic. Specifically, the quantifier $Q = I.\sigma.\#[A.\rho = v]$ actually depends on the values of $\theta(I.\sigma.A[i].\rho)$, where i ranges from 1 to $\#[A]$. The CPF for Q is straightforward: it simply counts the number of times $I.\sigma.A[i].\rho$ takes on the value v . It is possible that the same actual variable will correspond multiple formal variables $I.\sigma.A[i].\rho$. For example, if \mathcal{K} contains the statements $J \in I.A$, $K \in I.A$, $J.B = I$ and $K.B = I$, then $\theta(I.A[1].B.C)$ and $\theta(I.A[2].B.C)$ are both equal to $I.C$. In that case, the actual parent needs to be counted multiple times.

The fact that the flat BN relative to a set of basic variables also contains non-basic variables is no matter. Non-basic variables are still well-defined terms that have a value in any possible world. The parents and CPF of non-basic variables are taken from the local probability model for B , just as with basic variables. If the global dependency model is acyclic, the flat BN relative to any set of variables constructed in this manner will be well-defined, and therefore there will be a well-defined probability distribution over the values of any finite set of basic variables. The fact used to prove Lemma 5.4.6, namely, that the set of ancestors of a basic variable is the same in any flat BN in which the variable appears, continues to hold, so the lemma remains true. All subsequent lemmas proved in section 5.4.3 continue to hold as well, as does Theorem 5.4.10. We can therefore be confident that the introduction of multi-valued attributes does not cause problems for the semantics of our language.

6.2.3 Inference with Quantifiers

While multi-valued attributes and quantifiers do not pose particularly interesting problems in the language definition and semantics, except in the technical details, they do present new challenges for the inference algorithm. As usual, we could use the technique of constructing the flat BN relative to a set of variables to answer any query on that set of variables, but we would rather handle multi-valued attributes in the structured framework of **SVE**.

One approach to dealing with multi-valued attributes in inference is simply to transform the model to one containing single-valued attributes, and then use the **SVE** algorithm from the previous chapter. Under this approach, a multi-valued attribute A is converted into $\#[A]$ single-valued attributes $A[1], \dots, A[\#[A]]$. For a quantifier $\#[A.\sigma = v]$, we create a new simple attribute corresponding to the quantifier, and set its parents to be $A[1].\sigma, \dots, A[\#[A]].\sigma$. The CPF for the quantifier is obvious: it counts up the number of its parents that have the required value v .

We can handle inverse statements on A very easily in this approach. If \mathcal{K} contains the statement A *inverse-of* B , we simply replace it with $A[1]$ *inverse-of* B . Instance statements can be handled similarly. Each instance statement $J \in I.A$ can be replaced by the statement $I.A[i] = J$, using the numbering on the instance statements on $I.A$.

This approach is very simple, and immediately allows us to use the structured algorithm from the previous chapter for models with multi-valued attributes. It automatically benefits from the reuse of computation for different instances of the same class, because all the $A[i]$ will share the same probability model, and the same query will be asked on all of them, so only one recursive computation needs to be performed for A . However, the approach suffers from a serious shortcoming. The number of parents of a quantifier $\#[A.\sigma = v]$ is equal to $\#[A]$. The size of its CPF is therefore exponential in $\#[A]$. Our experimental results in Chapter 8 show that this exponential blowup causes the approach to become infeasible as $\#[A]$ gets to 7 or 8.

We deal with this problem by exploiting symmetry. When computing the value of a quantifier, we do not care which of the fillers³ of the multi-valued attribute have

³The word “filler” here is a synonym for the value of a complex attribute. We use it to avoid having always to use the word “value” for the values of both simple and complex attributes, which

the value we are looking for, we only care how many of them have that value. We can therefore use a simple combinatoric calculation to express the probability distribution over the value of the quantifier in terms of the probability that a single filler has the required value.

In the following discussion, we will assume that an **SVE** computation is being performed on a class C , and that A is a multi-valued attribute of C with range type C' . To simplify the presentation, we will assume at first that the KB contains no instance or inverse statements. As a result, all fillers of A are generic instances of C' . In addition, the recursive computation on C' will return a probability distribution over $Needed[A]$, which is not conditioned on any inputs. The set $Needed[A]$ contains all chains $A.\sigma$ such that a quantifier $\#[A.\sigma = v]$ is processed during the **SVE** computation on C .

We will first deal with the case where there is only a single quantifier $Q = \#[A.\sigma = v]$ on A , so that $Needed[A]$ contains only a single element. The recursive call on A will return a probability distribution over the value of σ for a generic instance of C' . Let α be the computed probability that σ has the value v . By simple combinatorics, the probability that $Q = m$ is equal to the binomial term $\binom{\#[A]}{m} \alpha^m (1 - \alpha)^{(\#[A] - m)}$. We can easily compute the CPF for Q in time $O(\#[A]^2)$, using a recurrence relation. Let $P_m(Q = k)$ denote the probability that k out of the first m fillers of A have value v . Then $P_0(Q = 0) = 1$, and

$$P_{m+1}(Q = k) = (1 - \alpha)P_m(Q = k) + \alpha P_m(Q = k - 1). \quad (6.1)$$

The CPF for Q is then given by $P_{\#[A]}$.

Now consider the case where there are multiple quantifiers Q_1, \dots, Q_ℓ on A , where $Q_i = \#[A.\sigma_i = v_i]$. In this case, the set $Needed[A]$ is $\{\sigma_1, \dots, \sigma_\ell\}$, and the recursive call for A returns a joint probability distribution over the values of the σ_i for a single generic instance of C' . We will denote this joint distribution by P_A . The method of computing the CPFs for the quantifiers is a little more complicated now, but similar. Since, for a particular value of A , the σ_i may not be independent of each other, the

can be confusing.

different quantifiers are also not independent of each other, and we need to compute a joint distribution over all of them. Therefore, rather than computing a separate factor for each quantifier, we compute a single factor over all of them. We can encode the contribution of a single filler $A[i]$ of A to the quantifiers in a *contribution vector* of length ℓ , in which the j -th component is 1 if $A[i].\sigma_j = v_j$, and 0 otherwise. We can compute a probability distribution over the 2^ℓ possible contribution vectors from P_A as follows: For each assignment of values \mathbf{y} to the σ , let $\kappa(\mathbf{y})$ be the contribution vector in which the j -th component is 1 iff $y_j = v_j$. Then the probability of a contribution vector κ , denoted α_κ , is given by

$$\alpha_\kappa = \sum_{\{\mathbf{y} \in \text{Val}[\sigma] : \kappa(\mathbf{y}) = \kappa\}} P_A(\sigma = \mathbf{y}).$$

The value of α_κ for all κ can be computed in time proportional to the size of P_A , which is exponential in ℓ .

Like the contribution vectors, each joint value of the quantifier variables can be encoded by a vector of ℓ components, where each component is between 0 and $\#[A]$. We can use a similar recurrence relation to Equation 6.1 to compute the joint distribution over the values of all the quantifiers. Again, $P_0(\mathbf{Q} = \mathbf{0}) = 1$, and

$$P_{m+1}(\mathbf{Q} = \mathbf{n}) = \sum_{\substack{\kappa \in \{0,1\}^\ell, \mathbf{n}' \in \{0, \dots, m\}^\ell \\ \mathbf{n} = \kappa + \mathbf{n}'}} \alpha_\kappa \cdot P_m(\mathbf{Q} = \mathbf{n}') \quad (6.2)$$

We need to compute $O((\#[A] + 1)^{\ell+1})$ summations, and each one involves $O(2^\ell)$ terms, so the time to compute the joint distribution over the quantifiers is $O((2(\#[A] + 1))^{\ell+1})$.

Now let us complicate matters a little further, by introducing inverse statements. If \mathcal{K} contains a statement B *inverse-of* A , then the recursive **SVE** call for A will return a set of inputs τ , and P_A will be a conditional probability distribution over σ given τ . We now need to repeat the above computation for each of the possible values \mathbf{u} of τ . Each such value will define a different distribution $\alpha^{\mathbf{u}}$ over the contribution

vectors, which we can then use in Equation 6.2. The result of the computation will be a factor over $\tau \cup Q$, expressing the conditional distribution over the values of the quantifiers given the values of the inputs.

Now suppose that A is itself an inverse of some other attribute D , and that D was actually the fifth argument of the **SVE** call on C . So the first filler of A is actually the object that generated the **SVE** call on C . The probability distribution over the values of the quantifiers will be conditioned on the value w of the inputs received from the calling object. The multi-valued attribute A will have $\#[A] - 1$ other, generic fillers, in addition to the calling object, and a recursive call will still be made for A to compute a distribution P_A over σ for these generic fillers. For each value w of the inputs received from the calling object, the probability distribution over the quantifiers can be computed using P_A , in a very similar manner to the computation above. The only difference is that now P_A is only used to determine the contributions of the $\#[A] - 1$ generic values of A . In fact, this situation can be handled with only a very small change to the recurrence relations. Instead of initializing with the equation $P_0(Q = \mathbf{0}) = 1$, we initialize with the given value w of the inputs, and set $P_1(Q = \kappa(w)) = 1$. The contributions of the generic fillers are then handled using Equation 6.2. If P_A also has inputs τ , this whole process will have to be repeated for all values in $Val[\tau]$, as described in the previous paragraph. The result of this computation will be a factor over $\sigma \cup \tau \cup Q$, expressing a conditional distribution over the values of the quantifiers given both the input values from the calling object and the inputs of A as determined by the recursive call for A .

Finally, we need to consider multi-valued attributes and quantifiers in the top-level object used for performing inference about named instances. Suppose $[I.A]$ is a top-level multi-valued attribute. Instance statements binding some of the values of $I.A$ may apply. If there are m such instance statements, each of the form $J_k \in I.A$, the values of the quantifiers will depend on the values of the $[J_k.A].\sigma$. In addition, there will be $\#[A] - m$ generic fillers of $[I.A]$. We treat this case in a similar manner to the previous paragraph. For a particular set of m values y^1, \dots, y^m for $[J_1.A].\sigma, \dots, [J_m.A].\sigma$, we initialize the recurrence equations with $P_m(Q = \sum_{k=1}^m \kappa(y^k)) = 1$, and use Equation 6.2 to compute contributions from the generic fillers. The result will be a factor

over $[J_1.A].\sigma \cup \dots \cup [J_m.A].\sigma \cup Q$. The total number of variables mentioned by this factor is $(m + 1)\ell$, so the size of the factor is exponential in both the number of quantifiers and in the number of instance statements on $I.A$.

Despite the exponential dependence on ℓ and m , this approach is still much better than the original approach of replacing A with $\#[A]$ single-valued attributes. The exponential dependence on ℓ is unavoidable in both approaches, because the size of the factor returned by the recursive call for A is already exponential in ℓ . The original approach also had an exponential dependence on $\#[A]$. For a KB with few instance statements, m will typically be much smaller than $\#[A]$, so the second approach that exploits symmetry is much better. In fact, our experiments in Chapter 8 show that in the absence of instance statements, increasing $\#[A]$ has very little impact on the cost of inference. Whether or not a KB contains few or many instance statements will vary from application to application. In fact, in the next section we will consider models in which every instance is named, and there are no generic instances. For such models, there is no symmetry to be exploited, and consequently no advantage to using the combinatoric approach.

6.3 Closed World Models

We now turn to another language that allows multi-valued attributes, in which the structure of the world is still fully known. This language allows multi-valued attributes to take on any number of values. Full knowledge of the relational structure is maintained by requiring that in all possible worlds, the domain entities are the explicitly named instances, and that all relationships between entities are explicitly listed in the KB. We call such models *closed world models*, because they satisfy the *closed world assumption*: for any possible world ω , an entity c is in Δ^ω if and only if $c = [I]^\omega$ for some named instance, and $[A]^\omega([I]^\omega, [J]^\omega)$ holds if and only if the KB contains an instance statement $I.A = J$ or $J \in I.A$.

Closed world models do not require any restrictions on the relational language, but they do make some restrictions on the statements in a KB. If A is a single-valued complex attribute with domain type C , then for every named instance I whose type is

a subtype of C , the KB must contain exactly one instance statement on $I.A$. Clearly there can be no more than one such statement, and the lack of such a statement would imply that the value of $I.A$ is some unnamed entity, contrary to the closed world assumption.

Also, inverse statements have no effect in a closed world model. As we have commented previously, inverse statements in conjunction with instance statements can cause other statements about named instances to hold, but we assume that all such implied statements will be made explicitly. Therefore, the only real effect of an inverse statement is to bind the values of attributes of unnamed entities. Since a closed world model has no unnamed entities, inverse statements have no effect.

The class probability models are defined as in the previous section, using quantifiers to model a dependency on the fillers of a multi-valued attribute. The range of the quantifier $\#[A.\sigma = v]$ depends on the number of values of A , which may vary from instance to instance. If \mathcal{K} contains m instance statements on $I.A$, then the quantifier can take on values from 0 to m . However, when defining the class probability models, we need to take into account all possible values of m . We therefore set the range of the quantifier to be $\{0, 1, \dots, \max \#[A]\}$, where $\max \#[A]$ is an upper bound on the number of values of A .

We may want the probability model for a simple attribute to depend not just on the number of related objects that have a certain value, but on the fraction of related objects that have that value, or on whether they all have that value, and so on. In the language of the previous section, we could derive these values directly from a quantifier and the fixed value of $\#[A]$. Now, the number of values of A may vary, and we may want to allow the properties of a simple attribute to depend on it. We therefore introduce $\#[A]$ as an attribute of the domain type of A . For any instance I , $I.\#[A]$ is equal to the number of instance statements on $I.A$. The range of $\#[A]$ is $\{0, 1, \dots, \max \#[A]\}$. For the purpose of creating attribute chains that can be used as the parents of simple attributes, this $\#[A]$ attribute is treated like a simple attribute. However, it does not have its own probability model, and the value of $I.\#[A]$ is always known.

The probabilistic semantics for closed world models is very simple. Since the only

entities in the domain correspond to named instances, and there are only finitely many named instances, the entire state of the world can be captured through a finite set of basic variables, namely, the values of simple attributes of the named instances. We can therefore define a probability distribution over the possible worlds using a flat BN over the basic variables.

This flat BN contains a node $I.A$ for every simple attribute A of a named instance I . In addition, if a class probability model uses a quantifier $Q = \#[A.\sigma.B = v]$, it contains a node $I.Q$ for every instance I of C . If there are k instance statements $I.A = J_1, \dots, I.A = J_k$, $I.A.\sigma$ has k values K_1, \dots, K_k , where $K_i = J_i.\sigma$.⁴ The quantifier Q therefore depends on the nodes $K_1.B, \dots, K_k.B$, and its CPF is defined in the obvious way.⁵ The flat BN also contains a node $I.\#[A]$ for each multi-valued attribute $\#[A]$ of the type of I . Since the value of $I.\#[A]$ is always observed, its CPF is immaterial.

Let \mathbf{X} denote the nodes corresponding to simple attributes of named instances, \mathbf{Y} denote the nodes corresponding to quantifiers, and \mathbf{Z} denote the nodes corresponding to the $\#[A]$. The values of \mathbf{Y} are fully determined by the values of \mathbf{X} , while the \mathbf{Z} have the same value \mathbf{z} in every possible world. There is a one-to-one correspondence between values of the \mathbf{X} and possible worlds in $\Omega_{\mathcal{K}}$. The flat BN as constructed here defines a joint probability distribution P over the \mathbf{X} , \mathbf{Y} and \mathbf{Z} . We define the semantics of a closed world model \mathcal{K} to be the distribution $P_{\mathcal{K}}$ over $\Omega_{\mathcal{K}}$ given by $P_{\mathcal{K}}(\omega) = P(\mathbf{X} = \mathbf{x} \mid \mathbf{Z} = \mathbf{z})$, where \mathbf{x} is the value of \mathbf{X} in ω .

Since there are no generic instances of a class in a closed world model, all inference happens in the top-level object. There are no recursive calls of the **SVE** algorithm. The algorithm in fact performs exactly the same computations as **VE** in the flat BN. There is no exploitation of the object structure of the model. A knowledge base with many interconnected named instances can lead to a horribly multi-connected BN over the attributes of the named instances. This is the case even if the class models are very simple, as shown by the following example.

⁴Recall that in a quantifier $\#[\sigma = v]$, all but the first attribute of σ is single-valued.

⁵Some of the K_i may be the same, due to instance statements, but we saw how to deal with that situation in the previous section.

Example 6.3.1: Consider a model used to predict the outcomes of sporting events. There are two classes: a **Team** class, with simple attribute **Quality**, and a **Game** class, with complex attributes **Home-Team** and **Away-Team**, both of type **Team**, and a simple attribute **Outcome**. The class probability models are very simple. In the **Team** probability model, **Quality** has no parents, and a prior probability distribution is specified over the **Quality**. In the **Game** model, **Outcome** is made to depend probabilistically on **Home-Team.Quality** and **Away-Team.Quality**.

An instantiation of this model may contain a **Team** instance for every team in a league, and a **Game** instance for every game played over the course of a season. The purpose of the model is to determine the qualities of the teams based on the outcomes of some games, and use those qualities to predict the outcomes of other games. Consider the flat BN for this model. If every team plays every other team, there will be an edge between every pair of teams in the moral graph for this BN. Thus the set of teams forms a clique of the moral graph, which is a subgraph of the induced graph for any variable elimination order. The tree-width for this network, which is the induced width for the best elimination order, is therefore at least the number of teams. The cost of inference, even for the best elimination order, is at least exponential in the number of teams. ■

Using a structured inference algorithm will not help in this case. As we discussed in Section 4.5.5, the computations performed by **SVE** are the same as the computations performed by standard **VE** in the flat BN, for some elimination order. The tree-width of the flat BN provides a lower bound on the cost of inference for any elimination order. Even if we found a clever way to exploit structure in the top-level object, we would still be bound by the tree-width, and inference would be infeasible even for a moderately sized league.

Example 6.3.1 illustrates that it is the interconnectedness of the relational model, rather than a complex set of class probability models, that causes inference to become intractable. While one can encourage the model designer to build simple class models, one cannot really avoid interconnected relational models, since they reflect the given structure of the domain. So the problem cannot be circumvented simply through model design.

Our only hope to perform inference in these highly connected models is to use an approximate inference algorithm. Sampling methods, such as Markov Chain Monte Carlo (MCMC) methods [72] are always a possible candidate for approximate inference, but their performance tends to vary from application to application. It would be interesting to see how well these methods perform here.

Variational methods [51] have performed very well where applicable. There is reason to believe that they might be applicable to the types of networks produced by relational probability models with many instances. The network in Example 6.3.1 is a *two-layered network*. The first layer consists of **Team.Quality** variables, each of which has no parents, and the second layer consists of **Game.Outcome** variables, each of which has two parents from the first layer. Networks for richer models will exhibit a similar layered nature, but in which the layers themselves are organized into a graph. Also, quantifier nodes exhibit a form of *causal independence* [39]. Causal independence characterizes the situation in which the different causes of an effect act independently of each other to produce the effect. A common example is the *noisy-or* model of causal influence [79]. Variational methods have proven to work very well for two-layer networks with noisy-or nodes, the so-called BN20 networks [47]. Since the networks produced by RPMs share some properties of BN20 networks, there is reason to hope that variational methods might also work for them.

It is an open question whether the object structure of the domain can be used to facilitate approximate inference. It is possible, for example, that some sort of localized MCMC sampling scheme that use the object structure will perform better than performing MCMC on the entire model. It is also possible that the structured representation can suggest new structure-based approximation algorithms. This topic is worthy of future investigation.

6.4 Number Uncertainty

In the previous two sections, we have extended our language to include multi-valued attributes in such a way that the number of fillers is always known. However, once we have multi-valued attributes in the language, it is natural to consider the case where

we have uncertainty as to the number of fillers. This is an example of *structural uncertainty* called *number uncertainty*. The name “structural uncertainty” arises from the fact that we have uncertainty over the relational structure of the possible worlds. Up till now, for any KB \mathcal{K} , the interpretation of complex attributes was the same in all possible worlds in $\Omega_{\mathcal{K}}$. This is longer the case when we allow uncertainty over the number of values of a multi-valued attribute.

6.4.1 Possible Worlds

We deal with number uncertainty over the number of values of A using the same $\#[A]$ quantities as in the previous two sections. Now, however, $\#[A]$ will be treated as a simple attribute, with its own local probability model. The $\#[A]$ attributes are called *number attributes*. The value of $I.\#[A]$ may vary in different possible worlds, but it does impose a constraint on the possible worlds, namely, that the number of values of $I.A$ should be equal to $I.\#[A]$. Formally, we make the following definitions.

Definition 6.4.1: A *typed relational language with number uncertainty* is a typed relational language $\langle \mathbf{C}, \sqsubseteq, \mathbf{A}, \mathbf{f}, \mathbf{R}, \mathbf{I} \rangle$, such that for each $R \in \mathbf{R}$, with domain type C , there is an associated simple attribute $\#[A]$ with domain type C , whose range is a set of integers $\{0, 1, \dots, \max \#[A]\}$.

Let \mathcal{K} be a KB with number uncertainty. A *possible world* for \mathcal{K} is an interpretation ω of \mathcal{L} such that:

1. For every relation R with domain type C , and every $c \in [C]^\omega$, $|\{d : [R]^\omega(c, d)\}| = [\#[R]]^\omega(c)$. A possible world that satisfies this property for a particular R is said to *respect* $\#[R]$.
- 2–6. Same as in Definition 6.2.1: instance and inverse statments, unique names assumption, and no proper subworlds assumption must hold. ■

Unlike previously, the set of entities in a possible world and the ways they are connected to each other are not fixed. There is no set of simple attributes whose values fully characterize the state of the world. We cannot prove exact analogues

of Theorems 4.2.11 and 5.2.8 that completely characterize the structure of possible worlds. However, we can define the maximal possible world structure, which is the structure of the world if all multi-valued attributes have their maximum number of values. We then get a structure like the one of Section 6.2. Indexed chains, standard indexed chains, variables and basic variables are defined as in Definition 6.2.3. A complex standard indexed chain denotes an entity that *could* exist in a possible world. The θ function is defined as in Section 6.2. It has the property that for any standard chain $I.\sigma$, if $[I.\sigma]^\omega$ exists, so does $[\theta(I.\sigma)]^\omega$, and they have the same value.

Definition 6.4.2: A *possible entity* is a complex standard indexed chain. The set of all possible entities for a KB \mathcal{K} will be denoted $\Sigma_{\mathcal{K}}$, or simply Σ . ■

It is convenient to define the $[]^\omega$ function on indexed chains to be the interpretation of an entity if it exists, null otherwise. Formally,

Definition 6.4.3: Let \mathcal{K} be a KB with number uncertainty, and ω a possible world for \mathcal{K} . The function $[]^\omega$ on indexed chains is defined by the following rules:

1. $[I.\epsilon]^\omega = [I]^\omega$.
2. If $[I.\sigma]^\omega = c$, and A is a single-valued attribute, $[I.\sigma.A]^\omega = [A]^\omega(c)$.⁶
3. If $[I.\sigma]^\omega = c$, and A is a multi-valued attribute, let $[A]^\omega(c) = \{d_1, \dots, d_m\}$, where $m = [\#[A]]^\omega(c)$.⁷ Then for $1 \leq i \leq m$, $[I.\sigma.A[i]]^\omega = d_i$. For $m < i \leq \max \#[A]$, $[I.\sigma.A[i]]^\omega = \perp$.⁸
4. If $[I.\sigma]^\omega = \perp$, $[I.\sigma.\rho]^\omega = \perp$.

If $[I.\sigma]^\omega \neq \perp$, we say that $I.\sigma$ *exists* in ω .

⁶As a result of the first two rules, the definition of $[]^\omega$ given here is the same as the standard one for single-valued chains.

⁷We assume that the d_i are ordered in such a way that if σ is empty, the value of the j -th instance statement on $I.A$ is d_j , while if σ is non-empty and A is the inverse of the last attribute of σ , d_1 is the value bound by the inverse statement.

⁸ \perp is an element not in Δ^ω signifying “undefined”.

Lemma 6.4.4: *If two distinct possible entities $I.\sigma$ and $J.\rho$ both exist in ω , then $[I.\sigma]^\omega \neq [J.\rho]^\omega$.*

Proof: Same as for Lemma 5.2.7, using the unique names assumption. ■

Theorem 6.4.5: *Let \mathcal{K} be a KB with number uncertainty, and ω a possible world for \mathcal{K} . There exists a map ϕ from Δ^ω into Σ such that for every $c \in \Delta^\omega$, $[\phi(c)]^\omega = c$.*

Proof: Let S be the subset of Σ consisting of the possible entities that exist in ω . Following the proof of Theorem 4.2.11, we use the no-proper-subworlds condition to show that the image of S under $[\]^\omega$ is Δ^ω . By Lemma 6.4.4, the $[\]^\omega$ function from S to Δ^ω is one-to-one, so $[\]^\omega$ is a one-to-one correspondence between S and Δ^ω , and it has an inverse ϕ satisfying $[\phi(c)]^\omega = c$. ■

Note that by Definition 6.4.3, the set S of possible entities that exist in ω is fully determined by the $[\#[A]]^\omega$. Accordingly, we could define a probabilistic model for a KB with number uncertainty by defining a probability model over the values of the number attributes, and then proceeding as in Section 6.2 to define a probability model for the remaining attributes. We would specify, for each multi-valued attribute A , a probability distribution over $\#[A]$. Following this approach, the probabilistic semantics would be defined very simply in two stages. The probability model for number attributes would define a probability distribution over the set of possible entities in the world, which would determine its structure. For each structure, we would then have a probability distribution over the values of the remaining simple attributes as described in Section 6.2. This is essentially the approach we took in [54].

However, if we are going to design a probabilistic language that allows us to express uncertainty over the relational structure of the domain, we might as well integrate that uncertainty into our existing probabilistic language. Doing so has the powerful effect of allowing aspects of the model structure to depend on and influence the values of simple attributes.

Example 6.4.6: For example, consider a model of the **Woman** class. Suppose the class has a multi-valued attribute **Children**. Uncertainty over the number of children is

expressed through the probability model over $\#[\text{Children}]$. It is natural for $\#[\text{Children}]$ to depend probabilistically on the attribute **Age**. Also, if the **Woman** class has an **Available-Time** attribute, it is natural for that to depend on $\#[\text{Children}]$. ■

The desired effect can be achieved by treating a number attribute as a full-fledged simple attribute in the class probability models. It has its own set of parents and CPF, and it can also appear as the parent of some other attribute. In the example above, $\#[\text{Children}]$ is treated in exactly this way. It has a parent **Age**, and the CPF for $\#[\text{Children}]$ defines a different probability distribution over the number of children for different values of **Age**. Also, one of the parents of **Available-Time** is $\#[\text{Children}]$.

Introducing number attributes directly into the probability models requires that we make sure that no cycles are introduced. The value of a number attribute of an entity determines part of the model structure, namely, the number of values of the associated multi-valued attribute for that particular entity. The values of other variables may depend on that aspect of model structure. In particular, the value of a quantifier on the multi-valued attribute depends on the number of entities that are contributing to the quantifier. We therefore add the following edges to the dependency graphs to deal with number attributes: if C has a multi-valued attribute A , we add an edge from $\#[A]$ to A in $\mathcal{G}[C]$, an edge from $C.\#[A]$ to $C.A$ in $\mathcal{G}[\mathcal{K}]$, and an edge from $I.\#[A]$ to $I.A$ for each instance I of type C .

As it happens, we also need another, rather technical, condition. For reasons that will become clear later, we need to prevent the following type of situation: A is a multi-valued attribute of C , that happens to be the inverse of some attribute, and there is a chain ρ , such that for an entity $I.\sigma$ of type C , $I.\sigma.\#[A]$ and $I.\sigma.\rho.\#[A]$ have a common ancestor. Such a circumstance will produce an infinite “sideways” chain of dependencies:

$$I.\sigma.\#[A] \leftarrow U_1 \rightarrow I.\sigma.\rho.\#[A] \leftarrow U_2 \rightarrow I.\sigma.\rho.\rho.\#[A] \dots$$

As we shall see, a sideways chain like this one can cause trouble, but only if A is an inverse attribute. To prevent this situation from occurring, we stipulate that if A is an inverse attribute, $\#[A]$ must precede all complex attributes in $\mathcal{G}[C]$.

6.4.2 Semantics

Even though the structure of the world is not known, and we are allowing aspects of the structure to depend probabilistically on values of other attributes, we already have most of the tools needed to define the probabilistic semantics for this integrated language. As we did previously, we will define a probability measure over the set of possible worlds. Our event space will again be generated by basic events, which consist of assignments of values to a finite number of basic variables. The basic variables will now include variables of the form $I.\sigma.\#[A]$, by virtue of the number attributes being treated just like other simple attributes. A basic variable of this form is called a *number variable*. As before, we define a probability for each basic event using a local BN, and combine these together to create a probability measure over the possible worlds.

One might think that defining basic events in this way could now produce an incoherent model. For example, suppose we could formulate the sentence “ $I.A$ has three fillers, and the fourth filler of $I.A$ has value x for B ” as a basic event. It is not even clear what probability this event should have. One could interpret the second half of the sentence as implying that $I.A$ has at least four values, so the event should have probability zero. On the other hand, perhaps the second half of the sentence is a counterfactual statement. If $I.A$ had had at least four fillers in ω , the fourth filler would have had a specific value in ω . We can ask for the probability that the fourth value of $I.A$ would have had certain properties, had $I.A$ had at least four values. Using our approach of defining the probabilities of basic events using local BNs, it would be much easier to enforce the second interpretation, but it is far from clear that that is the right thing to do.

Fortunately, we do not have to worry about such matters. We defined the event space so that the only statements that can be expressed about the values of multi-valued attributes are quantified statements. In our language, $I.A[4].B$ is not a basic variable, but $I.\#[A.B = v]$ is. Certainly, the value of $I.\#[A.B = v]$ depends on the number of fillers of $I.A$. However, there is no ambiguity to the sentence “ $I.A$ has three fillers, and four fillers of $I.A$ have value v for B ”, and the probability of the basic event $[I.\#[A] = 3, I.\#[A.B = v] = 4]$ should clearly be zero. We can easily

enforce this in a local BN by making the CPF for the quantifier depend on the number variable.

A key issue that must be dealt with in defining the semantics is that a KB may encode implicit constraints on the values of number variables. We must make sure that the probability measure over possible worlds defined by our semantics assigns probability zero to worlds that violate the constraints. The constraints arise due to instance and inverse statements. If there is an instance I with multi-valued attribute A , and m instance statements on $I.A$, then we know that in all possible worlds $[I.\#[A]]^\omega \geq m$. Meanwhile, if B *inverse-of* A , then we know that $I.\sigma.A.\#[B] \geq 1$.⁹ We can summarize by defining $I.\min\#[A]$ to be the number of instance statements on $I.A$, and $I.\sigma.A.\min\#[B]$ to be 1 if B *inverse-of* A , 0 otherwise. We know that $I.\sigma.\#[A] \geq I.\sigma.\min\#[A]$ must hold in every possible world.

Since the number variables are included in our probability model, we must design the model in such a way that a possible world that violates these constraints has probability zero. We can achieve this effect in one of two ways. One way is to change the model for the relevant number attributes to enforce these conditions. This process is sometimes known as *surgery* or *intervention* [80]. An intervention changes the probability model for the affected variable, and, as a result, the probability distributions over all variables that depend on the affected variable are changed. Note that since we are directly intervening in the model for the affected variable, its dependence on other variables is circumvented, so the distribution over the affected variable's parents is not changed as a result of the intervention. In our framework, the desired result can be achieved by changing the CPF of a variable $X = I.\sigma.\#[A]$, so that for each set of values \mathbf{u} of the parents, $CPF_X(k \mid \mathbf{u}) = 0$ for $k < I.\sigma.\min\#[A]$. Each CPF row is then normalized so that it sums to 1.

An alternative way to enforce the constraints is *conditioning*. In this approach, the CPFs of the affected variables are left unchanged, but the entire probability distribution defined by the BN is changed by observing that the constraints hold. The constraints serve not only to change the distribution over the affected variable and its descendants, but also act as evidence, changing the distribution over the

⁹Fortunately, the two conditions cannot both apply to the same entity.

parents of the affected variable.

We argue that conditioning is the right approach in our framework. This is despite the fact that the model intervention approach would be easier to implement, since we could use the same flat BN we have used all along and just change the CPFs of the number variables that appear in it. The reason that conditioning is better should be clear from the following example.

Example 6.4.7: Consider again the **Woman** class from Example 6.4.6. Let us add a **Mother** attribute to the class, and declare that **Children** *inverse-of* **Mother**. Now, let I be an instance of **Woman**, and let J be equal to $I.$ **Mother**. The inverse statement constrains $J.\#[\text{Children}]$ to be at least one. Since $J.$ **Available-Time** depends on $J.\#[\text{Children}]$, the probability distribution over $J.$ **Available-Time** should be affected by the constraint that $J.\#[\text{Children}] \geq 1$. Intuitively, if I know that a particular woman is a mother, then I will believe that she is less likely to have much free time. This will in fact be the case using both the intervention and conditioning approaches.

It is also intuitively clear that $J.$ **Age** should be affected by the constraint. My distribution over the age of a mother will provide a higher likelihood of her being an adult, than my distribution over the age of a woman about whom I know nothing. This effect, however, is achieved only by the conditioning approach, and not by model intervention. ■

We now show how to construct the flat BN relative to a set of basic variables \mathbf{X} . Two things are of note here. First, the BN contains nodes for the simple attributes of all possible entities that *may* influence the value of a quantifier, even though some of these possible entities do not exist in some possible worlds. For example, if $\max \#[A] = 2$, then $I.\#[A.B = v]$ depends on $I.A[1].B$ and $I.A[2].B$, as well as on $I.\#[A]$, even though the entity $I.A[2]$ does not exist if $I.\#[A] < 2$. The number variable controls which of the possible entities actually have an influence on the quantifier. In this example, if $I.\#[A] < 2$, the CPF for $I.\#[A.B = v]$ does not depend on the value of $I.A[2].B$. This is an example of *context-specific-independence* [12]. The variable $I.A[2].B$ is only relevant to $I.\#[A.B = v]$ in the context $I.\#[A] = 2$.

The second issue is that in order to enforce the correct semantics of conditioning

on the constraints on number variables, the flat BN must contain not only the variables \mathbf{X} and their ancestors, but also the number variables that can condition the variables in the network. We say that a number variable $I.\sigma.\#[A]$ is *conditionable* if $I.\sigma.\min \#[A] > 0$. A number variable is relevant to the variables in a BN if it is conditionable, and it is a descendant of one of the variables in the BN. Fortunately, as we shall see, there is a limited number of number variables that are relevant to any set of variables.

Definition 6.4.8: Let \mathbf{X} be a set of basic variables. The *ancestor set* \mathbf{X}^- of \mathbf{X} is defined as follows:

1. \mathbf{X}^- contains all nodes in \mathbf{X} .
2. If \mathbf{X}^- contains $I.\sigma.A$, \mathbf{X}^- contains $\theta(I.\sigma.v)$ for every parent v in the probability model of A .
3. If \mathbf{X}^- contains $I.\sigma.\#[A.\rho = v]$, \mathbf{X}^- contains $\#[I.\sigma.A]$, and $\theta(I.\sigma.A[i].\rho)$ for $1 \leq i \leq \max \#[A]$.

A number variable Y is *relevant to* \mathbf{X} if Y is conditionable, and Y has some ancestor in \mathbf{X} . ■

Fortunately, not all number variables in a model can be relevant to \mathbf{X} . In fact, only a number variable on a named instance, or on an instance that is a prefix of some $X \in \mathbf{X}$, can be relevant. As a result, there can only be finitely many relevant number variables, and we can find those by examining the candidates systematically.

Lemma 6.4.9: Let \mathbf{X} be a set of variables, and Y a number variable relevant to \mathbf{X} . Then Y is either equal to $I.\#[A]$ for some named instance I , or to $I.\sigma.\#[A]$, where $I.\sigma.B$ is a variable in \mathbf{X} .

Proof: Suppose Y is a relevant number variable, but not directly attached to a named instance. Because Y is conditionable, it must be equal to $I.\sigma.D.\#[A]$, with A inverse-of D . Let C be the range type of D . By the technical condition on the dependency graph, $\#[A]$ must precede every complex attribute of C in $\mathcal{G}[C]$. Therefore,

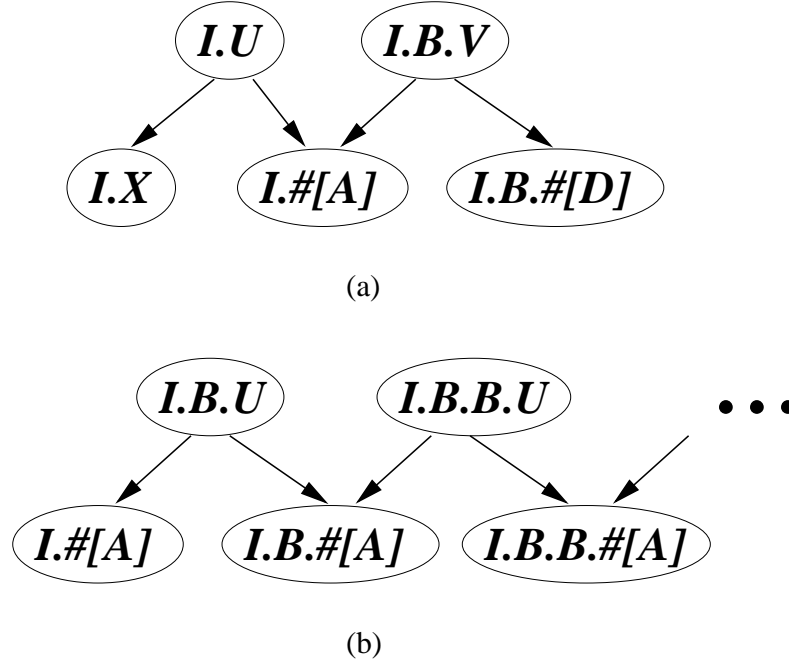


Figure 6.1: Relevant number variables.

every ancestor of $I.\sigma.D.\#[A]$ must be equal to $I.\sigma.D.B$ for some simple attribute B of C . In particular, since Y is relevant to \mathbf{X} , \mathbf{X} must contain a variable of this form.

■

Definition 6.4.10: The *complete set of relevant variables for \mathbf{X}* is defined by the following equations:

$$\begin{aligned}
 \mathbf{X}^0 &= \text{the ancestor set of } \mathbf{X}. \\
 \mathbf{Y}^n &= \text{the set of number variables relevant to } \mathbf{X}^n. \\
 \mathbf{X}^{n+1} &= \mathbf{X}^n \cup \text{the ancestor set of } \mathbf{Y}^n.
 \end{aligned}$$

The complete set of relevant variables for \mathbf{X} , written \mathbf{X}^* , is then $\cup_{i=0}^{\infty} \mathbf{X}^i$. ■

Why do we need an iterative process to compute the complete set of relevant variables? The reason is illustrated in Figure 6.1 (a). In this example, \mathbf{X} consists of the single variable $I.X$. This variable has a single ancestor $I.U$, so \mathbf{X}^0 is $\{I.X, I.U\}$.

Suppose that the number variable $I.\#[A]$ is conditionable, and depends on $I.U$. It then needs to be added to the relevant set of variables, because conditioning $I.\#[A]$ affects the distribution over $I.U$, which in turn affects the distribution over $I.X$. Since I is a named instance, and $I.\#[A]$ is conditionable, $I.\#[A]$ will be placed in \mathbf{Y}^0 and added to the set of variables. Since $I.\#[A]$ depends on $I.B.V$, the ancestor set of \mathbf{Y}^0 is $\{I.\#[A], I.B.V\}$, and \mathbf{X}^1 is $\{I.X, I.U, I.\#[A], I.B.V\}$. Now, suppose that $I.B.\#[D]$ is conditionable, and depends on $I.B.V$. Note that $I.B.\#[D]$ is *not* d-separated from $I.X$ by the conditioned variable $I.\#[A]$, because the path between them has converging arrows at $I.\#[A]$. Therefore, conditioning $I.B.\#[D]$ will affect the distribution over $I.X$, so $I.B.\#[D]$ needs to be added to the set of relevant variables. In fact, $I.B.\#[D]$ will be in \mathbf{Y}^1 . This “sideways” chain of relevant variables could be carried on further, with $I.B.\#[D]$ sharing an ancestor with some other conditionable number variable. The process of constructing the set of relevant variables is therefore iterative, to capture all number variables that can be reached in this way. This set is indeed sufficient:

Lemma 6.4.11: *Let \mathbf{X}^* be the complete set of relevant variables for \mathbf{X} , and let Y be a conditionable number variable not in \mathbf{X}^* . Let \mathcal{B} be a flat BN relative to some set of variables \mathbf{Z} , containing \mathbf{X} and Y . Then Y is d-separated from \mathbf{X} in \mathcal{B} by any subset of \mathbf{X}^* .*

Proof: It is clear that any number variable relevant to \mathbf{X}^* is in \mathbf{X}^* . For suppose number variable Y is relevant to $X \in \mathbf{X}^*$. Since $X \in \mathbf{X}^n$ for some n , $Y \in \mathbf{X}^{n+1} \subseteq \mathbf{X}^*$.

Therefore, if Y is a conditionable number variable not in \mathbf{X}^* , it cannot be the descendant of a variable in \mathbf{X}^* . Neither can it be an ancestor of a variable in \mathbf{X}^* . So a path between Y and \mathbf{X} must have converging arrows at a common descendant Z of Y and \mathbf{X}^* . Neither Z nor any of its descendants can be in \mathbf{X}^* , so the path is blocked by any subset of \mathbf{X}^* . ■

By Lemma 6.4.9, we can construct \mathbf{X}^{i+1} from \mathbf{X}^i by considering only conditionable number attributes on named instances or on prefixes of variables in \mathbf{X} . Thus the number of relevant number variables added at any stage is finite. In order to show

that Definition 6.4.13 makes sense, we must show in addition that the construction process terminates — that is, there is some number N such that for all $m > N$, $\mathbf{X}^m = \mathbf{X}^N$. If this is the case, the complete set of relevant variables is finite, and the BN is well defined. Figure 6.1 (b) shows the type of situation that we need to prevent: an infinite sideways chain of dependencies. If $I.B.\#[A]$ and $I.B.B.\#[A]$ share an ancestor $I.B.B.U$, then $I.B^k.\#[A]$ and $I.B^{k+1}.A$ will both share the ancestor $I.B^{k+1}.U$. As a result, we will have an infinite sideways chain of conditionable variables $I.B.\#[A], I.B.B.\#[A], I.B.B.B.\#[A], \dots$. In fact, preventing this type of situation was the motivation for the technical constraint we made on the dependency graph.

Lemma 6.4.12: *If $\mathcal{G}[\mathcal{K}]$ is acyclic, then there exists an integer N , such that for all $m \geq N$, $\mathbf{X}^m = \mathbf{X}^N$.*

Proof: We show that the length of chains σ such that $I.\sigma \in \mathbf{X}^*$ is bounded. A variable $X \in \mathbf{X}^*$ must be in \mathbf{X}^0 , or an ancestor of a conditionable number variable on a named instance, or an ancestor of a conditionable number variable on an unnamed instance. It is clear that the number of variables of the first two kinds is finite. Let ℓ be an upper bound on the lengths of the chains for those two kinds. We show by induction that no variable $I.\sigma$ with length of σ greater than ℓ is in \mathbf{X}^i . No variable not of the first kind is in \mathbf{X}^0 , by definition, and ℓ is an upper bound on the lengths of the chains for those variables. Suppose a variable X is in $\mathbf{X}^{i+1} - \mathbf{X}^i$. Then it must be an ancestor of a conditionable number variable that is the descendant of a variable in \mathbf{X} . If X is a variable of the second kind, the length of its chain is bounded by ℓ . If X is a variable of the third kind, let $Y = I.\sigma.\#[A]$ be the common descendant of X and \mathbf{X}^i . Since Y is conditionable, A must be the inverse of the last attribute in σ . So, the technical condition that $\#[A]$ must precede every complex attribute in $\mathcal{G}[C]$ holds (C being the range type of σ). Therefore, every ancestor of Y must be a simple variable on $I.\sigma$. This applies in particular to X and to the ancestor of Y in \mathbf{X}^i . Therefore X has the same chain length as a variable in \mathbf{X}^i . It follows that if the length of chains of variables in \mathbf{X}^i is bounded by ℓ , the same is true for \mathbf{X}^{i+1} , and for \mathbf{X}^* by induction.

Since there are only finitely many variables with chain length at most ℓ , there must be some N for which $\mathbf{X}^{N+1} = \mathbf{X}^N$. But then for all $m > N$, inductively, $\mathbf{X}^m = \mathbf{X}^N$, as required. ■

We can now proceed as in the previous chapter, defining the flat BN relevant to a set of variables *vara*, and using that to define a function f over basic events, specifying the probability of a basic event $[\mathbf{X} = \mathbf{x}]$ using $\mathcal{B}_{\mathbf{X}}$. Formally, we proceed as follows.

Definition 6.4.13: The flat BN relative to \mathbf{X} is a BN $\mathcal{B}_{\mathbf{X}}$ over \mathbf{X}^* , in which the parents of a node are as specified in the definition of ancestor set. The CPF of a node $I.\sigma.A$ is the same as the CPF in the local probability model for A . The CPF of a node $Q = I.\sigma.\#[A.\rho = v]$ is defined as follows: Let \mathbf{u} be the values of $\theta(I.\sigma.A[1].\rho), \dots, \theta(I.\sigma.A[\max \#[A]].\rho)$. Let m be the value of $I.\sigma.\#[A]$. Then $CPF_Q(n|m, \mathbf{u}) = 1$ if there are exactly n out of $\theta(I.\sigma.A[1].\rho), \dots, \theta(I.\sigma.A[m].\rho)$ such that the value of $\theta(I.\sigma.A[i].\rho)$ in \mathbf{u} is v , and 0 otherwise. ■

Definition 6.4.14: f is a function from the set \mathcal{F} of basic events to $[0, 1]$ defined as follows. Let $[\mathbf{X} = \mathbf{x}]$ be a basic event. Let \mathbf{Z} be the set of number variables in $\mathcal{B}_{\mathbf{X}}$, and let $[\mathbf{Z} \geq \min \#[\mathbf{Z}]]$ denote the event in which for each $Z \in \mathbf{Z}$, $[Z]^\omega \geq \min \#[Z]$. We define f by

$$f([\mathbf{X} = \mathbf{x}]) = P_{\mathcal{B}_{\mathbf{X}}}(\mathbf{X} = \mathbf{x} \mid \mathbf{Z} \geq \min \#[\mathbf{Z}]). \quad \blacksquare$$

In order to carry out our program of defining a probability measure over possible worlds, we must now prove that the function f is indeed additive on the set of basic events. In the previous chapter, this fact (Lemma 5.4.7) rested on the fact that the probability of an event involving \mathbf{X} was the same in all $\mathcal{B}_{\mathbf{Y}}$ that contained \mathbf{X} (Lemma 5.4.6). We can now use Lemma 6.4.11 to show that this still holds.

Lemma 6.4.15: Let \mathbf{X} and \mathbf{Y} be finite sets of basic variables of \mathcal{K} , with $\mathbf{X} \subseteq \mathbf{Y}^*$. Let \mathbf{Z} be the set of conditionable number variables in $\mathcal{B}_{\mathbf{Y}}$. Then for any value $\mathbf{x} \in$

$Val[\mathbf{X}]$,

$$P_{\mathcal{B}[\mathbf{Y}]}(\mathbf{X} = \mathbf{x} \mid \mathbf{Z} \geq \mathbf{min} \#[\mathbf{Z}]) = f([\mathbf{X} = \mathbf{x}]).$$

Proof: Let \mathbf{W} be the set of conditionable number variables in $\mathcal{B}_{\mathbf{X}}$. Since $\mathbf{X} \subseteq \mathbf{Y}^*$, $\mathbf{X}^i \subseteq \mathbf{Y}^*$ for all i (inductively), and $\mathbf{X}^* \subseteq \mathbf{Y}^*$. Therefore $\mathcal{B}_{\mathbf{X}}$ is a subnetwork of $\mathcal{B}_{\mathbf{Y}}$. By Lemma 6.4.11, for any variable $Z \in \mathbf{Z} - \mathbf{W}$, Z is d-separated from \mathbf{X} by \mathbf{W} in $\mathcal{B}_{\mathbf{Y}}$. Therefore

$$\begin{aligned} P_{\mathcal{B}[\mathbf{Y}]}(\mathbf{X} = \mathbf{x} \mid \mathbf{Z} \geq \mathbf{min} \#[\mathbf{Z}]) &= P_{\mathcal{B}[\mathbf{Y}]}(\mathbf{X} = \mathbf{x} \mid \mathbf{W} \geq \mathbf{min} \#[\mathbf{W}]) \\ &= P_{\mathcal{B}[\mathbf{X}]}(\mathbf{X} = \mathbf{x} \mid \mathbf{W} \geq \mathbf{min} \#[\mathbf{W}]) \\ &= f([\mathbf{X} = \mathbf{x}]). \end{aligned}$$

■

Once this lemma goes through, it follows as in the previous chapter that there exists a unique probability measure that agrees with f on the basic events. We can safely define the semantics of a KB with number uncertainty to be this probability measure.

Theorem 6.4.16: *Let $\langle \mathcal{K}, \mathcal{P} \rangle$ be a RPM with number uncertainty. There is a unique probability measure μ on $\langle \Omega_{\mathcal{K}}, \mathcal{E}_{\mathcal{K}} \rangle$, such that for any basic event $[\mathbf{X} = \mathbf{x}]$, $\mu([\mathbf{X} = \mathbf{x}]) = f(\mathbf{X} = \mathbf{x})$.*

6.4.3 Inference

In setting up the semantics of our language, we conditioned on the constraints on values of number attributes. When performing inference, therefore, we must take these constraints into account and condition on them. Fortunately, this is easy to do. We must make sure that if a variable is processed during the course of computation, any conditionable number variable that is descended from it is also processed. Luckily, we know that such a conditionable number variable is either a variable on the same instance as its ancestor, or a variable on a named instance. In the latter case, it

will be represented by a top-level variable $[I.\#[A]]$. We can take care of this case by setting $Needed([I.\#[A]])$ to be true whenever A is a multi-valued attribute and \mathcal{K} contains instance statements on $I.A$. This will force $[I.\#[A]]$ to be processed during the **SVE** computation. We can take care of the former case similarly, using the fifth argument D to **SVE**. At the beginning of the **SVE** computation, we check if there is a multi-valued attribute A that is an inverse of D . If so, we set $Needed(\#[A])$ to be true. It is sufficient to mark only those multi-valued attributes that are inverses to be needed, because only those can be conditionable for a call to **SVE** that is not on the top-level object.

In addition to forcing the relevant number variables to be processed, we must also condition on the constraints on their values. This can be achieved very simply by adding the following lines to **SVE**, just after the factor g_A for a simple variable A has been computed.

```
If  $A$  is a number attribute  $\#[B]$ 
 $g_A = g_A[A \in \{\min \#[B], \dots, \max \#[B]\}]$ .
```

If there is explicit evidence on B , and the evidence agrees with the constraint, this operation will have no effect, since all values of B not in agreement with the constraint will already have been eliminated as possibilities. Evidence that disagrees with the constraint will result in the entire factor having the value zero, and the result of the entire computation will be the zero factor. This will eventually be detected when **SolveQuery** tries to normalize the result. Of course, evidence disagreeing with the constraints is nonsensical — an example is asserting the sentence “My mother has no children”.

Inference with quantifiers in the presence of number uncertainty is done in much the same way as in Section 6.2.3, where the number of values of the multi-valued attributes was fixed. As we did there, we compute a joint distribution over the values of the quantifiers Q on A .¹⁰ Now, $\#[A]$ will be a parent of Q . As before, we use the

¹⁰For simplicity, we ignore the attributes on which A depends in the following discussion, as well as asserted values of A . It is clear that we can treat these situations in the same way as in Section 6.2.3.

recurrence relation of Equation 6.2 to compute $P_m(\mathbf{Q} = \mathbf{n})$, where m varies from 0 to $\max \#[A]$. The meaning of $P_m(\mathbf{Q} = \mathbf{n})$ is now the probability that the first m fillers of A will contribute the vector \mathbf{n} to the quantifiers \mathbf{Q} , *if there are at least m fillers*. The CPF for \mathbf{Q} , given $\#[A]$, is given by

$$P(\mathbf{Q} = \mathbf{n} \mid \#[A] = m) = P_m(\mathbf{Q} = \mathbf{n}).$$

6.5 Reference Uncertainty

In addition to uncertainty over the number of objects that are related to an object, one may also have uncertainty as to the identities of the related objects. This is a type of structural uncertainty that we call *reference uncertainty*. The reason for this name is that when we mention an object $I.A$, and we have uncertainty as to which object is related to I via attribute A , we do not know to which object the term $I.A$ actually refers. One can think of A as a pointer in a programming language, with the value of the pointer itself being uncertain. There is an element of indirection in getting from the term $I.A$ to the actual object related to I by A . First one needs to access the pointer, and then take the value of the pointer to find the related object. In our terminology, the pointer whose value is the identity of the related object is called a *reference variable*.

Reference uncertainty is quite natural at the instance level. If we have a knowledge base containing a number of named instances, we may know that I is related to either J or K via A , but not which. However, as we will see in Section 6.5.3, there is also a form of reference uncertainty that is appropriate at the class level. We shall begin by discussing instance-level reference uncertainty. For simplicity, we develop the theory for a KB with no multi-valued attributes.

6.5.1 Instance-Level Reference Uncertainty

The representation of instance-level reference uncertainty is quite simple. Instead of instance statements of the form $I.A = J$, instance statements can now take the form $I.A = \{J_1 : w_1, \dots, J_n : w_n\}$, where the J_i are all instances of the range type of A ,

and \mathbf{w} is a probability distribution over $\{1, \dots, n\}$. Such a statement corresponds to a situation where $I.A$ is equal to one of the J_i , but exactly which one is unknown. We have a probability distribution over the identity of $I.A$, with the probability that $I.A$ is J_i being w_i . We associate $I.A$ with a reference variable $Ref[I.A]$ whose value ranges over $\{J_1, \dots, J_n\}$. We will require that in any possible world ω , if $[Ref[I.A]]^\omega = J_i$, then $[I.A]^\omega = [J_i]^\omega$. A standard instance statement of the old kind can be viewed as a special case of the new kind of instance statement: $I.A = J$ is equivalent to $I.A = \{J : 1\}$. We can therefore assume for simplicity that the KB contains no instance statements of the old kind.

As we have formulated things so far, the value of a reference variable is independent of the values of all other variables. However, we can treat a reference variable just like a number attribute, giving it parents and a local conditional probability function. Each of the parents of a reference variable should be a basic variable of the KB. This allows the value of a reference variable to depend on other properties of the named instances, which can be a natural thing to do. For example, the identity of *Jane-Student.Advisor* could be *Professor-Koller* or *Professor-Mitchell*, depending on whether *Jane-Student.Studies-AI* is true or false.

Defining the probabilistic semantics for models with reference uncertainty is fairly straightforward. As with number uncertainty, even though we do not know the exact structure of the world, the structure is fixed given the values of the reference variables. We can once again define a probability measure over possible worlds in terms of the probabilities of assignments of values to basic variables. The basic variables now include reference variables.

One issue to be dealt with is that if a variable $I.A$ has a parent $I.B.\sigma$, and $I.B$ is referentially uncertain, with $Ref[I.B]$ ranging over J_1, \dots, J_n , we know that one of the $J_i.\sigma$ is the actual parent of $I.A$, but we do not know which. In other words, the θ function that was used to find which variable is actually the parent of another variable is not uniquely defined. We resolve this issue by making *all* of the $J_i.\sigma$ parents of $I.A$, as well as $Ref[I.A]$, and making the CPF of $I.A$ use the value of $Ref[I.A]$ to choose the appropriate parent.

Formally, we replace the θ function with one that returns two values. The first is

a set of possible referents. The second is the set of reference variables on which the identity of the chain depends. $\theta(I.\sigma)$ is now defined as follows:

Function $\theta(I.\sigma)$

If σ has the form $\tau.A.B.\tau'$, where B inverse-of A

Return $\theta(I.\tau.\tau')$.

Else if $\sigma = A.\sigma'$, and $I.A$ has an associated reference variable $Ref[I.A]$

Let the range of $Ref[I.A]$ be $\{J_1, \dots, J_n\}$.

Let $\langle \mathbf{X}_i, \mathbf{R}_i \rangle = \theta(J_i.\sigma')$.

Return $\langle \cup_{i=1}^n \mathbf{X}_i, \cup_{i=1}^n \mathbf{R}_i \cup Ref[I.A] \rangle$.

Else

Return $\langle \{I.\sigma\}, \emptyset \rangle$.

Let the second return value of $\theta(I.\sigma)$ be \mathbf{R} . If \mathbf{R} is non-empty, we say that $I.\sigma$ is *referentially uncertain*, otherwise it is *referentially certain*. For each value \mathbf{r} of the reference variables \mathbf{R} we define $\theta_{\mathbf{r}}(I.\sigma)$ to be the referent corresponding to the given values of the reference variables. Formally:

Function $\theta_{\mathbf{r}}(I.\sigma)$

If σ has the form $\tau.A.B.\tau'$, where B inverse-of A

Return $\theta_{\mathbf{r}}(I.\tau.\tau')$.

Else if $\sigma = A.\sigma'$, and $I.A$ has an associated reference variable $Ref[I.A]$

Let J_i be the value of $Ref[I.A]$ in \mathbf{r} .

Return $\theta_{\mathbf{r}}(J_i.\sigma')$.

Else

Return $I.\sigma$.

We change the definition of the dependency relationship \leftarrow so that $X \leftarrow Y$ iff Y is in either return value of $\theta(I.\sigma.v)$ for some parent v of A . This causes both the possible parents and the reference variables used to resolve them to be parents of X .

In order to guarantee acyclicity of the dependency model using the global dependency graph $\mathcal{G}[\mathcal{K}]$, we must treat each of the possible values of $I.A$ in the same way we used to treat the actual asserted value. Namely, if $B \in Imp[A]$, and J_i is a possible value of $I.A$, there is an edge from $J_i.B$ to $I.A$ in $\mathcal{G}[\mathcal{K}]$.

We need to define the CPF of each variable to take into account all the possible parents as well as the relevant reference variables. Let $X = I.\sigma.A$ be a variable, and let the parents in the probability model for A be v_1, \dots, v_m . Let $\langle \mathbf{U}_i, \mathbf{R}_i \rangle$ be $\theta(I.\sigma.v_i)$. The parents of X will then be $\cup_{i=1}^m (\mathbf{U}_i \cup \mathbf{R}_i)$. Let $\mathbf{u}_i, \mathbf{r}_i$ be the values of \mathbf{U}_i and \mathbf{R}_i respectively. Using the θ_r function defined above, $\theta_{\mathbf{r}_i}(\mathbf{u}_i)$ is the value of the particular actual parent of X corresponding to the formal parent v_i of A , as determined by the values of the reference variables. We therefore define the CPF for X as follows:

$$\begin{aligned} \text{CPF}_X(x \mid \mathbf{u}_1, \mathbf{r}_1, \dots, \mathbf{u}_m, \mathbf{r}_m) = \\ \text{CPF}_A(x \mid (\theta_{\mathbf{r}_1}(I.\sigma.v_1))(\mathbf{u}_1), \dots, (\theta_{\mathbf{r}_m}(I.\sigma.v_m))(\mathbf{u}_m)). \end{aligned}$$

As was the case with number uncertainty, the CPF for X exhibits *context-specific independence (CSI)* [12]. The CSI is in fact much stronger in this case than it was previously. For any given value \mathbf{r}_i for the reference variable \mathbf{r}_i , only one of the \mathbf{U}_i will be relevant. Any U that is not equal to some $\theta_{\mathbf{r}_i}$ will be irrelevant to X in the context $\mathbf{R} = \mathbf{r}$.

6.5.2 Inference With Reference Uncertainty

How do we do inference with reference uncertainty? One possibility is to condition over all the possible values of the reference variables. This uses the fact that

$$P(Q, E) = \sum_{\mathbf{r} \in \text{Val}[\mathbf{R}]} P(Q, E, \mathbf{R} = \mathbf{r}).$$

For each assignment \mathbf{r} of values to the reference variables, we have a model with known structure, for which we already know how to do inference. The obvious difficulty with this approach is that the number of different possible structures, and therefore the number of terms in the summation, is exponential in the number of reference variables.

An alternative possibility is to use the fact that the reference variables behave just like simple variables, and can appear as variables in the probability model for the top-level object. Variables that have referentially uncertain parents will depend

on the reference variables, and their CPFs will be as we have described. We can then use the same inference algorithms that we have used until now, and they will work unchanged to deal with the reference uncertainty. The problem with this approach is that if a reference variable $Ref[I.A]$ has many values J_1, \dots, J_k , and some variable X depends on $I.A.\sigma$, all the variables corresponding to $J_i.\sigma$ will have to be parents of X . Therefore the size of the CPF for X will be exponential in k . Furthermore, because all the $J_i.\sigma$ are parents of X , they will all be connected to each other in the moral graph. In fact, the different $J_i.\sigma$ are *not* conditionally independent of each other given X , but they *are* conditionally independent of each other given both X and $Ref[I.A]$.

By conditioning on the value of $Ref[I.A]$, we can make X depend on only one of the $J_i.\sigma$. Furthermore, we succeed in disconnecting the $J_i.\sigma$ from each other in the moral graph, so the resulting inference will be much cheaper. It seems then, that we are in a bind. Not conditioning results in exponential blowup in the number of values of the reference variables. However, as we said earlier, conditioning on the values of all reference variables results in exponential blowup in the number of reference variables.

It seems that what we need is a way to exploit the benefits of conditioning *locally*, so that we do not need to consider all values of all reference variables simultaneously, but only those that affect a particular part of the network. Zhang and Poole [101] have developed algorithms for performing inference with CSI that achieves just this. Their machinery is fairly sophisticated, and requires a significant extension to standard BN inference algorithms. As it turns out, in our case we can achieve the same effects using standard inference algorithms with a simple trick.¹¹ The idea is that rather than represent the CPF for a variable X with referentially uncertain parents explicitly, we represent it as a product of factors. The idea is as follows. Let us start with a simple case: $X = I.\sigma.A$, A has a single parent v , and the identity of v rests on one reference

¹¹The method presented here resembles the network transformation described by Boutilier, Friedman, Goldszmidt and Koller [12] in some ways, but in fact the two methods are orthogonal, and complement each other. Their methods transforms a tree-structured CPF into a BN that contains multiplexer nodes, while our method can be used to perform inference efficiently in the resulting network.

variable R , that ranges over J_1, \dots, J_k . Let U_i be $\theta_{J_i}(I.\sigma.v)$, i.e., the actual parent of X in the case that $R = J_i$. As defined above, the CPF of X is

$$CPF_X(x \mid \mathbf{u}, r) = CPF_A(x \mid (\theta_r(I.\sigma.v))(\mathbf{u})).$$

CPF_X can be decomposed into a product $\prod_{i=1}^k f_i(x, u_i, r)$, where f_i is a factor over $\{X, U_i, R\}$ defined by

$$f_i(x, u_i, r) = \begin{cases} CPF_A(x \mid u_i) & \text{if } r = J_i \\ 1 & \text{if } r \neq J_i \end{cases} \quad (6.3)$$

It is easy to see that $CPF_X = \prod_i f_i$, because

$$\begin{aligned} \prod_i f_i(x, u_i, r) &= f_{j:r=J_j}(x, u_j, r) \prod_{i \neq j} f_i(x, u_i, r) \\ &= CPF_A(x \mid u_j) \prod_{i \neq j} 1 \\ &= CPF_A(x \mid (\theta_r(I.\sigma.v))(\mathbf{u})). \end{aligned}$$

The benefit of this decomposition is that each of the factors is small. Each factor involves only the child, the reference variable, and one of the possible actual parents. The total size of the factors is linear in the number of values of the reference variable, whereas the size of the product CPF is exponential. Also, since no two possible parents appear in the same factor, there does not have to be an edge between them in the moral graph. Therefore the inference algorithm can exploit the fact that the different possible parents are conditionally independent given the reference variable and the child. This is precisely the effect obtained by conditioning on the value of the reference variable, but it is achieved locally.

If X has other, referentially certain, parents W_1, \dots, W_n , each of the f_i will be a factor over the \mathbf{W} as well as over X, U_i and R . Also, if the identity of $\theta(I.\sigma.v)$ depends on a set of reference variables \mathbf{R} , there will be a factor $f_{\mathbf{r}}$ for each $\mathbf{r} \in \mathbf{R}$. Let $U_{\mathbf{r}}$ denote $\theta_{\mathbf{r}}(I.\sigma.v)$, i.e., the actual parent of X selected by \mathbf{r} . Equation 6.3 is

replaced by

$$f_{\mathbf{r}}(x, \mathbf{w}, u_{\mathbf{r}}, \tilde{\mathbf{r}}) = \begin{cases} CPF_A(x \mid \mathbf{w}, u_{\mathbf{r}}) & \text{if } \tilde{\mathbf{r}} = \mathbf{r} \\ 1 & \text{otherwise.} \end{cases} \quad (6.4)$$

Now suppose that for several parents v^1, \dots, v^m of A , $I.\sigma.v^i$ is referentially uncertain. Let \mathbf{R}^i be the set of reference variables determining the value of $I.\sigma.v^i$. For each such v^i , we let $\langle \mathbf{U}^i, \mathbf{R}^i \rangle$ be $\theta(I.\sigma.v^i)$. We could deal with this situation in the same way as before, by creating a factor $f_{\mathbf{r}}$ for each complete assignment $\mathbf{r} = \langle \mathbf{r}^1, \dots, \mathbf{r}^m \rangle$ to all the reference variables. Each factor would mention X , the referentially certain parents \mathbf{W} , all the reference variables \mathbf{R} , and one variable from each of the \mathbf{U}^i . We would set

$$f_{\mathbf{r}}(x, \mathbf{w}, u_{\mathbf{r}^1}^1, \dots, u_{\mathbf{r}^m}^m, \tilde{\mathbf{r}}) = \begin{cases} CPF_A(x \mid \mathbf{w}, u_{\mathbf{r}^1}^1, \dots, u_{\mathbf{r}^m}^m) & \text{if } \tilde{\mathbf{r}} = \mathbf{r} \\ 1 & \text{otherwise.} \end{cases} \quad (6.5)$$

Unfortunately, the number of factors $f_{\mathbf{r}}$ we will need is equal to the number of different values of the \mathbf{R} , so it exponential in the number of referentially uncertain parents of X . In addition, each u^i appears in a factor with all the reference variables, even those that do not determine the identity of the actual i -th parent. In fact, a further decomposition is possible. For each v^i , we define a factor g^i that mentions the possible parents \mathbf{U}^i , the reference variables \mathbf{R}^i and a dummy variable Y^i . The range of Y^i is the same as that of v^i , and g^i is defined by

$$g^i(\mathbf{u}^i, \mathbf{r}^i, y^i) = \begin{cases} 1 & \text{if } (\theta_{\mathbf{r}^i}(I.\sigma.v^i))(\mathbf{u}^i) = y^i \\ 0 & \text{otherwise.} \end{cases} \quad (6.6)$$

The effect of g^i is to ensure that the dummy variable Y^i contains the correct value of the i -th actual parent of X , as determined by \mathbf{r}^i . Each g^i can be decomposed into

product form as before: $g^i(\mathbf{u}^i, \tilde{\mathbf{r}}^i, y^i) = \prod_{\mathbf{r}^i \in \text{Val}[\mathbf{R}^i]} g_{\mathbf{r}^i}^i(u_{\mathbf{r}^i}^i, \tilde{\mathbf{r}}^i, y^i)$, where

$$g_{\mathbf{r}^i}^i(u_{\mathbf{r}^i}^i, \tilde{\mathbf{r}}^i, y^i) = \begin{cases} 1 & \text{if } \tilde{\mathbf{r}}^i = \mathbf{r}^i \text{ and } u_{\mathbf{r}^i}^i = y^i \\ 0 & \text{if } \tilde{\mathbf{r}}^i = \mathbf{r}^i \text{ and } u_{\mathbf{r}^i}^i \neq y^i \\ 1 & \text{otherwise.} \end{cases} \quad (6.7)$$

We can now use the dummy variables to select the correct values of the parents of X , and define the CPF for X in terms of the CPF for A and the dummy variables:

$$\begin{aligned} CPF_X(x \mid \mathbf{w}, \mathbf{u}, \mathbf{r}) &= \sum_{\mathbf{y} \in \text{Val}[\mathbf{Y}]} CPF_A(x \mid \mathbf{w}, \mathbf{y}) \prod_{i=1}^m g^i(\mathbf{u}^i, \mathbf{r}^i, Y^i) \\ &= \sum_{\mathbf{y} \in \text{Val}[\mathbf{Y}]} CPF_A(x \mid \mathbf{w}, \mathbf{y}) \prod_{i=1}^m \prod_{\mathbf{r}^i \in \text{Val}[\mathbf{R}^i]} g_{\mathbf{r}^i}^i(u_{\mathbf{r}^i}^i, \mathbf{r}^i, Y^i). \end{aligned}$$

To see that this transformation is correct, observe that $g^i(\mathbf{u}^i, \mathbf{r}^i, y^i)$ is 0 if $y^i \neq u_{\mathbf{r}^i}^i$, and 1 otherwise. Therefore, for a given value of \mathbf{r} ,

$$\sum_{\mathbf{y} \in \text{Val}[\mathbf{Y}]} CPF_A(x \mid \mathbf{w}, \mathbf{y}) \prod_{i=1}^m g^i(\mathbf{u}^i, \mathbf{r}^i, y^i) = CPF_A(x \mid \mathbf{w}, \mathbf{y}),$$

where each $y^i = u_{\mathbf{r}^i}^i$. This is exactly as required.

Figure 6.2 illustrates the effect of this decomposition on inference. In this example, variable X has a single referentially certain parent V , as well as three referentially uncertain parents. The i -th referentially uncertain parent has two possible values U_1^i and U_2^i , selected by the reference variable R^i . Y^i is the dummy variable added by the decomposition. The figure shows the graph for the set of factors making up the CPF of X . There is a clique over X, V, Y^1, Y^2 and Y^3 . This clique corresponds to the CPF_A term, and its size is the same as that of CPF_A in the local probability model for A . There would be a clique of this size even if all parents of X were referentially certain. In addition, each of the Y^i and R^i are connected to each other, and also to each of the U_j^i . However, the different U_j^i are not connected to each other. The graph is in fact triangulated, and so we can eliminate variables in some order without

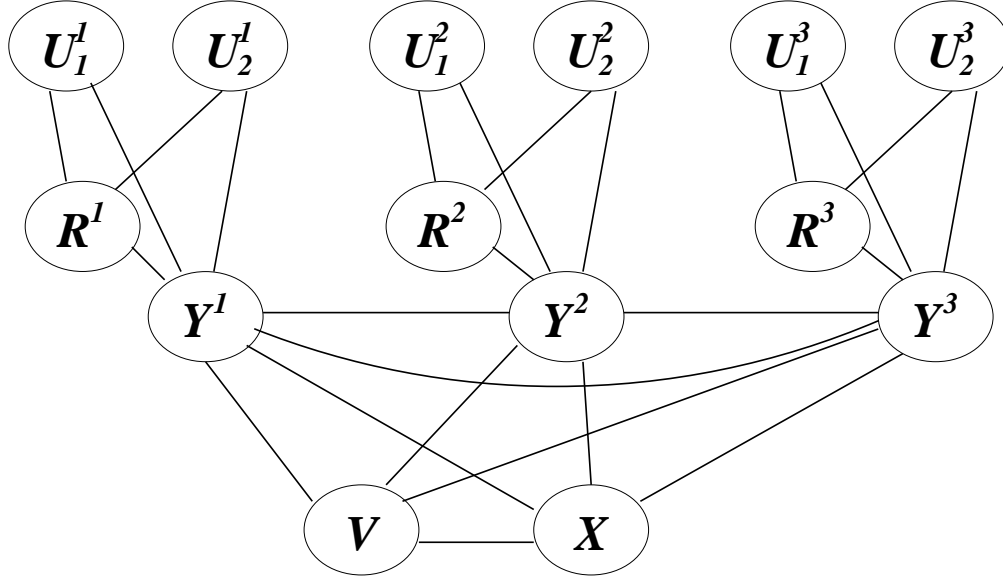


Figure 6.2: Induced graph for decomposed CPF with reference uncertainty.

adding edges to the graph. In fact, the graph shows that the dummy variables \mathbf{Y} serve to separate the reference variables and possible parents from X and V .

6.5.3 Class-Level Reference Uncertainty

Thus far we have focused on structural uncertainty about the relationships between individual instances. We can also model a more general kind of reference uncertainty at the class level. The basic mechanism for this is the *same-as statement*. A same-as statement has two possible forms. One form is A *same-as* σ , where σ is an attribute chain whose domain type is a supertype of the domain type of A , and whose range type is a subtype of the range type of A . A statement of this form imposes the constraint on possible worlds ω that for every $c \in [C]^\omega$ (C being the domain type of A), $[A]^\omega(c) = [\sigma]^\omega(c)$. An example of this type of statement is that for the **PhD-Student** class, **Taking-Reading-Course** *same-as* **Advisor.Teaches-Reading-Course**. The second form of same-as statement is A *same-as* I , where I is a named instance whose type is a subtype of the range type of A . This type of statement imposes the constraint that for all $c \in [C]^\omega$, $[A]^\omega(c) = [I]^\omega$. An example is the statement that for the **Stanford-Student**

subclass of *Student*, *Attends* *same-as* *Stanford-University*.

Arbitrary same-as statements can cause difficulties. For example, consider the statement *A same-as A.A*. From this statement we can deduce that if the value of *c.A* is *d*, then so is the value of *d.A*. The introduction of arbitrary same-as statements therefore produces a very expressive logical language. To avoid dealing with such an expressive language, we restrict the same-as statements in the following way.

Definition 6.5.1: A set of same-as statements in a KB \mathcal{K} is said to be *stratified* if we can number the attributes and named instances of \mathcal{K} in such a way that:

1. If \mathcal{K} contains a statement *A same-as* σ , *A* has a higher number than every attribute in σ .
2. If \mathcal{K} contains a statement *A same-as* *I*, *A* has a higher number than *I*.
3. If \mathcal{K} contains a statement *I.A = J*, *A* has a higher number than *J*. ■

In the absence of reference uncertainty, restricting the same-as statements to be stratified means that we can uniquely resolve any attribute chain in a finite amount of time. The reason is that any application of a same-as statement replaces elements of a chain with lower-ranked elements. Application of an instance statement shortens a chain without increasing the highest rank of elements in the chain. Application of an inverse statement results in removing two elements from a chain. Therefore, only a finite number of statements can be applied to any chain. As a result, we can prove analogues of Lemmas 5.2.6 and 5.2.7, and define a function θ such that $\theta(I.\sigma)$ is a chain *J.ρ* to which no instance, inverse or same-as statements can be applied, and such that $[I.\sigma]^\omega = [J.\rho]^\omega$ in every possible world. We omit the details since they are very much the same as before.

Another restriction we make on the use of same-as statements is that an attribute on which a same-as statement is defined may not participate in an inverse statement. Allowing inverse statements to interact with same-as statements can particularly cause trouble with named instances. For example, the statements *A same-as I*, and *B inverse-of A* would force *I.B* to be equal to every entity of *C*, which means that *C* must have exactly one instance.

We can use same-as statements to model reference uncertainty, by associating a number of possible same-as statements with an attribute A , and defining a probability distribution over them. The attribute A will have an associated reference attribute $Ref[A]$, whose range is a set $\{\sigma_1, \dots, \sigma_m, I_1, \dots, I_n\}$, such that for each σ_j or I_j , $A \text{ same-as } \sigma_j$ or $A \text{ same-as } I_j$ is a valid same-as statement. We enforce the stratification requirement of Definition 6.5.1 in a strict manner: we require that it be possible to enumerate all the attributes and named instances of \mathcal{K} as if each of the $A \text{ same-as } \sigma_j$ and $A \text{ same-as } I_j$ was an actual same-as statement in \mathcal{K} . We impose the constraint on possible worlds that $[Ref[A]]^\omega(c) = \sigma$ implies $[A]^\omega(c) = [\sigma]^\omega(c)$, while $[Ref[A]]^\omega(c) = I$ implies $[A]^\omega(c) = [I]^\omega$.

Semantically, we can treat class-level reference uncertainty in much the same way that we did instance-level reference uncertainty. If attribute A of domain type C has an associated reference variable $Ref[A]$, then for each entity $I.\sigma$ of type C we will have a reference variable $I.\sigma.Ref[A]$. We redefine the θ function as in the previous section to return two values, the first being a set of potential parents, and the second being the reference variables that are used to select the actual parent.

We do need to make some modifications to the global dependency graph to guarantee acyclicity of the dependency model in the presence of same-as statements. Specifically, we need to make sure that for *every* possible parent Y of X , $[Y]$ precedes $[X]$ in $\mathcal{G}[\mathcal{K}]$. First, we add an edge from $C.Ref[A]$ to $C.A$. Also, for each $B \in Imp[A]$, we do the following. If I is a value in $Val[Ref[A]]$, we add an edge from $I.B$ to $C.A$. If $\sigma = D.\sigma'$ is a value in $Val[Ref[A]]$, we add B to $Imp[D]$. The latter step will cause an edge from $C'.B$ to $C.D$ to be added (C' being the range type of D), and may cause other edges to be added, if D is associated with its own reference variable. This process must terminate due to the stratification requirement.

When performing inference with class-level reference uncertainty, we would like to exploit the same types of decompositions that we used for instance level reference uncertainty. In addition, we want to perform the inference in the **SVE** framework. We accomplish this as follows. When processing an attribute A which is referentially uncertain, we know the set of chains beginning with A that are needed to solve the query — these are the chains in $Needed[A]$. We treat each such chain $A.\sigma$ as its

own variable, that serves the same purpose as the dummy variables introduced in Section 6.5.2.

The variable $A.\sigma$ is treated as a simple attribute. One of its parents is $Ref[A]$, and it also has a parent $\rho.\sigma$ or $I.\sigma$ for each ρ or I in $Val[Ref[A]]$. The CPF for the variable $A.\sigma$ selects the value of the appropriate actual parent, according to the value of $Ref[A]$. As described in Section 6.5.2, this CPF can be represented as a product of factors, each one mentioning $Ref[A]$ and one of the possible parents. The **SVE** algorithm is modified by adding the following lines to the part in which complex attributes are processed:

```

If  $A$  has a reference attribute  $Ref[A]$ 
  For each  $A.\sigma$  in  $Needed[A]$  do:
    For each value  $B.\rho$  in  $Val[Ref[A]]$  do:
       $Needed[B] = Needed[B] \cup \{B.\rho.\sigma\}$ .
       $g_{A.\sigma}^{B.\rho}$  is the factor over  $A.\sigma$ ,  $B.\rho.\sigma$  and  $Ref[A]$  such that
         $g_{A.\sigma}^{B.\rho}(x, y, z) = 1$  if  $z \neq B.\rho$ ,
         $g_{A.\sigma}^{B.\rho}(x, y, z) = 1$  if  $x = y$  and  $z = B.\rho$ ,
         $g_{A.\sigma}^{B.\rho}(x, y, z) = 0$  if  $x \neq y$  and  $z = B.\rho$ .
    Else //  $A$  is not referentially uncertain

```

In addition, we replace the line defining the set of factors for the variable elimination phase with the following:

$$\mathbf{f} = \{g_A : Needed[A] \neq \emptyset\} \cup \{g_{A.\sigma}^{B.\rho} : A.\sigma \in Needed[A] \text{ and } B.\rho \in Val[Ref[A]]\}.$$

Allowing statements of the form A *same-as* I poses additional issues. Such a statement means that for every entity c of C , the probability model for c can depend on the specific instance I . The probability models for the generic entities of C are no longer independent of each other, but are correlated through their common dependence on I . Furthermore, $c.A$ is no longer encapsulated inside C .

We can deal with this issue in the **SVE** framework as follows: if, in processing a query on C , a chain $I.\sigma$ is required, $I.\sigma$ is added to the set of inputs τ returned by

the **SVE** call. The **SVE** call also returns a conditional distribution over the answer to the query given the value of $I.\sigma$. The same conditional distribution will hold for all generic instances of C , and for all such instances, the values of the query variables will be conditionally independent given the values of the inputs τ . The input $I.\sigma$ will be passed upwards to the top-level instance, where it can be processed naturally.

6.5.4 Enumerated Classes and the Ace of Spades Problem

Once we allow the value of an attribute of a class to be a named instance, and we allow uncertainty over which named instance it is, a natural extension is to define *enumerated classes*, which consist of a specific set of named instances, together with a probability distribution over the instances in the class.

Definition 6.5.2: Let I_1, \dots, I_n be named instances, and let C be a common superclass of the types of the I_i . Let \mathbf{w} be a probability distribution over $\{1, \dots, n\}$. Then $\langle C, \mathbf{I}, \mathbf{w} \rangle$ defines an *enumerated class* C' which is a subclass of C . The class C is called the *base class* of C' . ■

We allow the range type of a complex attribute A to be an enumerated class $C' = \langle C, \mathbf{I}, \mathbf{w} \rangle$. We can treat this situation simply as syntactic sugar for the model in which A has range type C , and is associated with a reference variable $Ref[A]$. The range of $Ref[A]$ is $\{I_1, \dots, I_n\}$, $Ref[A]$ has no parents in its local probability model, and the CPF for $Ref[A]$ is defined by $CPF_{Ref[A]}(I_i) = w_i$.

We can also allow the type of a named instance to be an enumerated class. This situation is also dealt with as syntactic sugar. A named instance J whose type is enumerated class $\langle C, \mathbf{I}, \mathbf{w} \rangle$ is not actually considered to be a named instance in the KB. However, if there is an instance statement $K.A = J$, it is replaced by the referentially uncertain instance statement $K.A = \{I_1 : w_1, \dots, I_n : w_n\}$. If J is a value in the range of reference attribute A , it is replaced by I_1, \dots, I_n . The CPF for $Ref[A]$ is modified by defining, for any value \mathbf{u} for the parents of $Ref[A]$, $CPF_{Ref[A]}(I_i \mid \mathbf{u}) = w_i CPF_{Ref[A]}(J \mid \mathbf{u})$.

Enumerated classes provide us with a way of addressing the *ace of spades problem* [19, 87, 36]. This problem arose in attempts to combine logical and probabilistic representations. Consider the following reasonable sentence:

(A) The probability that a card is the Ace of Spades is $1/52$.

Cheeseman proposed that such a sentence should be represented by the statement

(B) $\forall x P(x = \text{AceOfSpades} \mid \text{Card}(x)) = 1/52$.

As Schubert points out, the formula (B) does not capture the sentence (A). Under the rules of classical logic, one should always be able to instantiate a universally quantified variable. However, instantiating x with *AceOfSpades* produces the sentence

(C) $P(\text{AceOfSpades} = \text{AceOfSpades} \mid \text{Card}(\text{AceOfSpades})) = 1/52$,

which is clearly wrong. Halpern points out that the confusion is the result of confounding statements about degrees of belief with statistical statements. The sentence (A) is a statistical statement about the distribution of cards in the world, while the formula (C) is a statement about the degree of belief in the sentence *AceOfSpades = AceOfSpades*. Both Schubert and Halpern hold that the correct translation of sentence (A) uses a quantifier $\forall^r x$ meaning “for a randomly chosen x ”. Logics with this type of quantifier have been developed by Bacchus [4] and Halpern [35].

The semantics of our language is defined in terms of a probability measure over possible worlds. Our language is avowedly concerned with describing degrees of belief in statements characterized by events in the event space, and not with describing statistical statements. Nevertheless, the use of enumerated classes allows us to deal easily with sentences like (A). The key is to formalize the notion of a “randomly chosen card” by specifying the process through which the card is chosen. We can formalize the process of choosing a card from a deck with uniform probability, by creating an enumerated subclass **Card-From-Deck** of **Card**, containing all the different cards, with a uniform probability over them. If we then want to talk about the probability that a randomly chosen card is the Ace of Spades, we create a **Draw** class,

with a complex attribute *Drawn-Card* of type *Card-From-Deck*. Observing that for a particular drawing I , the drawn card is the Ace of Spades is performed by setting $I.Ref[Drawn-Card] = Ace-Of-Spades$. It holds that for an instance I of class *Draw*, $P(I.Ref[Drawn-Card] = Ace-Of-Spades) = 1/52$, which is the meaning of the sentence (A).

6.5.5 Type Uncertainty

In addition to uncertainty about the number of objects in the domain and the relationships between them, we may also have uncertainty about their types. We call this kind of structural uncertainty *type uncertainty*. One may represent type uncertainty at both the class and instance level. Type uncertainty can be handled quite simply using the mechanisms we have developed for reference uncertainty.

At the instance level, suppose I is a named instance, whose type is known to be a subclass of C , and let C_1, \dots, C_n be subclasses of C . We may be uncertain as to which of the C_i is the actual type of I . For example, we may not know whether *Jane-Student* is an *Undergraduate-Student* or *Graduate-Student*, but we have a probability distribution over the two possibilities. To model this situation, we associate a type variable $Type[I]$ with I , whose value ranges over C_1, \dots, C_n . The type variable has a local probability model, with a set of parents and a CPF.

We can deal with this situation using syntactic sugar. We introduce named instances I_1, \dots, I_n , with the type of I_j being C_j . The type variable $Type[I]$ is replaced with a reference variable $Ref[I]$, ranging over I_1, \dots, I_n . $Ref[I]$ will have the same parents as $Type[I]$, and the CPF of $Ref[I]$ will be defined by $CPF_{Ref[I]}(I_j \mid \mathbf{u}) = CPF_{Type[I]}(C_j \mid \mathbf{u})$.

At the class level, we may have uncertainty as to the range type of the complex attribute A on C . For example, we may have uncertainty over whether a value of the *Taking* attribute of *Seminar-Course* is a *Undergraduate-Student* or a *Graduate-Student*. If it is known that the range type of A is a subclass of D , and D_1, \dots, D_n are subclasses of D , we can associate A with a type attribute $T[A]$.

Again we treat this situation as syntactic sugar. We introduce attributes A_1, \dots, A_n

on C , where the range type of A_i is D_i . We replace $Type[A]$ with reference attribute $Ref[A]$, ranging over the chains A_1, \dots, A_n , set the parents of $Ref[A]$ to be the same as those of $Type[A]$, and define the CPF of $Ref[A]$ by $CPF_{Ref[A]}(A_i \mid \mathbf{u}) = CPF_{Type[A]}(C_i \mid \mathbf{u})$.

It is clear that one can combine type uncertainty with other kinds of reference uncertainty using this method. For example, we may not know whether the value of A is the instance I , the same as the value of chain σ , or a different entity whose type is either C_1 or C_2 . We can model this situation by introducing two new attributes A_1 and A_2 of class C_1 and C_2 respectively, and associate A with a reference variable $Ref[A]$ taking values in the range $\{I, \sigma, A_1, A_2\}$.

6.6 Discussion and Possible Extensions

We began this chapter by considering languages with multi-valued attributes, in situations where the number of values is always known. We then presented the two main kinds of structural uncertainty: number uncertainty and reference uncertainty, and also described type uncertainty as a variation on reference uncertainty. We developed the theory of reference uncertainty in the context of a language with no multi-valued attributes. It is possible to combine reference uncertainty with multi-valued attributes, but the range of linguistic possibilities is large.

In its most general form, we are talking about uncertainty over the set of values of a multi-valued attribute. Such a set can be described by listing the identities of the individual members, as they would have appeared as the values of reference attributes. So, for example, if A is multi-valued, a possible value for $Ref[A]$ is $\{I, \sigma, C, C, C'\}$, meaning that A has five fillers, one of which is I , another of which is the same as σ , while the other three are generic, two from class C and one from class C' . This is an extremely general language, but the range of values of $Ref[A]$ may be enormous, and completely impractical to list explicitly.

One can imagine a variety of ways to specify such set-valued reference uncertainty more compactly. For example, a number attribute may specify the number of variables in the set, and then a separate reference attribute may specify uncertainty over the

particular fillers. Care must be taken in defining the semantics, to make sure that two different fillers are not the same entity. Alternatively, a more restricted language may combine a number attribute and a type attribute, with both the number of fillers and the types of the individual fillers being uncertain. A third alternative would have a variable for each of a specified set of instances, specifying whether that instance is a filler, and an additional number variable determining the number of possible fillers. The space of possible languages is large, and we do not yet have a strong sense for which would be the most practical.

Reference attributes allow the value of a filler to be selected from some set of possibilities. The set is *extensionally* specified — the different possibilities are listed explicitly. One can also imagine selecting the value of a filler from some *intensionally* specified set. This may be expressed by saying that for a single-valued attribute A , the filler of A is one of the fillers of a multi-valued attribute B . Thus far, accomodating this extension would not be too difficult, but one can easily imagine that the identity of the filler of A depends probabilistically on the properties of the fillers of B . This kind of selection process is a very natural one — for example, the winner of a race is one of the participants in the race, and the identity of the winner depends on the speed of the various participants. This is such a natural situation that we want to find a way to model it in the language, but we have not as yet found an appropriate way to express this situation that is natural, compact and supports efficient inference.

Chapter 7

Recursive Probability Models

7.1 Introduction and Examples

In Chapter 5, we went to some trouble to make sure that the dependency model for a RPM contains no infinite dependency chains. There are two reasons for this. The first reason is semantic. In the absence of infinite chains, the probability distribution over any finite set of variables can be specified by looking at the finite set of ancestors of those variables. We used this fact to guarantee that for any KB, there exists a unique probability measure embodying the statements in the class probability models. The second, related reason is that eliminating infinite chains guarantees that any query can be answered by looking at only a finite set of variables, and so exact probabilistic inference can be performed in a finite amount of time.

Despite the usefulness of doing so, eliminating infinite dependency chains does limit the expressiveness of our language in a fundamental way. Probability models with infinite dependency chains are quite natural and common. We call such models *recursive probability models*. The reason for the name goes back to our intuition of defining a probabilistic model in terms of a stochastic generative process that generates instances of the model. If the model has infinite dependency chains, the behavior of the process is similar to the behavior of a recursive program in a programming language. The program may never terminate due to the recursion, but we can still consider what the state of the world would look like if we allowed the

program to run for an infinite amount of time. In particular, the program will define local relationships between the variables that can be expressed in terms of conditional probabilities. These local relationships can be used to define semantics for the language.

We begin with some examples.

Example 7.1.1:

A very common example of a recursive probability model is a *Markov chain*. We consider here only discrete, finite-state, homogeneous chains. In such a chain, the state of the world X^t at any time t is an element in a set D of cardinality n . The chain is characterized by an initial probability distribution P^0 over D , and a transition function, that specifies the conditional probability of the state X^{t+1} given the state X^t . The transition function can be described using a $n \times n$ matrix T , in which $T(x, x')$ specifies the probability of transitioning from state x to x' . The matrix T must satisfy the constraint that for all x , $\sum_{x'} T(x, x') = 1$. In other words, T is a CPF from D to D .

A Markov chain describes a probability distribution over the state of the world at time t , by $P^t = T^t P^0$. The state of the world at time 0 is described by the initial distribution P^0 , and the distribution at time $t + 1$ is taken by applying the transition matrix T to the distribution at time t . Two natural questions arise with regard to a Markov chain. The first is whether there exists a distribution P such that $P = TP$. Such a distribution is called an *invariant* of the chain. The second question is, given that an invariant P exists, is P unique, and does $T^t P^0$ converge to P for any P^0 ? A chain that satisfies the second property is called *ergodic*, and the distribution P is called the *stationary distribution* of the chain.¹ A sufficient but not necessary condition for a chain to be ergodic is that all of the entries of T are non-zero.

The theory of Markov chains is extremely well-studied, and can be found in many textbooks (see, for example [52]). Our interest here is not in the theoretical properties of Markov chains but in their representation. A homogeneous finite-state Markov chain can be represented very simply in our language as follows. There is

¹Various definitions of ergodic can be found in the literature, not all of which are exactly equivalent to the one here. Our definition is the same as that of [72].

an `Instant` class, with simple attribute `State`, ranging over D , and complex attributes `Previous` and `Next`, which are inverses of each other. The `State` attribute has as parent `Previous.State`, and its CPF is the transition matrix T . There is also a `First-Instant` subclass of `Instant`. In the `First-Instant` model, `State` has no parents, and its CPF is specified by the initial distribution P^0 . With this representation, one can take both a “forwards” and a “backwards” view of the chain. An instance I of `First-Instant` defines the sequence of variables $I.State, I.Next.State, I.Next.Next.State, \dots$, embodying a Markov chain beginning in the initial state. On the other hand, an instance J of `Instant` defines the variables $J.State, J.Previous.State, J.Previous.Previous.State, \dots$. The distribution over $J.State$ is the result of an infinite dependency chain, in which each step in the chain is associated with the transition matrix T . If the chain is ergodic, the distribution over $J.State$ is the stationary distribution of the chain.

We would like an inference algorithm for our language that is capable of handling both these situations. It should be able to answer queries looking forward from an initial state, or queries about the stationary state of the distribution. Of course, it should be a general algorithm, and not perform computations that rely on the knowledge that the model is a Markov chain.

The first type of query can be answered by only looking at a finite number of states, and can be handled by the **SVE** algorithm described in previous chapters. To solve the second type of query exactly requires looking at infinitely many variables. However, as we shall show, we can compute an iterative approximation to the answer to the query, that will produce better and better bounds. In the case of an ergodic chain, the bounds will converge to the correct answer. For a non-ergodic chain, there may not be a single correct answer. Nevertheless, the computed bounds will be sound, in the sense that a correct answer to the query will always lie between them. ■

Example 7.1.2: Many natural extensions to Markov chains have been explored. A simple extension is the *Hidden Markov Model (HMM)* [85]. In a HMM, the state at time t is decomposed into a hidden state X^t and observed state Y^t . In addition to the distribution over the initial state, an HMM is characterized by a *transition model* T , specifying the conditional probability of X^{t+1} given X^t , and an *observation model*, specifying the conditional probability of the observations Y^t given the state

X^t . HMMs can be very easily modeled in our language by adding an **Observation** attribute to the **Instant** class from Example 7.1.1. The parent of **Observation** is **State**, and its CPF is the observation model O .

In a *dynamic Bayesian network* [21, 53], the state at time t is further decomposed into a set of variables $\mathbf{X}^t = X_1^t, \dots, X_n^t$. Each of the X_i^t has a set of parents, which are other variables from the previous or current instant, and a local CPF. A DBN can obviously be represented in our framework, with the **Instant** class now containing an attribute for each of the X_i , and each parent of X_i being either an attribute X_j from the same time slice, or a **Previous.X_j** from the previous time slice.

All the models discussed so far are well-studied models, recast in our relational language. However, once we have a relational representation, we can imagine representing more complex dynamic models. One such representation, *dynamic OOBNs* (*DOOBNs*) was proposed in [27]. A DOOBN consists of a number of subsystems, each of which evolve in time. The state of an object at time t can depend on its state at a previous time, or on the states of other objects at time t . We allow the different subsystems to evolve at different rates, so for example, in a model of freeway traffic, the weather may evolve much more slowly than the position of a car. Although the representation of complex dynamic systems requires further study, it appears that the use of an object-based representation language can greatly facilitate the representation and inference for these types of systems. ■

Example 7.1.3: So far, we have considered models in which the infinite dependency chains are *linear*. The state at each point in time has one parent, which is the state at the previous instant. A natural non-linear recursive probability model is a model for the transmission of genetic material from generation to generation. For simplicity, we will describe a model for the propagation of a single gene, with two states. This model can easily be extended to multiple genes, with multiple states, and to model the correlations between propagation of different genes due to their proximity in the genome.

Our example model contains a single **Person** class, with two complex attributes **Mother** and **Father**. The class has the simple attributes **Phenotype**, describing the observed property of the person, and **M-Chromosome** and **P-Chromosome**, representing

the gene type inherited from the mother and father respectively. The parents of **Phenotype** are **M-Chromosome** and **P-Chromosome**, and the CPF for **Phenotype** takes into account which state of the gene is dominant and which is recessive. The parents of **M-Chromosome** are **Mother.M-Chromosome** and **Mother.P-Chromosome**, with a person inheriting one of the mother's two chromosomes at random. Similarly, the parents of **P-Chromosome** are **Father.M-Chromosome** and **Father.P-Chromosome**.

A family tree can be modeled by instantiating a set of **Person** instances, and connecting parents to children as specified in the tree. A knowledge base of this form defines a probability distribution over the genetic properties of the people in the tree, given the genetic material at the roots. Typically, one will observe the **Phenotype** attribute for some of the people in the tree, and use that to predict properties of other people. The properties of the people at the roots of the tree are actually not known, however. In fact, they are the result of an infinite chain of genetic transmissions. This infinite model is only an approximation to the true model — presumably life does not go back infinitely far. Also, in the true family tree, presumably the same ancestor will show up via different paths. This can be modeled exactly if the ancestor is a named instance actually appearing in the tree, but for unnamed instances, the unique names assumption forces the model to assume that all unnamed ancestors are distinct. Nevertheless, it is not an unreasonable way to model things. For a large and richly detailed family tree, the information about named people in the tree will tend to dominate the uncertainty about the roots. ■

Example 7.1.4: Recursive probability models are often used in modeling natural language. A commonly used model is a *stochastic context-free grammar (SCFG)* [17], which is a probabilistic variant of a context-free grammar (CFG). A CFG has an alphabet of symbols, divided into *non-terminal* and *terminal* symbols. Each non-terminal X is associated with a set of *productions* of the form $X \rightarrow a_1 \dots a_n$, where each a_i is a terminal or non-terminal symbol. One of the non-terminals is identified as a special *sentence symbol* S . In a SCFG, X also has an associated probability distribution P_X over the productions associated with X .

A SCFG specifies a generative process that generates strings of non-terminals. The process is as follows:

```

String = S.
While String contains non-terminals do:
  Choose the leftmost non-terminal  $X$  in String.2
  Choose a production  $X \rightarrow a_1 \dots a_n$ , according to  $P_X$ .
  Replace  $X$  in String by  $a_1 \dots a_n$ .
Return String.

```

If the process terminates, it will return a string of terminal symbols. However, there may be a positive probability of non-termination, depending on the specific probabilities over the productions. For example, consider a grammar with a single terminal a , a single non-terminal S , and two productions $S \rightarrow SS$ and $S \rightarrow a$. Let the probability of the first production be α . It can be shown that the process terminates with probability 1 iff $\alpha \leq 1/2$. In case of non-termination, we may say that the process generates the special symbol \perp , or alternatively, we may view it as generating an infinitely long string of symbols. The latter view is particularly appropriate if the grammar is in *Greibach normal form*, in which every production begins with a terminal symbol. In that case, each application of a production adds a terminal to the string. We view the generative process as defining a probability measure over strings of non-terminals.

A SCFG can be specified in our language as follows. There is a **String** class, with three attributes, the simple attribute **First**, ranging over the terminal symbols, the boolean simple attribute **Is-Empty**, and the complex attribute **Rest** of range type **String**. Since we will never actually instantiate instances of **String**, but only of its subclasses, the local probability models of the simple attributes are immaterial, but for definiteness we specify them to have no parents and the uniform CPF.

There is an **Concat** subclass of **String**, representing the concatenation of two strings. This subclass illustrates how something that we normally think of as a function can be represented in our language. The **Concat** class has two complex attributes **Left** and **Right** of type **String**, corresponding to the arguments of the function. The **Is-Empty** attribute of **Concat** depends on **Left.Is-Empty** and **Right.Is-Empty**, and its CPF is the **and** function. The **First** attribute of **Concat** depends on **Left.First**, **Left.Is-Empty** and

²Any non-terminal could be chosen here. The leftmost one is specified for the sake of definiteness.

`Right.First`, and its CPF specifies it to be equal to `Left.First` if `Left.Is-Empty` is false, otherwise it is equal to `Right.First`.

To specify the `Rest` attribute of the `Concat` class, we use a reference attribute `Ref[Rest]`. `Ref[Rest]` has two possible values and depends on `Left.Is-Empty`. One possible value is `Right.Rest` — `Ref[Rest]` takes this value if `Left.Is-Empty` is true. The other possibility is that we need to “compute” the value of `Concat(Left.Rest, Right)`, and assign this value to the `Rest` attribute of `Concat`. This type of argument passing to a function can be modeled using the binding mechanisms of OOBNS. We add a `Subcall` attribute to `Concat`, itself of type `Concat`, to represent the result of the recursive subcall. The arguments are passed to the `Subcall` attribute by binding the `Left` and `Right` attributes: $\Theta[\text{Subcall.Left}]$ is `Left.Rest`, and $\Theta[\text{Subcall.Right}]$ is `Right`. The result of the recursive computation can be referred to by setting the value of the `Ref[Rest]` reference attribute to be `Subcall`.

Given basic `String` and `Concat` classes, we can now construct a class for every symbol in a SCFG. We will assume that the grammar is in *Chomsky normal form*: every production is either $X \rightarrow X_1X_2$ or $X \rightarrow a$, where X_1 and X_2 are non-terminals, and a is a terminal. Every CFG can be transformed into Chomsky normal form. The class C_a for a terminal symbol a is a subclass of `String`, in which the CPF for `First` sets the value of `First` to be a with probability 1, while the CPF for `Is-Empty` sets `Is-Empty` to be false with probability 1. The range type of `Rest` is the subclass `Empty-String` of `String`. In the `Empty-String` class, `Is-Empty` is true with probability 1.

There is an abstract `Non-Terminal` class, with complex attribute `Value` of range type `String`. The class C_X for a non-terminal X is a subclass of `Non-Terminal`, defined as follows. For each production $R_i = X \rightarrow a$, C_X has an attribute R_i of range type C_a . For each production $R_i = X \rightarrow X_1X_2$, C_X has complex attributes X_1 and X_2 , of range type C_{X_1} and C_{X_2} respectively, and a complex attribute R_i of range type `Concat`, with the bindings $\Theta[R_i.\text{Left}] = X_1.\text{Value}$, $\Theta[R_i.\text{Right}] = X_2.\text{Value}$. The `Value` of X could be the value of any of the R_i , and it is determined using the reference attribute `Ref[Value]`, ranging over the R_i . `Ref[Value]` has no parents, and its CPF is specified according to P_X .

In [55], we presented a *stochastic functional language* for describing recursive probability models. This simple language makes it very easy to specify generative processes functionally. The focus of the language is on the generative process itself, rather than the objects and attributes, which makes it more convenient for describing models that have a generative flavor, such as SCFGs. In this language, we could use a standard functional definition of **Concat**, and each non-terminal is also represented by a simple function. The stochastic functional language is only superficially different from the language of relational probability models. The procedure we have outlined here for transforming a SCFG into our framework is a general one that can be applied to any functionally specified model.

The standard inference task in a SCFG is to compute the probability that a certain string s was generated by the grammar. The algorithm used for this task is the *inside* algorithm. The algorithm works bottom up, computing for each substring s and non-terminal X the probability π_s^X that s was generated by X . If s is a single symbol a , π_s^X is the probability according to P_X of the production $X \rightarrow a$. If s consists of more than one symbol, the probability π_s^X is computed by considering all possible ways in which s could have been formed using some production $R_i = X \rightarrow X_{i1}X_{i2}$. I.e.,

$$\pi_s^X = \prod_i P_X(R_i) P(s \text{ was generated by } R_i).$$

The probability that s was generated by R_i is equal to the sum over all ways in which s can be decomposed into substrings s_1 and s_2 of the probability that s_1 was generated by X_{i1} and s_2 was generated by X_{i2} . As a result, we have

$$\pi_s^X = \prod_i P_X(R_i) \sum_{\substack{s_1, s_2 \\ s_1 s_2 = s}} \pi_{s_1}^{X_{i1}} \pi_{s_2}^{X_{i2}}.$$

The inside algorithm uses dynamic programming, so that π_s^X is computed using the previously computed and stored values for $\pi_{s'}^{X'}$ for all substrings s' of s and all non-terminals X' . As it turns out, the **SVE** algorithm, adapted for recursive models as we shall describe in this chapter, mimics the behavior of the inside algorithm. In processing a query on a string s , it recursively generates subqueries for each of

the substrings of s . Eventually the process bottoms out and results are propagated upwards. The caching mechanism of **SVE** serves to implement the dynamic programming aspect of the algorithm — the results of the low-level queries are stored and reused and do not need to be recomputed each time.

Of course, once we have expressed SCFGs in our language, we can easily go on to specify richer grammars. For example, we may want high-level non-terminals to pass case, gender and number information to lower-level non-terminals. This can easily be achieved by adding the appropriate attributes to the higher-level non-terminal object, and have the lower-level non-terminals refer to them. Another possibility is to include contextual information such as the topic of discussion in the object models. Our language and inference algorithm provide a ready-made framework for experimenting with these richer models. ■

7.2 Language Definition

The language used in this chapter is a selection of language features discussed in earlier chapters that are particularly well suited to representing recursive probability models. Specifically, we want the power of the relational language discussed in Chapter 5, together with the ability to define OOBN-style bindings, so we use the integrated language discussed in Section 5.6.1 that supports OOBN-style bindings in relational probability models. Also, as we saw in the SCFG example, reference uncertainty is important when defining recursive stochastic functions, so we allow that in our language.

However, we restrict the values of reference attributes to be attribute chains, which are treated like same-as statements. That is the type of reference uncertainty that is useful for defining stochastic functions. Furthermore, allowing named instances as values of reference attributes causes difficulties in recursive models. The problem is that if a value of a reference attribute of C is the named instance I , then every instance of C can depend on I . This was also true for non-recursive models; however, in the case of non-recursive models, only finitely many instances of C can be relevant for a particular query. In recursive models, however, there may be infinitely many

instances of C that are relevant to a query and depend on I . An attribute of I may therefore receive evidence from infinitely many children. Our whole analysis in this chapter rests on the fact that even though infinitely many variables may be relevant to answering a query, the probability model is made up of local interactions, and each variable is still only connected to a small number of neighbors. We therefore disallow reference attributes to take on named instances as values.

We also disallow multi-valued attributes and number uncertainty in this language. However, that is only for convenience, so as not to include too many features in the language that are not really relevant to recursive models. Multi-valued attributes and number uncertainty are orthogonal to recursive models, and incorporating them into the analysis of this chapter does not pose special difficulties.

Because we now allow infinite dependency chains, we can relax the constraints placed on the global dependency graph described in Section 5.3. We still want to prevent cyclic dependencies, while allowing infinite dependencies. The global dependency graph can still be used for this purpose. Recall that a cycle in the global dependency graph can represent a cyclic chain of dependencies or an infinite chain of dependencies. If a cycle in the graph is detected, we check to see which kind of dependency chain it represents. If the cycle contains a node $I.A$ where I is a named instance, it represents a cyclic dependency chain. If not, it may represent either kind of dependency chain. For example, the cycle [Person.Happy, Person.Mother, Person.Happy] represents the infinite dependency chain resulting from the dependency of a person's happiness on their mother's happiness. On the other hand, the cycle [Person.Happy, Person.Mother, Person.Healthy, Person.Child, Person.Happy] represents the cyclic dependency resulting from the dependence of a mother's health and her child's happiness on each other.

We can detect which type of dependency chain a particular cycle represents by making the following observation. If a cycle does not contain a complex attribute, then the cycle represents a dependency chain among attributes of the same object, so it represents a cyclic dependency chain. If the cycle contains complex attributes, it involves dependencies among attributes of multiple objects, and only represents a cyclic dependency chain if it ends up back in the object in which it started. That can

only happen if each time the chain moves from one object to another via a complex attribute, that move is later cancelled via an inverse attribute.

We can therefore test whether a cycle represents a cyclic or infinite dependency chain by the following procedure. Let the cycle consist of a sequence of nodes $C_1.A_1, \dots, C_n.A_n$. First, all nodes $C_i.A_i$ where A_i is simple are removed from the sequence. Then, if the resulting sequence contains a pair of consecutive nodes $C_i.A_i, C_j.A_j$, where A_j *inverse-of* A_i , the pair is removed. This operation is repeated until no more such pairs exist. If the resulting sequence is empty, the cycle represents a cyclic dependency chain, otherwise it represents an infinite dependency chain. In the first example above, the sequence after removing simple attributes consists of the single node `Person.Mother`. The sequence cannot be reduced, so the cycle represents an infinite dependency chain. In the second example, the sequence of complex attributes is `[Person.Mother, Person.Child]`. Since `Child` *inverse-of* `Mother`, the pair of nodes can be removed, resulting in an empty sequence, and the cycle represents a cyclic dependency chain.

7.3 Measure-Theoretic Semantics

For the semantics of recursive models, we use a measure-theoretic approach that is similar to that for relational probability models. The set of possible worlds $\Omega_{\mathcal{K}}$ and event space $\mathcal{E}_{\mathcal{K}}$ are defined in the same way as before. The difference between our approach here and before is that we can no longer specify exactly what the measure over $\langle \Omega_{\mathcal{K}}, \mathcal{E}_{\mathcal{K}} \rangle$ should be. Instead, we specify constraints on the measure. These constraints are expressed through *Bayesian network fragments* that describe the probability model.

Each of these BN fragments describes a conditional probability distribution over some set of nodes given their parents. In order for the fragments to represent the probability model in a coherent way, we require that the set of variables contained in the fragment be *self-contained*, in the sense that every dependency chain between two variables in the set should be fully contained in the set. If a set of nodes is not self-contained, the fragment containing those nodes does not capture all the dependencies

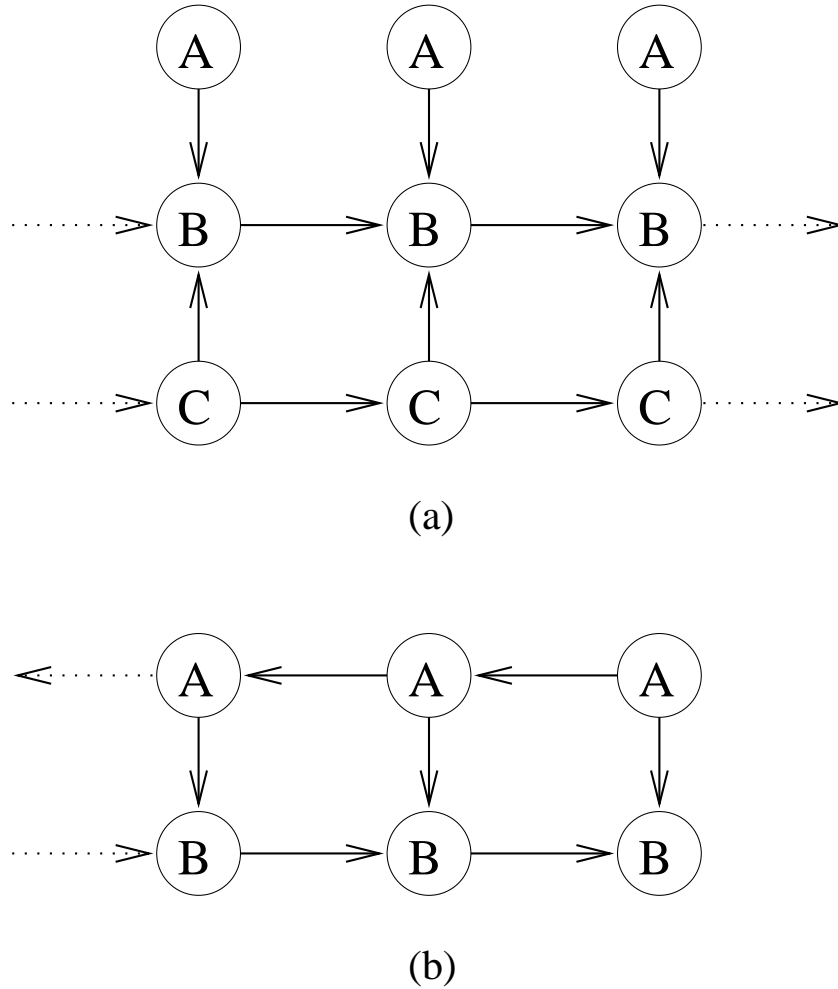


Figure 7.1: Self-contained sets: (a) Normal case. (b) Pathological case.

between variables in the set, and cannot be expected to capture the probabilistic relationships between the variables in a faithful manner.

Definition 7.3.1: A set \mathbf{X} of variables is *self-contained* if there is no dependency chain from $X_1 \in \mathbf{X}$ to $X_2 \in \mathbf{X}$ that passes through a node $Y \notin \mathbf{X}$. ■

The notion of self-containedness is reasonable and natural, and in fact all the examples we have considered so far, such as dynamic Bayesian networks, genetic models and stochastic context free grammars have natural self-contained sets of nodes. For example, Figure 7.1(a) shows a dynamic Bayesian network with three variables

in each time slice. The set of variables in any time slice is self contained, as is the set of variables in any adjacent sequence of time slices.

However, there are pathological examples of models in which some finite set of variables is not contained in any finite self-contained set. Such a model is illustrated in Figure 7.1(b). This example consists of a single class C , with two complex attributes A_1 and A_2 , and two simple attributes B_1 and B_2 . B_1 depends on $A_1.B_1$, while B_2 depends on B_1 and on $A_2.B_2$. The problem arises because A_1 and A_2 are inverses of each other. For an instance I of C , there is a dependency chain of the form

$$I.B_1 \rightarrow \dots \rightarrow I.A_1. \overset{(n \text{ times})}{\dots} .B_1 \rightarrow I.A_1. \overset{(n \text{ times})}{\dots} .B_2 \rightarrow \dots \rightarrow I.B_2,$$

for all n . Since there are infinitely many and arbitrarily long dependency chains from $I.B_1$ to $I.B_2$, the model can have no finite self-contained sets containing these two variables.

Definition 7.3.2: A recursive probabilistic KB \mathcal{K} is *normal* if, for every finite set of variables \mathbf{X} of \mathcal{K} , \mathbf{X} , there is a finite set of variables \mathbf{Y} such that $\mathbf{X} \subseteq \mathbf{Y}$ and \mathbf{Y} is self-contained. We define \mathbf{X}^* to be

$$\cap \{ \mathbf{Y} : \mathbf{X} \subseteq \mathbf{Y} \text{ and } \mathbf{Y} \text{ is self-contained.} \quad \blacksquare$$

Remark 7.3.3: It is clear that \mathbf{X}^* is self-contained, since the intersection of any two self-contained sets is self-contained. Also, if $\mathbf{X}_1 \subseteq \mathbf{X}_2$, then $\mathbf{X}_1^* \subseteq \mathbf{X}_2^*$. \blacksquare

We define the probabilistic semantics by requiring that a probability measure over the set of basic variables respect the network fragments for all finite self-contained sets of variables. This is a reasonable definition, and makes good sense for the examples we considered in the introduction, even though it is not particularly useful in pathological cases. First we need to define a network fragment relative to a set of variables.

Definition 7.3.4: Let \mathbf{X} be a finite set of variables. The *Bayesian network fragment* relative to \mathbf{X} , denote $\mathcal{B}_{\mathbf{X}}$, consists of $\mathbf{X} \cup Pa[\mathbf{X}]$. For each $X = I.\sigma.A \in \mathbf{X}$, the parents of X in $\mathcal{B}_{\mathbf{X}}$ are the nodes Y such that $X \leftarrow Y$, and the CPF of X is derived

from the CPF of A , as described in Definition 5.4.5. Nodes in $Pa[\mathbf{X}] - \mathbf{X}$ are called the *roots* of $\mathcal{B}_{\mathbf{X}}$, and have no parents or CPF in $\mathcal{B}_{\mathbf{X}}$.³ ■

A BN network fragment does not define a probability distribution, because no CPFs are provided for the root nodes, but it does specify a conditional probability distribution over \mathbf{X} given the roots. We require that any model of a recursive probabilistic KB satisfy the conditional distributions so defined, whenever \mathbf{X} is a finite self-contained set of variables.

Definition 7.3.5: A probability measure μ over $\langle \Omega_{\mathcal{K}}, \mathcal{E}_{\mathcal{K}} \rangle$ is a *model* for \mathcal{K} if, for every finite self-contained set of variables \mathbf{X} in \mathcal{K} ,

$$\mu(\mathbf{X} \mid \mathbf{Y}_{\mathbf{X}}) = \mathcal{B}_{\mathbf{X}}(\mathbf{X} \mid \mathbf{Y}_{\mathbf{X}}),$$

where $\mathbf{Y}_{\mathbf{X}}$ is $Pa[\mathbf{X}] - \mathbf{X}$. ■

For non-recursive probability models, we used a set of flat BNs to define the probability distributions over assignments of values to basic variables, and generated the measure μ over $\Omega_{\mathcal{K}}$ from them. Lemma 5.4.6 shows that the μ defined in this way does in fact satisfy the constraints of Definition 7.3.5, while Theorem 5.4.10 shows that it is the only measure satisfying these constraints. Thus the semantics presented here for recursive models is an extension of the semantics of Chapter 5.

We will now prove that for recursive probability models, a measure satisfying Definition 7.3.5 always exists. However, unlike in Chapter 5, we can no longer guarantee uniqueness of such a measure.

The basic idea of the proof is as follows. We wish to show that there exists an additive function over basic events, that can then be extended to a probability measure. We grow this additive function by considering larger and larger self-contained sets of variables. We show that for any pair of variables \mathbf{X}_1 and \mathbf{X}_2 , with $\mathbf{X}_1 \subset \mathbf{X}_2$, it is possible to define the function over events mentioning the variables in \mathbf{X}_1 in such a

³The term “roots” used here does not denote the set of nodes with no parents in $\mathcal{B}_{\mathbf{X}}$. A node in \mathbf{X} that happens to have no parents is not considered a root; it has a CPF, and its probability model is defined by $\mathcal{B}_{\mathbf{X}}$. Rather, a root is a node that is present only because it is the parent of some node in \mathbf{X} , and it does not have an CPF.

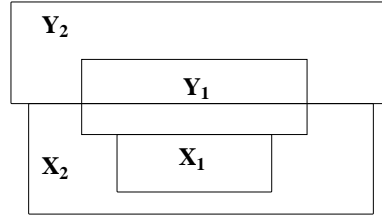


Figure 7.2: Illustrative figure for Lemma 7.3.7.

way that it is consistent with the BN fragment for \mathbf{X}_2 . We then use compactness to show that we can therefore define the function over events mentioning variables in \mathbf{X}_1 in such a way that it is simultaneously compatible with the BN fragment for every set containing \mathbf{X}_1 . The function defined in this way is additive and agrees with all the BN fragments, so can be used to generate the required probability measure.

For the key proof step we require the following lemmas. Lemma 7.3.6 states simply that if \mathbf{X} is self-contained and Y is a root of $\mathcal{B}_{\mathbf{X}}$, it cannot be a descendant of any $X \in \mathcal{B}_{\mathbf{X}}$. Lemma 7.3.7 uses this fact to show that as we go from one self-contained set to a larger self-contained set, the roots of the smaller set d-separate the smaller set from the roots of the larger set. This lemma is very useful, and we shall use it a number of times. This is used in turn to prove Lemma 7.3.9, which states that any distribution over the larger set of variables that is compatible with the BN fragment for the larger set marginalizes down to a distribution over the smaller set of variables that is compatible with the BN fragment for the smaller set. It is this property that allows us to grow the function over basic events in a consistent manner.

Lemma 7.3.6: *Let \mathbf{X} be a finite self-contained set of variables, and Y a root of $\mathcal{B}_{\mathbf{X}}$. Then Y cannot be a descendant of any $X \in \mathbf{X}$.*

Proof: Suppose not. Then there exists a dependency chain from some $X_1 \in \mathbf{X}$ to Y . But $Y \in Pa[\mathbf{X}]$, so it is a parent of some $X_2 \in \mathbf{X}$. Therefore there is a dependency chain from X_1 to X_2 that goes through $Y \notin \mathbf{X}$, contrary to assumption that \mathbf{X} is self-contained. ■

Lemma 7.3.7: *Let \mathbf{X}_1 and \mathbf{X}_2 be two finite self-contained sets of variables, with*

$\mathbf{X}_1 \subseteq \mathbf{X}_2$. Let \mathbf{Y}_1 and \mathbf{Y}_2 be the roots of $\mathcal{B}_{\mathbf{X}_1}$ and $\mathcal{B}_{\mathbf{X}_2}$ respectively. Then \mathbf{X}_1 is d -separated from \mathbf{Y}_2 by \mathbf{Y}_1 in $\mathcal{B}_{\mathbf{X}_2}$.

Figure 7.2 illustrates Lemma 7.3.7. The figure shows each of the four sets \mathbf{X}_1 , \mathbf{X}_2 , \mathbf{Y}_1 and \mathbf{Y}_2 as rectangles. We can consider all probabilistic dependencies to flow vertically downwards. \mathbf{X}_1 and \mathbf{Y}_1 are disjoint, but every dependency into \mathbf{X}_1 from outside \mathbf{X}_1 must come from \mathbf{Y}_1 , so we draw the two rectangles adjacent, with the bottom border of \mathbf{Y}_1 completely covering the top border of \mathbf{X}_1 . By Lemma 7.3.6, we are justified in not drawing part of \mathbf{Y}_1 underneath \mathbf{X}_1 to allow dependencies to flow in the other direction. The situation is similar for \mathbf{X}_2 and \mathbf{Y}_2 . Also, \mathbf{X}_2 contains \mathbf{X}_1 , while $\mathbf{Y}_2 - \mathbf{X}_2$ contains $\mathbf{Y}_1 - \mathbf{X}_2$. It is intuitively clear from the figure that there can be no path from \mathbf{Y}_2 down to \mathbf{X}_1 that does not pass through \mathbf{Y}_1 , and that \mathbf{X}_1 has no descendants in \mathbf{Y}_1 or \mathbf{Y}_2 . We prove the lemma formally as follows.

Proof: Suppose not. Then the set S of nodes $X \in \mathbf{X}_1$ such that there is an active path from \mathbf{Y}_2 to X is non-empty. Since \mathbf{X} is finite, there must be some node Z such that none of its parents are in S . Let Z be such a node. Suppose the path from \mathbf{Y}_2 to Z has an edge entering Z . The predecessor of Z in the path cannot be in \mathbf{X} , or else it would be a parent of Z in S , contrary to definition of Z . So the predecessor of Z in the path must be a node $Y \in \mathbf{Y}_1$. But then the path is blocked at Y . We are left with the possibility that path has an edge leaving Z . By Lemma 7.3.6, no node in \mathbf{Y}_2 can be a descendant of Z (since $Z \in \mathbf{X}_1 \subseteq \mathbf{X}_2$), so the path must have converging arrows at some node W descended from Z . But, again by Lemma 7.3.6, since W and all its descendants are descended from Z , none of those nodes can be in \mathbf{X}_1 . So the path is blocked at W . We conclude that there can be no active path from \mathbf{Y}_2 to Z , contrary to assumption. ■

The proof of Lemma 7.3.9 and Theorem 7.3.10 rely on a mapping defined for the BN fragment corresponding to a set of variables \mathbf{X} . This mapping maps probability distributions over the roots of the fragment to probability distributions over \mathbf{X} . We use the notation $\Delta_{\mathbf{X}}$ to denote the space of probability distributions over \mathbf{X} . It is a subset of R^n , where n is the number of values in $Val[\mathbf{X}]$. $\Delta_{\mathbf{X}}$ consists of points $\mathbf{p} \in R^n$ satisfying the constraints that $p_i \geq 0$ and $\sum_{i=1}^m p_i = 1$. $\Delta_{\mathbf{X}}$ is a closed and

bounded, and therefore compact, subset of R^n .

Definition 7.3.8: Let \mathbf{X} be a finite set of nodes, and \mathbf{Y} the roots of $\mathcal{B}_{\mathbf{X}}$. We define the mapping $F_{\mathbf{X}}$ from $\Delta_{\mathbf{Y}}$ to $\Delta_{\mathbf{X}}$ as follows. For each $p \in \Delta_{\mathbf{Y}}$,

$$F_{\mathbf{X}}(p) = \sum_{\mathbf{y} \in \text{Val}[\mathbf{Y}]} p(\mathbf{y}) \mathcal{B}_{\mathbf{X}}(\mathbf{X} \mid \mathbf{y}). \quad \blacksquare$$

Lemma 7.3.9: Let \mathbf{X}_1 and \mathbf{X}_2 be finite self-contained sets of variables, with $\mathbf{X}_1 \subseteq \mathbf{X}_2$, and let \mathbf{Y}_1 and \mathbf{Y}_2 be the roots of $\mathcal{B}_{\mathbf{X}_1}$ and $\mathcal{B}_{\mathbf{X}_2}$ respectively. If $p \in \Delta_{\mathbf{X}_2}$ is in the image of $\Delta_{\mathbf{Y}_2}$ under $F_{\mathbf{X}_2}$, then $\sum_{\mathbf{X}_2 - \mathbf{X}_1} p$ is in the image of $\Delta_{\mathbf{Y}_1}$ under $F_{\mathbf{X}_1}$.

Proof: If $p \in \Delta_{\mathbf{X}_2}$ is in the image of $\Delta_{\mathbf{Y}_2}$ under $F_{\mathbf{X}_2}$, there is some $q \in \Delta_{\mathbf{Y}_2}$ such that

$$p(\mathbf{X}_2) = \sum_{\mathbf{y}_2} q(\mathbf{y}_2) \mathcal{B}_{\mathbf{X}_2}(\mathbf{X}_2 \mid \mathbf{y}_2).$$

Let p' be $\sum_{\mathbf{X}_2 - \mathbf{X}_1} p$. We have

$$\begin{aligned} p'(\mathbf{x}_1) &= \sum_{\mathbf{x}_2 \in \text{Val}[\mathbf{X}_2 - \mathbf{X}_1]} p(\mathbf{x}_1, \mathbf{x}_2) \\ &= \sum_{\mathbf{y}_2} q(\mathbf{y}_2) \sum_{\mathbf{x}_2} \mathcal{B}_{\mathbf{X}_2}(\mathbf{x}_1, \mathbf{x}_2 \mid \mathbf{y}_2) \\ &= \sum_{\mathbf{y}_2} q(\mathbf{y}_2) \mathcal{B}_{\mathbf{X}_2}(\mathbf{x}_1 \mid \mathbf{y}_2) \\ &= \sum_{\mathbf{y}_2} q(\mathbf{y}_2) \sum_{\mathbf{y}_1 \in \text{Val}[\mathbf{Y}_1]} \mathcal{B}_{\mathbf{X}_2}(\mathbf{y}_1 \mid \mathbf{y}_2) \mathcal{B}_{\mathbf{X}_2}(\mathbf{x}_1 \mid \mathbf{y}_1, \mathbf{y}_2) \\ &= \sum_{\mathbf{y}_2} q(\mathbf{y}_2) \sum_{\mathbf{y}_1} \mathcal{B}_{\mathbf{X}_2}(\mathbf{y}_1 \mid \mathbf{y}_2) \mathcal{B}_{\mathbf{X}_2}(\mathbf{x}_1 \mid \mathbf{y}_1) \quad (\text{by Lemma 7.3.7}) \\ &= \sum_{\mathbf{y}_2} q(\mathbf{y}_2) \sum_{\mathbf{y}_1} \mathcal{B}_{\mathbf{X}_2}(\mathbf{y}_1 \mid \mathbf{y}_2) \mathcal{B}_{\mathbf{X}_1}(\mathbf{x}_1 \mid \mathbf{y}_1) \\ &= \sum_{\mathbf{y}_1} \sum_{\mathbf{y}_2} q(\mathbf{y}_2) \mathcal{B}_{\mathbf{X}_2}(\mathbf{y}_1 \mid \mathbf{y}_2) \mathcal{B}_{\mathbf{X}_1}(\mathbf{x}_1 \mid \mathbf{y}_1) \\ &= F_{\mathbf{X}_1}(\sum_{\mathbf{y}_2} q(\mathbf{y}_2) \mathcal{B}_{\mathbf{X}_2}(\mathbf{y}_1 \mid \mathbf{y}_2)). \end{aligned}$$

Therefore p' is in the image of $\Delta_{\mathbf{Y}_1}$ under $F_{\mathbf{X}_1}$, as required. \blacksquare

We are now ready to prove our main result of this section.

Theorem 7.3.10: Let \mathcal{K} be a normal recursive probabilistic KB. There exists a probability measure μ over $\langle \Omega_{\mathcal{K}}, \mathcal{E}_{\mathcal{K}} \rangle$ that is a model for \mathcal{K} .

Proof: We begin by constructing an increasing sequence of finite self-contained sets that covers all the variables of \mathcal{K} . For $n > 0$, define

$$\mathbf{X}_n = \{X : \text{the length of } X \text{ is at most } n\}.$$

Let \mathbf{Z}_n denote the self-contained set \mathbf{X}_n^* , defined in Definition 7.3.2. \mathbf{Z}_n is finite since \mathcal{K} is normal, and self-contained by Remark 7.3.3. By the same remark, $\mathbf{Z}_1 \subseteq \mathbf{Z}_2 \subseteq \dots$.

Next, for each \mathbf{Z}_n , we define a set of probability distributions over \mathbf{Z}_n that are consistent with the BN fragment $\mathcal{B}_{\mathbf{Z}_n}$, for some distribution over the roots of $\mathcal{B}_{\mathbf{Z}_n}$. Let \mathbf{Y}_n be the roots of $\mathcal{B}_{\mathbf{Z}_n}$. Define the set $S_n \subseteq \Delta_{\mathbf{Z}_n}$ to be the image of $\Delta_{\mathbf{Y}_n}$ under the mapping $F_{\mathbf{Z}_n}$. S_n is the continuous image of the compact set $\Delta_{\mathbf{Y}_n}$, so it is compact. S_n is obviously also non-empty, since it is the image of a non-empty set.

Next, we use compactness to show that for each \mathbf{Z}_n , there is some non-empty subset of S_n of distributions that are consistent with some distribution in S_m for every $m \geq n$. For $m \geq n$, define the set $T_n^m \subseteq \Delta_{\mathbf{Z}_n}$ to be the image of S_m under the marginalization operator $\sum_{\mathbf{Z}_m - \mathbf{Z}_n}$. We show the following:

1. T_n^m is closed. This is clear, since it is the continuous image of a compact set, and therefore compact, and therefore closed.
2. T_n^m is non-empty. This is obvious, since it is the image of a non-empty set.
3. $T_n^m \subseteq S_n$. This follows immediately from Lemma 7.3.9.
4. If $m_1 < m_2$, $T_n^{m_1} \supseteq T_n^{m_2}$. This follows from point 3, as follows. $T_n^{m_2}$ is the image of S_{m_2} under the marginalization operator $G_2 = \sum_{\mathbf{Z}_{m_2} - \mathbf{Z}_n}$, which is the composition of the operators $H = \sum_{\mathbf{Z}_{m_2} - \mathbf{Z}_{m_1}}$ and $G_1 = \sum_{\mathbf{Z}_{m_1} - \mathbf{Z}_n}$. Since the image of H is a subset of S_{m_1} by point 3, and $T_n^{m_1}$ is the image of S_{m_1} under G_1 , it follows that $T_n^{m_1} \supseteq T_n^{m_2}$.

It follows, therefore, that $(T_n^m)_{m=n}^\infty$ is a decreasing sequence of closed, non-empty subsets of S_n . Define T_n to be $\cap_{m=n}^\infty T_n^m$. By compactness of S_n , T_n is a non-empty subset of S_n .

Each T_n is a set of distributions over \mathbf{Z}_n such that for every p in T_n , p is consistent with the BN fragment \mathbf{Z}_n for some distribution q over \mathbf{Y}_n , and furthermore, for every $m \geq n$, p is equal to $\sum_{\mathbf{Z}_m - \mathbf{Z}_n} p'$ for some $p' \in T_n$. The final stage of the construction is to use the T_n to define a function F over basic events, as follows.

We construct a sequence of distributions F_i over \mathbf{Z}_i as follows. For F_1 , choose any element of $T_{\mathbf{Z}_1}$. For F_i with $i > 1$, choose an element of $T_{\mathbf{Z}_i}$ such that $F_{i-1} = \sum_{\mathbf{Z}_i - \mathbf{Z}_{i-1}} F_i$. Since $F_{i-1} \in T_{\mathbf{Z}_{i-1}}$, such a choice must exist. For $i < j$, we have inductively that $F_i = \sum_{\mathbf{Z}_j - \mathbf{Z}_i} F_j$. It follows that we can define the function F over basic events unambiguously by setting $F(E) = F_i(E)$ for any i such that all variables mentioned by E are in \mathbf{Z}_i .

It is easy to see that F is additive. Let E_1, \dots, E_n be a set of disjoint basic events such that their union E is a basic event. Let i be such that all variables mentioned in any of the E_i or in E are in \mathbf{Z}_i . Then, since F_i is a probability distribution,

$$F(E) = F_i(E) = F_i(\cup_{j=1}^n E_j) = \sum_{j=1}^n F_i(E_j) = \sum_{j=1}^n F(E_j).$$

Lemma 5.4.8, which states that no basic event is the infinitely countable disjoint union of basic events, still applies, so additivity is sufficient to ensure countable additivity. Therefore F extends to a measure μ over $\langle \Omega_{\mathcal{K}}, \mathcal{E}_{\mathcal{K}} \rangle$. It is easy to see that μ is a probability measure, since $\Omega_{\mathcal{K}}$ can be expressed as the event $[\bigvee_{x \in \text{Val}_{[X]}} X = x]$. Letting \mathbf{Z}_i be such that $X \in \mathbf{Z}_i$, $\mu(\Omega_{\mathcal{K}}) = F_i([\bigvee_{x \in \text{Val}_{[X]}} X = x]) = 1$.

Finally, μ satisfies the condition of Definition 7.3.5. Let \mathbf{X} be a finite self-contained set of variables, and \mathbf{W} be $Pa[\mathbf{X}_i] - \mathbf{X}_i$. Let \mathbf{Z}_i be such that $\mathbf{X} \cup \mathbf{W} \subseteq \mathbf{Z}_i$. $\mu(\mathbf{X} \cup \mathbf{W}) = F_i(\mathbf{X} \cup \mathbf{W})$, and $F_i \in T_i \subseteq S_i$, so there is some distribution q over \mathbf{Y}_i , such that

$$\mu(\mathbf{X} \cup \mathbf{W}) = \sum_{\mathbf{y} \in \text{Val}[\mathbf{Y}_i]} q(\mathbf{y}) \mathcal{B}_{\mathbf{Z}_i}(\mathbf{X} \cup \mathbf{W} \mid \mathbf{y}).$$

It follows that

$$\begin{aligned}
 \mu(\mathbf{X} \mid \mathbf{W}) &= \sum_{\mathbf{y}} q(\mathbf{y}) \mathcal{B}_{\mathbf{Z}_i}(\mathbf{X} \mid \mathbf{W}, \mathbf{y}) \\
 &= \sum_{\mathbf{y}} q(\mathbf{y}) \mathcal{B}_{\mathbf{Z}_i}(\mathbf{X} \mid \mathbf{W}) \quad (\text{by Lemma 7.3.7}) \\
 &= \mathcal{B}_{\mathbf{Z}_i}(\mathbf{X} \mid \mathbf{W}) \\
 &= \mathcal{B}_{\mathbf{X}}(\mathbf{X} \mid \mathbf{W}).
 \end{aligned}$$

Therefore μ is a model of \mathcal{K} , as required. ■

7.4 Approximate Inference

Exact inference is in general infeasible in recursive probability models, since it may require solving an infinite BN. However, one can consider approximate inference. A natural approach is to use an idea along the lines of the *localized partial evaluation (LPE)* algorithm of Draper and Hanks [23]. LPE is an algorithm for approximate inference in standard BNs. The basic idea is to identify some *active subset* of the nodes in the network. Information about nodes outside the active subset is summarized through $[0, 1]$ interval bounds on their probabilities. The bounds on the nodes outside the subset are then combined with the CPFs for the nodes inside the subset to compute bounds on the probabilities of the active nodes. This process results in the computation of valid interval bounds to the answer to any query.

LPE is an iterative, anytime algorithm. By considering larger and larger active subsets, better and better bounds are computed to the answer to the query. In the limit, the active subset is the entire network, and a point-valued answer is returned.

The extension of the LPE idea to recursive models is natural. An active set of nodes is constructed by growing the BN backward a certain number of generations from the query and evidence variables. The parents of this active set are summarized by $[0, 1]$ probability bounds. The bounds on the roots and the local probabilities of nodes in the active set are used to compute better and better bounds on the answer to the query.

Computing with interval-bounded rather than point-valued probabilities requires

a slight modification to the **VE** framework. The basic operands of the algorithm are now *interval-bound factors*, and the product and marginalization operations must now be redefined for interval-bound factors. An *interval-bound factor* f over variables \mathbf{X} is a function from $Val[X]$ to intervals $[p_1, p_2]$, where p_1 and p_2 are real numbers. A convenient way to represent an interval-bound factor f is as a pair of factors f_* and f^* , with $f_*(\mathbf{x}) \leq f^*(\mathbf{x})$ for all $\mathbf{x} \in Val[\mathbf{X}]$. A naive way to define the product of two interval bound factors $f = \langle f_*, f^* \rangle$ and $g = \langle g_*, g^* \rangle$ is to define the product h by $h_* = f_*g_*$ and $h^* = f^*g^*$. Similarly, $\sum_X f$ can be defined to be the factor g specified by $g_* = \sum_X f_*$ and $g^* = \sum_X f^*$.

These operations are certainly sound. That is, if the true value of f lies between f_* and f^* , and similarly for g , then the true value of fg lies between f_*g_* and f^*g^* . Likewise, the true value of $\sum_X f$ lies between $\sum_X f_*$ and $\sum_X f^*$. So propagating interval bounds on the answer to a query using these operations will certainly result in a valid bound on the answer, as long as the initial bounds are valid. However, these operations are very conservative, and the bounds on the product and marginals can be tightened considerably. The reason is that in addition to the bounds on the true values of f and g , we also have constraints on the values resulting from the fact that the factors correspond to CPFs, or are computed from CPFs. For example, if f is $P(Y | X)$, we have the constraint $\sum_y f(x, y) = 1$ for all x . So in fact $\sum_Y f$ is known to be 1 for all values of x . However, $f_*(x, y)$ may be less than $f(x, y)$, so $\sum_y f_*(x, y)$ may be some number α_x less than 1. Using the conservative marginalization operator will allow us only to derive that $\sum_Y f \geq \alpha_x$.

A better approach to computing with interval-bound probabilities was developed by Tessem [96], and used by Draper and Hanks in LPE. At the core of Tessem's approach is the following observation. Suppose we are in the process of eliminating variable X , which appears in factors f^1, \dots, f^n . Let the variables other than X mentioned by f^i be \mathbf{Y}^i . Suppose also, that we have the constraint that for all $\mathbf{y}^1 \in Val[\mathbf{Y}^1]$, $\sum_{x \in Val[X]} f_1(x, \mathbf{y}^1) = 1$. There is only one factor that has this property, since a variable appears as a parent in all but one of the factors in which it participates. We want to compute $g = \sum_X \prod_i f^i$, which is a factor over $\mathbf{Y} = \cup_i \mathbf{Y}^i$. So for each $\mathbf{y} \in Val[\mathbf{Y}]$, we want to compute lower and upper bounds on $g(\mathbf{y})$ from the given

interval bounds for the f^i , subject to the constraint. Write $p_x^i = f^i(x, \mathbf{y}^i)$ (\mathbf{y}^i being the projection of \mathbf{y} on \mathbf{Y}^i), $p_x = \prod_{i=2}^n p_x^i$ and $q_x = p_x^1$. Also, let p_{x*}^i and p_x^{i*} be the lower and upper bounds on p_x^i as specified by the interval bounds for f^i , and write $p_{x*} = \prod_{i=2}^n p_{x*}^i$, $p_x^* = \prod_{i=2}^n p_x^{i*}$. Our goal is to solve

$$\begin{array}{ll} \text{minimize (or maximize)} & \sum_x p_x q_x \\ \text{subject to} & p_{x*} \leq p_x \leq p_x^* \\ & p_{x*}^1 \leq q_x \leq p_x^{1*} \\ & \sum_x q_x = 1 \end{array}$$

This is a simple linear programming problem. Because of its special form, the minimization problem can be solved by the following procedure:

```

For all  $x$ , set  $q_x = 0$ .
 $S = \text{Val}[X]$ .
 $r = 1$ .
While  $r > 0$  do:
  Choose  $x \in S$  that minimizes  $p_{x*}$ .
  Set  $q_x = \min(r, p_x^{1*})$ .
   $S = S - \{x\}$ .
   $r = r - q_x$ .
Return  $\sum_x q_x p_{x*}$ .

```

The solution to the maximization problem is analogous.

In presenting our algorithm, we assume that the factor sums and products are sound operations on interval-valued factors, perhaps implemented as described by Tessem. The algorithm itself is independent of the actual specification of these operations.

The following theorem states that the approach outlined here to approximate inference in recursive probability models is *sound*. That is, for any recursive probabilistic KB \mathcal{K} , and any query on \mathcal{K} , if we expand the network backwards from the set of query and evidence variables, use interval bounds for the roots, and compute interval bounds for the answer to the query, then the answer to the query lies within

the bounds for any model of \mathcal{K} .

Theorem 7.4.1: *Let \mathbf{X} be a finite well-contained set of variables in a recursive probabilistic KB \mathcal{K} , and $\mathcal{B}_{\mathbf{X}}$ be the BN fragment over \mathbf{X} as described in Definition 7.3.4. Let \mathbf{Q} and \mathbf{E} be subsets of \mathbf{X} , and \mathbf{q} and \mathbf{e} assignments of values to \mathbf{Q} and \mathbf{E} respectively. Let \mathbf{Y} be the roots of $\mathcal{B}_{\mathbf{X}}$. If we use $[0, 1]$ bounds for the prior probability distributions over \mathbf{Y} , and compute interval bounds $[p_1, p_2]$ on $P(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e})$ using **VE** with valid operations on interval-bounds factors, then for any model μ of \mathcal{K} , $p_1 \leq \mu(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}) \leq p_2$.*

Proof: By Definition 7.3.5, for any value \mathbf{y} of the roots \mathbf{Y} , μ must satisfy $\mu(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}, \mathbf{Y} = \mathbf{y}) = P_{\mathbf{X}}(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}, \mathbf{Y} = \mathbf{y})$. Therefore μ satisfies

$$\mu(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}) = \sum_{\mathbf{y}} \mu(\mathbf{y}) P_{\mathbf{X}}(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}, \mathbf{Y} = \mathbf{y}).$$

Let \mathbf{y}_* and \mathbf{y}^* be the values of \mathbf{y} that minimize and maximize $P_{\mathbf{X}}(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}, \mathbf{Y} = \mathbf{y})$ respectively. Since $\sum_{\mathbf{y}} \mu(\mathbf{y}) = 1$, we have

$$P_{\mathbf{X}}(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}, \mathbf{Y} = \mathbf{y}_*) \leq \mu(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}) \leq P_{\mathbf{X}}(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}, \mathbf{Y} = \mathbf{y}^*).$$

Now, it is clear that composition of valid interval-bound operators yields a valid interval-bound operator. That is, if f_1, \dots, f_n is a set of point-valued factors, and g_1, \dots, g_n a set of interval-bound factors such that for each f_i , $g_{i*} \leq f_i \leq g_i^*$, and \otimes is an operation on factors composed of valid interval-bound operators, then $[\otimes(g_1, \dots, g_n)]_* \leq \otimes(f_1, \dots, f_n) \leq [\otimes(g_1, \dots, g_n)]^*$. Now, let the computed answer to $P(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e})$ from $\mathcal{B}_{\mathbf{X}}$ using **VE** with interval-bound operations be $[p_1, p_2]$. The operations performed by **VE** are the composition of valid interval-bound operations, and hence define a valid interval-bound operation \otimes . For each root Y_i , let f_i be the CPF for Y_i that assigns probability 1 to y_{i*} . Each f_i is within the interval bounds $[0, 1]$ used for Y_i . Therefore

$$p_1 \leq \otimes(f_1, \dots, f_n) = P_{\mathbf{X}}(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}, \mathbf{Y} = \mathbf{y}_*) \leq \mu(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}).$$

Similarly

$$p_2 \geq P_{\mathbf{X}}(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}, \mathbf{Y} = \mathbf{y}^*) \geq \mu(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}).$$

■

It also holds, as we show next, that as we expand larger and larger networks to solve a query, the bounds on the answer to the query as defined by the network do not get worse. In fact, if all CPF entries are positive, then the bounds will always improve.

Theorem 7.4.2: *Let \mathbf{X}_1 and \mathbf{X}_2 be two finite self-contained sets of variables, with $\mathbf{X}_1 \subseteq \mathbf{X}_2$, and let \mathbf{Y}_i be the roots of $\mathcal{B}_{\mathbf{X}_i}$. Let \mathbf{y}_{i*} and \mathbf{y}_i^* be the values of \mathbf{y}_i that minimize and maximize $P_{\mathbf{X}_i}(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}, \mathbf{Y}_i = \mathbf{y}_i)$ respectively. Let \mathbf{Q} and \mathbf{E} be subsets of \mathbf{X}_1 , and \mathbf{q} and \mathbf{e} assignments of values to \mathbf{Q} and \mathbf{E} respectively. Then*

$$\begin{aligned} P_{\mathbf{X}_1}(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}, \mathbf{Y}_1 = \mathbf{y}_{1*}) &\leq P_{\mathbf{X}_2}(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}, \mathbf{Y}_2 = \mathbf{y}_{2*}) \quad \text{and} \\ P_{\mathbf{X}_2}(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}, \mathbf{Y}_2 = \mathbf{y}_2^*) &\leq P_{\mathbf{X}_1}(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}, \mathbf{Y}_1 = \mathbf{y}_1^*). \end{aligned}$$

Furthermore, if $\mathbf{Y}_1 \subseteq \mathbf{X}_2$, the CPFs of all variables contain only positive entries, and $P_{\mathbf{X}_1}(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}, \mathbf{Y} = \mathbf{y}_{1*}) < P_{\mathbf{X}_1}(\mathbf{Q} = \mathbf{q} \mid \mathbf{E} = \mathbf{e}, \mathbf{Y} = \mathbf{y}_1^*)$, then the inequalities are strict.

Proof: By Lemma 7.3.7, \mathbf{Y}_2 is d-separated from $\mathbf{Q} \cup \mathbf{E}$ by \mathbf{Y}_1 in $\mathcal{B}_{\mathbf{X}_2}$. We therefore have

$$\begin{aligned} P_{\mathbf{X}_2}(\mathbf{q} \mid \mathbf{e}, \mathbf{y}_{2*}) &= \sum_{\mathbf{y}_1} P_{\mathbf{X}_2}(\mathbf{y}_1 \mid \mathbf{y}_{2*}) P_{\mathbf{X}_2}(\mathbf{q} \mid \mathbf{e}, \mathbf{y}_1, \mathbf{y}_{2*}) \\ &= \sum_{\mathbf{y}_1} P_{\mathbf{X}_2}(\mathbf{y}_1 \mid \mathbf{y}_{2*}) P_{\mathbf{X}_2}(\mathbf{q} \mid \mathbf{e}, \mathbf{y}_1) \\ &= \sum_{\mathbf{y}_1} P_{\mathbf{X}_2}(\mathbf{y}_1 \mid \mathbf{y}_{2*}) P_{\mathbf{X}_1}(\mathbf{q} \mid \mathbf{e}, \mathbf{y}_1) \\ &\geq P_{\mathbf{X}_1}(\mathbf{q} \mid \mathbf{e}, \mathbf{y}_{1*}). \end{aligned}$$

The \geq in the above sequence holds because $\sum_{\mathbf{y}_1} P_{\mathbf{X}_2}(\mathbf{y}_1 \mid \mathbf{y}_{2*}) = 1$, and because \mathbf{y}_{1*} minimizes $P_{\mathbf{X}_1}(\mathbf{Q} \mid \mathbf{E}, \mathbf{y}_1)$. If the roots of \mathbf{X} are in \mathbf{Y} , and all CPT entries are positive, $P_{\mathbf{X}_2}(\mathbf{y}_1^* \mid \mathbf{y}_{2*}) > 0$. If in addition $P_{\mathbf{X}_1}(\mathbf{q} \mid \mathbf{e}, \mathbf{y}_{1*}) < P_{\mathbf{X}_1}(\mathbf{q} \mid \mathbf{e}, \mathbf{y}_1^*)$, the \geq

can therefore be replaced by $>$, and the inequality is strict. The proof for the second inequality is similar. ■

Note that this theorem only applies to the optimal bounds defined by the network. Unfortunately, propagating the interval bounds using methods such as those of Tessem may not actually produce the optimal bounds. So, while the theorem says that considering larger and larger networks should in theory produce better and better bounds, it remains to be seen whether this holds true in practice.

Also note that the theorem does not say that the bounds eventually converge. In fact, it is possible that the lower and upper bounds may eventually converge to p_1 and p_2 with p_1 strictly less than p_2 . Indeed, the query may have a different answer in different models of the KB, and any number between p_1 and p_2 may be the correct answer for some measure μ that is a model of the KB.

An alternative to propagating interval bounds is to pick a particular probability distribution for the roots of \mathcal{B}_X — the uniform distribution is a reasonable choice. The query can then be solved in \mathcal{B}_X for that particular distribution, yielding an approximate solution to the answer to the query. The produced solution is approximate in the sense that it lies between the optimal interval bounds for \mathcal{B}_X , and the answer to the query in any model of the KB also lies within those bounds. Therefore, if the optimal bounds eventually converge for larger and larger KBs, the approximate solution will eventually converge to the unique true solution to the query. This approach is certainly easier to implement, since it does not require computing with interval-bound factors. However, it provides no guarantees as to the quality of the solution produced.

7.5 Structured Approximation Algorithms

7.5.1 Iterative SVE

Throughout this thesis, we have demonstrated the advantages of an inference algorithm that exploits the object structure of the domain. This leads us to believe that

the same should be true for an approximate inference algorithm for recursive probability models. We therefore develop the ideas of the previous section in the context of the object-based **SVE** algorithm. The version of **SVE** for recursive models is called **Iterative SVE** or **ISVE**. **Iterative SVE** takes one more argument than standard **SVE**, namely, the *order* of the desired approximation. A 0-th order approximation to the answer to the query is a $[0, 1]$ interval bound over the joint probability of the query variables, if we are propagating interval bounds, or a uniform distribution over the query variables, if we are computing a point-valued approximation. For $n > 0$, the n -th order approximate solution is computed using the $(n - 1)$ -th order solution for the recursively generated queries on complex attributes.

The **Iterative SVE** algorithm is closely based on the **SVE** algorithm developed in previous chapters. The version of **SVE** on which the following pseudocode is based reflects the particular language elements used in this chapter. Because we wish to allow OOBN style bindings to accomodate stochastic programs, as described in Example 7.1.4, we use the version of **SVE** that can handle both OOBN style and RPM style interfaces, as presented in Section 5.6. We also accomodate same-as statements, as described in Section 6.5.3, because these are also useful in defining stochastic functions, as in Example 7.1.4. As it turns out, caching makes interesting things happen in the iterative framework, so we present the cached version of the algorithm. In fact, the **UncachedISVE** subroutine contains almost no changes from the non-iterative version of **SVE**. However, this is the first time that the algorithm has been presented for this combination of language features, so we present the subroutine in full. Lines of code that are changed from non-iterative versions of the algorithm are marked with a ***.

```

*Algorithm IterativeSVE( $C, \sigma, \rho, e, D, n$ )
* If  $\langle C, \sigma, \rho, e, D \rangle$  is in Cache and
* Order( $\langle C, \sigma, \rho, e, D \rangle$ )  $\geq n$ 
*   Return Cache[ $\langle C, \sigma, \rho, e, D \rangle$ ].
  Else
*    $\langle \tau, f \rangle = \text{UncachedISVE}(C, \sigma, \rho, e, D, n)$ .
*   Cache[ $\langle C, \sigma, \rho, e, D \rangle$ ] =  $\langle \tau, f \rangle$ .

```

* $\text{Order}(\langle C, \sigma, \rho, e, D \rangle) = n.$

Return $\langle \tau, f \rangle.$

*Algorithm **UncachedISVE**(C, σ, ρ, e, D, n)

* If $n = 0$

* Return $\langle \emptyset, \text{ZeroOrderApproximation}(\sigma) \rangle.$

For each attribute A of C do

$\text{Needed}[A] = \emptyset.$

For each $\sigma \in \sigma \cup \rho$ do:

Let $\sigma = A.\sigma'.$

$\text{Needed}[A] = \text{Needed}[A] \cup \{\sigma\}.$

For each attribute A of C ,

in a bottom-up order consistent with $\mathcal{G}[C]$ do:

If $\text{Needed}[A] \neq \emptyset$

If A is simple

If A is a chain $\rho \in \rho$

Let a be the value assigned to A in e

$g_A = \text{CPF}_A[A = a].$

Else

$g_A = \text{CPF}_A.$

Else // A is complex

If not $A \in D$

If A has a reference attribute $\text{Ref}[A]$

For each $A.\sigma$ in $\text{Needed}[A]$ do:

For each value $B.\rho$ in $\text{Val}[\text{Ref}[A]]$ do:

$\text{Needed}[B] = \text{Needed}[B] \cup \{B.\rho.\sigma\}.$

$g_{A.\sigma}^{B.\rho}$ is the factor over $A.\sigma, B.\rho.\sigma$ and $\text{Ref}[A]$

such that $g_{A.\sigma}^{B.\rho}(x, y, z) = 1$ if $z \neq B.\rho,$

$g_{A.\sigma}^{B.\rho}(x, y, z) = 1$ if $x = y$ and $z = B.\rho$,
 $g_{A.\sigma}^{B.\rho}(x, y, z) = 0$ if $x \neq y$ and $z = B.\rho$.
 Else // A is not referentially uncertain
 Let C' be the range type of A .
 Let $\sigma' = \{\sigma' : A.\sigma' \in \text{Needed}[A]\}$.
 Let $\rho' = \{\rho' : A.\rho' \in \rho\}$.
 Let e' be the value in $\text{Val}[\rho']$,
 such that for each $\rho' \in \rho'$,
 the value assigned to ρ' in e'
 is the same as the value assigned to $A.\rho'$ in e
 Let $D' = \{B : B \text{ inverse-of } A\} \cup \{B : \text{there exists a binding } \Theta[A.B]\}$.
 * $\langle \tau', f_A \rangle = \text{IterativeSVE}(C', \sigma', \rho', e', D', n - 1)$.
 $g_A = \text{Rename}(f_A, \psi)$ where
 $\psi(B.\tau) = \tau$ if B inverse-of A ,
 $\psi(B.\tau) = \Theta[A.B].\tau$ if there is a binding $\Theta[A.B]$,
 $\psi(B.\tau) = A.B.\tau$ otherwise.
 For each $B.\sigma$ mentioned by f_A
 $\text{Needed}[B] = \text{Needed}[B] \cup \{\sigma\}$.

 $\tau = \cup_{D \in D} \text{Needed}[D]$.
 $f = \{g_A : \text{Needed}[A] \neq \emptyset\} \cup$
 $\{g_{A.\sigma}^{B.\rho} : A.\sigma \in \text{Needed}[A] \text{ and } B.\rho \in \text{Val}[\text{Ref}[A]]\}$.
 Let $\phi = \{\phi : \phi \text{ is mentioned by some } f \in f\} - (\sigma \cup \tau)$.
 For each $\phi \in \phi$ do
 Let g be $\{g \in f : g \text{ mentions } \phi\}$.
 $h_\phi = \prod g$.
 $k_\phi = \sum_\phi h_\phi$.
 $f = f - g \cup \{k\}$.
 $f = \prod f$.
 Return $\langle \tau, f \rangle$.

ISVE is called from a call to **SolveQuery**, which generates repeated ISVE calls

on the top-level object, requesting approximations of increasing order to the answer to the query. **SolveQuery** is an anytime algorithm, which can be run as long as desired, using any termination criteria to trade off running time versus quality of solution.

Algorithm SolveQuery(Q, E, e)

```

  Let  $\sigma$  be  $\{[I.A].\sigma' : I.A.\sigma' \in Q\}$ .
  Let  $\rho$  be  $\{[I.A].\rho' : I.A.\rho' \in E\}$ .
  Let  $e'$  be the value in  $Val[\rho]$ ,
    such that for each  $[I.A].\rho' \in \rho$ ,
    the value assigned to  $[I.A].\rho'$  in  $e'$ 
    is the same as the value assigned to  $I.A.\rho'$  in  $e$ .
*  $n = 1$ .
* Repeat as long as desired:
*  $\langle \emptyset, g \rangle = \mathbf{ISVE}(T, \sigma, \rho, e', \emptyset, n)$ .
   $h = \mathbf{Rename}(g, \psi)$ , where
     $\psi([I.A].\sigma) = I.A.\sigma$ .
   $f = \mathbf{Normalize}(h)$ .
*  $n = n + 1$ .
Return  $f$ .
```

In addition to the standard benefits of using SVE, namely exploiting small interfaces between objects and reoccurrence of the same type of object many times in a model, SVE provides an additional benefit in the framework of an iterative approximation algorithm. This benefit is that the results of earlier iterations can be used in later iterations. In particular, if a call to $\mathbf{ISVE}(C, \sigma, \rho, e, D, n)$ recursively generates a call to $\mathbf{ISVE}(C, \sigma, \rho, e, D, n - i)$, the $(n - i)$ -th approximate solution to the query will have been computed i iterations previously, and can simply be retrieved from the cache. Actually, if $i > 1$, there is no need to use the $(n - i)$ -th approximation, since the $(n - 1)$ -th approximation will also be in the cache. Therefore, the cache needs only to maintain the best computed approximate solution to the query, along with the order of the approximation. Whenever the n -th order approximation for a query

is needed, the algorithm first checks whether the cache contains at least an n -th order approximate answer; if it does, it returns it, otherwise it computes it and places it in the cache.

Example 7.5.1: Consider the genetic model from Example 7.1.3. Suppose the KB has a single named instance I of **Person**, and we wish to compute a probability distribution over I .Phenotype. We begin with a call to **SolveQuery**, with $Q = \{I$.Phenotype $\}$, and $E = e = \emptyset$. This results in a call to

$$\text{ISVE}(T, \{[I$$
.Phenotype $]\}, \emptyset, \emptyset, \emptyset, 1).$

The arguments $\langle T, \{[I$.Phenotype $]\}, \emptyset, \emptyset, \emptyset \rangle$ are not found in the cache, so a call is made to

$$\text{UncachedISVE}(T, \{[I$$
.Phenotype $]\}, \emptyset, \emptyset, \emptyset, 1).$

In this call, $[I$.Phenotype $]$ is needed, and it is the first attribute processed. Processing it results in $[I$.M-Chromosome $]$ and $[I$.P-Chromosome $]$ becoming needed. Processing $[I$.M-Chromosome $]$ results in $Needed[\text{Mother}]$ containing **Mother**.M-Chromosome and **Mother**.P-Chromosome. Similarly, after processing $[I$.P-Chromosome $]$, $Needed[\text{Father}]$ contains **Father**.M-Chromosome and **Father**.P-Chromosome. Now, processing I .Mother will result in a recursive call to

$$\text{ISVE}(\text{Person}, \{\text{M-Chromosome}, \text{P-Chromosome}\}, \emptyset, \emptyset, \emptyset, 0).$$

The arguments are not found in the cache, so a call to **UncachedISVE** is made with the same arguments. Since n is 0, a zero-order approximation is returned. For example, if we are computing with point-valued probabilities, this is the uniform distribution over the query variables. This approximation is then stored in the cache, together with the fact that its order is 0. Next, I .Father is processed, resulting in another call to

$$\text{ISVE}(\text{Person}, \{\text{M-Chromosome}, \text{P-Chromosome}\}, \emptyset, \emptyset, \emptyset, 0).$$

These arguments are now in the cache, and the associated order is 0, so the stored solution is immediately returned. This is the standard benefit of caching that we have exploited all along — the fact that the **Mother** and **Father** share the same probability model. All complex attributes of the top-level object have now been eliminated, so the variable elimination process is performed, and the 1st-order approximate answer to the query is returned.

In **SolveQuery**, we decide that we want to continue the computation, so we ask for a 2nd-order approximation. After looking in the cache, this results in a call to

$$\mathbf{UncachedISVE}(T, \{[I.\text{Phenotype}]\}, \emptyset, \emptyset, \emptyset, 2).$$

Eventually, *I.Mother* will be processed, resulting in a call to

$$\mathbf{ISVE}(\text{Person}, \{\text{M-Chromosome}, \text{P-Chromosome}\}, \emptyset, \emptyset, \emptyset, 1).$$

Although $\langle \text{Person}, \{\text{M-Chromosome}, \text{P-Chromosome}\}, \emptyset, \emptyset, \emptyset \rangle$ is now in the cache, the associated order is 0, so the stored answer is not used. Instead, a call is made to

$$\mathbf{UncachedISVE}(\text{Person}, \{\text{M-Chromosome}, \text{P-Chromosome}\}, \emptyset, \emptyset, \emptyset, 1).$$

Again, *Mother* will eventually be processed, and a call to

$$\mathbf{ISVE}(\text{Person}, \{\text{M-Chromosome}, \text{P-Chromosome}\}, \emptyset, \emptyset, \emptyset, 0)$$

will be generated. This time, the zero-order approximation in the cache can be used, so it is returned immediately. It will also be used when **Father** is processed. After eliminating variables, the **UncachedISVE** for **Mother** will return, and the new approximation for the arguments $\langle \text{Person}, \{\text{M-Chromosome}, \text{P-Chromosome}\}, \emptyset, \emptyset, \emptyset \rangle$ will be stored, along with its order, 1. This approximation is immediately reused when processing *I.Father*. After eliminating variables, the call to **UncachedISVE** for the top-level object will return, with the 2nd-order approximation to the answer to the query.

If we then continue, asking for a 3rd-order approximation to the answer to the query, we will eventually call **UncachedISVE**, asking for a 2nd-order approximation for $\langle \text{Person}, \{\text{M-Chromosome}, \text{P-Chromosome}\}, \emptyset, \emptyset, \emptyset \rangle$. This will in turn result in a request for the 1st-order approximation for $\langle \text{Person}, \{\text{M-Chromosome}, \text{P-Chromosome}\}, \emptyset, \emptyset, \emptyset \rangle$, which is found in the cache. In general, asking for the n -th order approximate answer to the top-level query will result in two calls to **UncachedISVE**, one for the top-level object, and one when processing *I.Mother*.

We see here the great savings resulting from exploiting structure in the iterative approximation algorithm. The amount of work done in each successive approximation is constant. This is in contrast to performing the inference in ever expanding flat BNs. Because each person has two parents, the size of the flat BN going back n generations is $O(2^n)$, so the amount of work done in successive iterations grows exponentially. ■

7.5.2 Analysis

A question that arises is whether the phenomenon in Example 7.5.1 is true in general. Does the amount of computation performed in successive iterations always become constant, after a certain number of iterations? The answer, in many cases, is yes. After a certain number of iterations in which the query expands, a *recursive fringe* is reached, in which the only queries asked at the fringe are queries that were asked earlier during the computation process. As a result, approximations of adequate order will always be found in the cache for the fringe queries, and no further work will need to be performed for them.

Whether or not a recursive fringe is reached depends on whether or not the KB contains OOBN-style bindings. We can prove that if a KB contains no OOBN-style bindings, a recursive fringe is always reached. We begin by defining some notation and terminology.

Definition 7.5.2: We let Q stand for the arguments $\langle C, \sigma, \rho, e, D \rangle$ to **SVE**, so that a call to **ISVE** is described by the pair $\langle Q, n \rangle$. We say that $Q' = \langle C, \sigma', \rho, e, D \rangle$ *subsumes* Q if $\sigma' \supseteq \sigma$. The set of input chains τ returned by the **ISVE** call on $\langle Q, n \rangle$ will be denoted τ_n^Q . For an **ISVE** call $\langle Q, n \rangle$ ($n > 0$), we let $\langle Q.A, n-1 \rangle$ denote

the recursive call on complex attribute A of C , with the appropriate arguments as generated by the algorithm. We say that $\langle Q, n \rangle$ *directly generates* $\langle Q.A, n-1 \rangle$. This notation is extended to attribute chains, so $\langle Q.A_1 \dots A_{m-1}.A_m, n-m \rangle$ is the recursive call directly generated by $\langle Q.A_1 \dots A_{m-1}, n-m+1 \rangle$ on attribute A_m . We say that $\langle Q, n \rangle$ *generates* $\langle Q.A_1 \dots A_m, n-m \rangle$. ■

Lemma 7.5.3: *Let \mathcal{K} be a KB with no OOBN-style bindings. For every query Q on \mathcal{K} there is a bound β , such that if $\langle Q_1, n-m \rangle$ is generated by $\langle Q, n \rangle$, the lengths of all chains in the arguments σ and ρ to Q_1 , and in $\tau_{n-m}^{Q_1}$ are bounded by β .*

Proof: We first prove the theorem under the assumption that \mathcal{K} contains no same-as statements or reference attributes. The bound β is then the maximum length of all chains in arguments to Q and of parents of simple attributes of \mathcal{K} .

The proof is by induction on n . The statement is clearly true for $n = 0$, since the only query generated by $\langle Q, 0 \rangle$ is $\langle Q, 0 \rangle$, and τ is empty for a zero-order approximation. Now, consider $n > 0$ and assume the theorem is true for all n' with $n' < n$. For $m = 0$, the statement is again clearly true. We will show that it is true for any $\langle Q.A, n-1 \rangle$. The result will then follow from the induction hypothesis for all $\langle Q.A.\sigma, n-m \rangle = \langle Q.A.\sigma, (n-1) - (m-1) \rangle$.

By the definition of the **ISVE** algorithm, the arguments to $Q.A$ are $\sigma' = \{\sigma' : A.\sigma' \in \text{Needed}[A]\}$, and $\rho' = \{\rho' : A.\rho' \in \rho\}$ where ρ is part of the argument to Q . The length of chains in ρ' is clearly bounded by β . We must show that the lengths of chains in $\text{Needed}[A]$ is bounded by β . A chain $A.\sigma$ can be placed in $\text{Needed}[A]$ for one of three reasons. First, $A.\sigma$ could be a query or evidence chain in σ or ρ , in which case its length is bounded by β by definition. Second, $A.\sigma$ could be the parent of some simple attribute, in which case its length is again bounded by β by definition. Finally, it could be that for some complex attribute D that was processed before A , $A.\sigma = \psi(B.\tau)$, for some $B.\tau$ in the return value $\tau_{n-1}^{Q.D}$ from the call to $\langle Q.D, n-1 \rangle$. We show by induction on the complex attributes A , in the order that they are processed by **ISVE**, that the length of a chain placed in $\text{Needed}[A]$ for the third reason is also bounded by β . The third case cannot apply to the first complex attribute processed, so the base case of the induction holds trivially. For the induction

step, we may assume by induction that for any B processed before A , the lengths of chains in $\tau_{n-1}^{Q,B}$ is bounded by β . Therefore the length of $B.\tau$ is bounded by β . Since the KB contains no OOBN-style bindings, B must be an inverse of D , and $A.\sigma = \tau$, so the length of $A.\sigma$ is also bounded by β , as required.

We now relax the assumption that the KB contains no same-as statements or reference attributes, but require that all same-as statements be stratified, as defined in Definition 6.5.1. We can therefore define the *potential length* of any chain σ as being the maximal length of the chain produced from σ by processing all possible same-as statements. We set the bound β now to be the maximum potential length of all chains in the query arguments and of all parents of attributes in \mathcal{K} . We repeat the argument above, showing that the potential length of any chain in $Needed[A]$ is bounded by β . It is clear that the length of a chain is bounded by its potential length, so this is sufficient. For the three reasons mentioned above that a chain could be in $Needed[\sigma]$, the same argument as above applies. There is now an additional reason why $A.\sigma$ could be in $Needed[A]$, namely, if $A.\sigma = A.\rho.\tau$, $B.\tau \in Needed[B]$, and there is a same-as statement B *same-as* $A.\rho$, or $A.\rho$ is a value in $Val[Ref[B]]$. But if the potential length of $B.\tau$ is bounded by β , the same is true for the potential length of $A.\rho.\tau$ by the definition of potential length, so the inductive argument presented above continues to hold. ■

It follows from the lemma that there are only finitely many distinct queries generated by any query Q . There are only finitely many attributes, so the number of chains whose length is bounded by β is also finite. A query Q is defined by subsets of these chains, so the number of such queries is also finite. This fact immediately suggests that the set of queries generated by Q stops growing at some point, and, therefore, that after a certain number of iterations, the set of queries generated becomes constant. In order to prove that this assertion is indeed true, we need to show that the work done in each iteration grows monotonically. Formally, we define a tree representing the work done in a particular iteration, and then show that the tree corresponding to one iteration can be embedded in the tree corresponding to the next iteration.

Definition 7.5.4: For any $\langle Q, n \rangle$, we define a tree T_n^Q as follows. The root of T_n^Q is Q . If Q' is a node of T_n^Q at depth m , Q' has a child $Q'.A$ for each $\langle Q'.A, n - m - 1 \rangle$ directly generated by $\langle Q', n - m \rangle$. The node Q' is annotated with the input chains $\tau_{n-m}^{Q'}$ returned by $\langle Q', n - m \rangle$. The edge from $\langle Q', n - m \rangle$ to $\langle Q'.A, n - m - 1 \rangle$ is annotated by the attribute A . The children of Q are listed in the order in which the attributes are processed by **ISVE**. ■

Lemma 7.5.5: *If Q^2 subsumes Q^1 , then there is a one-to-one mapping ϕ from the nodes in $T_n^{Q^1}$ to the nodes in $T_{n+1}^{Q^2}$,⁴ such that*

1. *the depth of Q' in $T_n^{Q^1}$ equals the depth of $\phi(Q')$ in $T_{n+1}^{Q^2}$;*
2. *$\phi(Q')$ subsumes Q' ;*
3. *$\tau_{n-m}^{Q'} \subseteq \tau_{n+1-m}^{\phi(Q')}$ (m being the depth of Q' in $T_n^{Q^1}$);*
4. *if there is an edge from Q' to Q'' in $T_n^{Q^1}$ annotated by A , there is an edge from $\phi(Q')$ to $\phi(Q'')$ in $T_{n+1}^{Q^2}$ annotated by A .*

Proof: By induction on n . For the base case, $n = 0$, and $T_0^{Q^1}$ contains only Q^1 . Define ϕ to take Q^1 to Q^2 . It is clear that conditions 1–4 are satisfied.

For the induction step, let Q_1^1, \dots, Q_m^1 be the children of Q^1 in $T_n^{Q^1}$, with the edge from Q^1 to Q_i^1 annotated by A_i . Let C be the first argument of Q^1 and Q^2 , representing the class on which they are defined. For each attribute A of C , let $Needed^1[A]$ be the set $Needed[A]$ produced when processing $\langle Q^1, n \rangle$, and $Needed^2[A]$ be $Needed[A]$ when processing $\langle Q^2, n+1 \rangle$. We claim that $Needed^1[A] \subseteq Needed^2[A]$ for all A . It will follow from this claim that for each of the Q_i^1 , $\langle Q^2, n+1 \rangle$ generates $\langle Q_i^2, n \rangle$ on attribute A_i , with Q_i^2 subsuming Q_i^1 . By the induction hypothesis, therefore, there is a one-to-one mapping ϕ_i from $T_{n-1}^{Q_i^1}$ to $T_n^{Q_i^2}$ satisfying conditions 1 and 2. We can therefore define a one-to-one mapping ϕ from $T_n^{Q^1}$ to $T_{n+1}^{Q^2}$ by defining ϕ to be the union of the ϕ_i , together with the assignment $\phi(Q^1) = \langle Q^2 \rangle$. It is clear that ϕ satisfies

⁴It turns out that the theorem is also true for $T_n^{Q^1}$ and $T_n^{Q^2}$, but we are interested in proving that the generated computation grows richer from one iteration to the next, which is why we compare $T_n^{Q^1}$ and $T_{n+1}^{Q^2}$.

conditions 1, 2 and 4 because the ϕ_i satisfy them Condition 3 is satisfied because it holds for all non-root nodes by the fact that the ϕ_i satisfy condition 3, and it holds for the root because $\tau_n^{Q^1} \subseteq \tau_{n+1}^{Q^2}$, which is true because $Needed^1[D] \subseteq Needed^2[D]$ for all D in the input set \mathbf{D} .

The claim is proved by induction on the attributes of C , following the order in which they are processed. A chain can be placed in $Needed[A]$ either at the beginning of the computation, while processing σ and ρ , or after processing some other attribute B . For the first attribute A processed, only the first reason can apply, and since Q^2 subsumes Q^1 , $Needed^1[A] \subseteq Needed^2[A]$. For the induction step, assume that $Needed^1[B] \subseteq Needed^2[B]$ for all attributes processed before A . In particular, if B is the complex attribute A_i annotating the edge from Q^1 to Q_i^1 , $\langle Q^2, n+1 \rangle$ will generate a query $\langle Q_i^2, n \rangle$ on attribute B , with Q_i^2 subsuming Q_i^1 . Therefore, by the main induction hypothesis, there is a mapping ϕ_i from $T_{n-1}^{Q_i^1}$ to $T_n^{Q_i^2}$ with the required properties. In particular ϕ_i must map Q_i^1 to Q_i^2 , and $\tau_{n-1}^{Q_i^1} \subseteq \tau_n^{Q_i^2}$. Now, a chain $A.\sigma$ could be placed in $Needed^1[A]$ in several ways. $A.\sigma$ could be placed in $Needed^1[A]$ at the beginning of the computation of $\langle Q^1, n \rangle$, in which case it will be placed in $Needed^2[A]$ at the beginning of the computation of $\langle Q^2, n \rangle$, because Q^2 subsumes Q^1 . $A.\sigma$ could be the parent of a simple attribute B such that $Needed^1[B]$ is non-empty, in which case $Needed^2[B]$ is also non-empty by induction hypothesis, so $A.\sigma$ is in $Needed^2[A]$. $A.\sigma$ could be in $\tau_{n-1}^{Q_i^1}$, in which case it is also in $\tau_n^{Q_i^2}$ as we have shown, and again $A.\sigma$ is in $Needed^2[A]$. Finally, $A.\sigma$ could be equal to $A.\rho.\tau$, with $B.\tau \in Needed^1[B]$ and $B.\rho$ a possible value of $Ref[B]$. Again by induction hypothesis $B.\tau \in Needed^2[B]$, so $A.\sigma \in Needed^2[A]$. We therefore see that $Needed^1[A] \subseteq Needed^2[A]$, and the claim is proven by induction. ■

Definition 7.5.6: If Q_1 and Q_2 are two nodes of a tree T , with the depth of Q_1 greater than the depth of Q_2 , and $Q_1 = Q_2$,⁵ we say that Q_1 is *shielded* by Q_2 in T . The *unshielded part* of a tree T consists of nodes of T that are reached by a path consisting of only unshielded nodes. ■

Consider processing a query Q . The computation performed in the n -th iteration

⁵Equality here means equality of the arguments $\langle C, \sigma, \rho, e, D \rangle$.

is represented by the tree T_n^Q . If Q_1 at depth m is shielded in T_n^Q , calling **ISVE** on $\langle Q_1, n - m \rangle$ will result in a cache hit for Q_1 , with order greater than $n - m$, so the result in the cache will be used. Calls to **UncachedISVE** are only made for unshielded queries.⁶

Theorem 7.5.7: *Let \mathcal{K} be a KB with no OOBN-style bindings, and let Q be a query on the top-level object of \mathcal{K} . There exists an N such that for all $n \geq N$, the unshielded part of T_n^Q equals the unshielded part of T_N^Q , and for each unshielded Q' at depth m in T_N^Q , $\tau_{n-m}^{Q'} = \tau_{N-m}^{Q'}$. The unshielded part of T_N^Q will be denoted T^Q , and is called the limit tree for Q .*

Proof: We define a tree T_∞^Q , representing the limit of the T_n^Q , as follows. For each complex attribute chain ρ on C , of length $m \geq 0$, we define Q_n^ρ and τ_n^ρ as follows. If T_n^Q contains a node Q' reached by a path annotated by ρ , Q_n^ρ is Q' and τ_n^ρ is $\tau_{n-m}^{Q'}$, otherwise Q_n^ρ and τ_n^ρ are \emptyset . It follows from Lemma 7.5.5 that if Q_n^ρ is not \emptyset , Q_{n+1}^ρ is not \emptyset and subsumes Q_n^ρ , and $\tau_n^\rho \subseteq \tau_{n+1}^\rho$. By Lemma 7.5.3, the size of the arguments to Q_n^ρ is bounded. Therefore the limits $Q^\rho = \lim_{n \rightarrow \infty} Q_n^\rho$ and $\tau^\rho = \lim_{n \rightarrow \infty} \tau_n^\rho$ exist, and the limits are achieved after a certain number n^ρ of iterations. T_∞^Q contains a path for each complex attribute chain on C , and the node reached by chain ρ of length m is Q^ρ . Let T^Q be the unshielded part of T_∞^Q . By Lemma 7.5.3, T^Q is finite. Let T' be T^Q , together with all children of nodes in T^Q . T' is also finite, and every leaf of T' either corresponds to a query on a class with no complex attributes, or is shielded in T' . Let N be the maximum, over all nodes Q^ρ in T' , of n^ρ . Then for all $n \geq N$, $Q_n^\rho = Q^\rho$, for all Q^ρ in T' , and the unshielded part of T_n^Q is T^Q . Also, since $N \geq n^\rho$, and the limit τ^ρ is achieved by the n^ρ -th iteration, $\tau_n^\rho = \tau_N^\rho$, or, letting $Q' = Q^\rho$, $\tau_{n-m}^{Q'} = \tau_{N-m}^{Q'}$. ■

Corollary 7.5.8: *If \mathcal{K} is a KB with no OOBN-style bindings, and Q is a query on \mathcal{K} , the asymptotic complexity of computing an n -th order approximation to the solution of Q using the **Iterative SVE** algorithm is linear in n .*

⁶A cache hit is also possible for an unshielded query $\langle Q_1, n - m \rangle$, if there is a query $\langle Q_2, n - m \rangle$ with $Q_1 = Q_2$ that is processed first. However, it is enough to show that the set of unshielded queries becomes constant to show that the number of calls to **UncachedISVE** is bounded by a constant.

Proof: Immediate from Theorem 7.5.7, since after the N -th iteration, the amount of work done in each successive iteration is constant. ■

Note that this result does *not* say that the number N of iterations that must be performed before the recursive fringe is reached is small, nor does it say that the amount of work done in each iteration after the recursive fringe is reached is small. In fact, both may be very large compared to the size of the KB.

The assumption that \mathcal{K} contains no OOBN-style bindings was used in the proof of Lemma 7.5.3 to limit the ways in which a chain could be placed in $Needed[A]$. In the presence of a binding $\Theta[D.B]$, $A.\rho$ could be $\psi(B.\tau) = \Theta[D.B].\tau$. Since it is possible that the length of $\Theta[D.B].\rho$ is greater than β , the proof breaks down. In fact, as the following example shows, bindings can cause the theorem to fail for quite natural models.

Example 7.5.9: Suppose we wish to define a stochastic function F that takes a string as argument, and returns a symbol. The first symbol in the string is returned with probability $1/2$, the second with probability $1/4$, and so on. $F(X)$ can be defined recursively: with probability $1/2$, it returns $S.First$, otherwise it returns $F(X.Rest)$. We can define the function in our language by defining a class F with three attributes. Attribute **Input** corresponds to the input of the function, and is of type string. Attribute **Call** is of type F , and corresponds to the recursive call. The argument to the recursive call is passed by the binding $\Theta[Call.Input] = Input.Rest$. Attribute **Output** is a simple attribute, ranging over the set of symbols. It depends on **Input.First** and **Call.Output**, and its CPF sets it to be equal to one or the other of the parents, each parent being chosen with probability $1/2$.

Now, suppose that we have a program in which a string is generated by some interesting SCFG, and then a symbol is extracted from the string by the function F . This program is encoded in a class that has two attributes, an attribute **The-String** whose type is the sentence symbol of the SCFG, and an attribute **The-Symbol**, whose type is F . The generated string is passed to F by the binding $\Theta[The-Symbol.Input] = The-String$.

Let Q denote the argument list $\langle F, \{Output\}, \emptyset, \emptyset, \{Input\} \rangle$. Let τ^n denote the set

of needed input chains returned by $\langle Q, n \rangle$. The only complex non-input attribute of F is **Call**, and when **Call** is processed, $Needed[\text{Call}] = \{\text{Output}\}$, so the only recursive call directly generated by $\langle Q, n \rangle$ is $\langle Q, n - 1 \rangle$. Let us examine what happens when we call **SolveQuery** to compute a probability distribution over **The-Symbol.Output**. The n -th iteration results first in a call to $\langle Q, n - 1 \rangle$ — this will return a set of chains beginning with **Input** on which **Output** depends, and a conditional probability distribution over **Output** given these chains. We will let τ^n denote the set

$$\{\tau : \text{Input}.\tau \text{ is in the set } \tau \text{ returned by } \langle Q, n - 1 \rangle\}.$$

On renaming, each of the chains **Input**. τ will be changed to **The-Sentence**. τ , and **The-Sentence**. τ will be placed in $Needed[\text{The-Sentence}]$. The set of query chains passed to the recursive call on **The-Sentence** is

$$\{\tau : \text{The-Sentence}.\tau \in Needed[\text{The-Sentence}]\},$$

which is equal to τ^n . Computation therefore proceeds by querying the SCFG for a distribution over τ^n .

What is the set τ^n ? For $n = 1$, it is the set of input chains resulting from a call to $\langle Q, 0 \rangle$. Since a zero-order approximation does not depend on any inputs, $\tau^1 = \emptyset$. For $n > 1$, the call to $\langle Q, n - 1 \rangle$ recursively generates a call to $\langle Q, n - 2 \rangle$. The solution to $\langle Q, n - 2 \rangle$ was computed in the $(n - 1)$ -th iteration, so will be found in the cache. The set of input chains required by $\langle Q, n - 2 \rangle$ is

$$\{\text{Input}.\tau : \tau \in \tau^{n-1}\}.$$

After renaming, using the binding $\Theta[\text{Call.Input}] = \text{Input.Rest}$, this becomes the set

$$\{\text{Input.Rest}.\tau : \tau \in \tau^{n-1}\}.$$

Since **Output** also depends directly on **Input.First**, the set of input chains required by

$\langle Q, n-1 \rangle$ is

$$\{\text{Input.First}\} \cup \{\text{Input.Rest}.\tau : \tau \in \tau^{n-1}\},$$

so τ^n is

$$\{\{\text{First}\} \cup \{\text{Rest}.\tau : \tau \in \tau^{n-1}\}.$$

It is easy to see by induction that

$$\tau^n = \bigcup_{i=0}^{n-2} \text{Rest}^i.\text{First}.$$

We see therefore that the set of inputs to the call on `The-Sentence` grows with each iteration. There is nothing pathological about this example. In fact, this is exactly the type of behavior we would want — in each iteration, we consider more and more of the string generated by the grammar. Perhaps the surprising thing is that this type of behavior cannot be generated without the OOBN-style bindings. ■

7.5.3 ISVE and Fixed-Point Equations

If the limit tree T^Q exists, then after a certain number N of iterations the algorithm starts performing exactly the same computations on each iteration. We can understand the algorithm as executing a set of fixed point equations. Following the notation of the proof of Theorem 7.5.7, we will label a node in T^Q by Q^ρ , we will denote the query chains argument to Q^ρ by σ^ρ , the class argument to Q^ρ by C^ρ , and the input chains return value by τ^ρ . In the n -th iteration, with $n > N$, the algorithm computes a factor f^ρ , which is a conditional distribution over σ^ρ given τ^ρ . We will let f_n^ρ denote the value of f^ρ computed in the n -th iteration.

Now, f_n^ρ is computed by performing a variable elimination computation over the CPFs for the simple attributes of C^ρ , and the factors computed from recursive queries on complex attributes of C^ρ . The query on $\rho.A$ may or may not result in a cache hit. If it is shielded, it will result in a cache hit. If the solution is retrieved from

the cache, it is equal to $f_{n-1}^{\rho'}$ for some other chain ρ' . If the query on $\rho.A$ does not result in a cache hit, its solution will be $f_n^{\rho.A}$. Therefore, f_n^ρ is defined as a sum-of-products of factors expression involving local CPFs of C^ρ , values of $f_{n-1}^{\rho'}$ computed in the previous iteration, and values of $f_n^{\rho.A}$ for complex attributes of C^ρ computed in the same iteration. Since the local CPFs are constant, we can write

$$f_n^\rho = F^\rho(\mathbf{f}_{n-1}^{\rho'}, \mathbf{f}_n^{\rho.A}), \quad (7.1)$$

where $\mathbf{f}_{n-1}^{\rho'}$ denotes the set of factors on all chains ρ' computed in the $(n-1)$ -th iteration, and $\mathbf{f}_n^{\rho.A}$ denotes the set of factors computed for complex attributes A in the n -th iteration. We have an equation of this form for each Q^ρ in T^Q . The values of f_n^ρ can be computed in a bottom up fashion. Values for leaves depend only on values of factors from the previous iteration. If a node is processed only after all its children, all factors on which it depends will be available at the time that it is processed. This is in fact exactly the way **ISVE** does things — the variable elimination process for a query happens only after factors have been computed for all its subqueries.

Each function F^ρ is a polynomial in its arguments.⁷ The reason it is a polynomial and not necessarily a linear function is because the same argument may appear multiple times in the sum-of-products expression. This will happen if different attributes have the same queries asked on them, as, for example, the **Mother** and **Father** attributes in Example 7.5.1. If we substitute the equations defining $\mathbf{f}_n^{\rho.A}$ in Equation 7.1, we obtain inductively that each F^ρ is a polynomial in $\mathbf{f}_{n-1}^{\rho'}$. We can therefore take all the F^ρ as defining a polynomial fixed-point equation F , defining \mathbf{f}_n^ρ in terms of \mathbf{f}_{n-1}^ρ . We can view the behavior of the **ISVE** algorithm, after the limit tree is reached, as computing successive iterations of this fixed point equation.

The fact that when a recursive fringe is reached, the solution to a query is defined by a polynomial fixed point equation, suggests that once the recursive fringe is found the algorithm can stop iterating and produce the fixed point equations explicitly.

⁷To be precise, it is only a polynomial if its arguments are point-valued probabilities, because then each element in the result is defined in terms of sums and products of the arguments. If the arguments are interval-bound probabilities, then the operations used are interval-bound operations which are not exactly sums and products.

These can then be solved directly, using a standard mathematical computing package such as Mathematica [99]. Detecting whether a recursive fringe is found is easy. At the beginning of each iteration a **Changed** flag is set to *False*. The **Changed** flag is set to *True* for one of two reasons. The first is obvious: **ISVE** is called for a query that is not in the cache. In that case it is clear that a recursive fringe has not yet been reached. The second situation is when **ISVE** is called on a query that is in the cache, but the order of the cached result is lower than the order of the desired approximation, and the set of input chains τ returned by the call to **UncachedISVE** is larger than the set in the cache. Although a new query is not asked in this situation, enlarging the set of input chains changes the structure of the computation. The number of variables in the fixed point equation depends on the number of input chains, so the fixed point equation should not be defined until the set of input chains converges.

If, at the end of an iteration, the **Changed** flag is still *False*, we know a recursive fringe has been reached. At that point, a symbolic variable is created for each of the unshielded queries in the query tree.⁸ The symbolic variable is then stored in the cache as the solution to the query. One last iteration is then performed, resulting in **UncachedISVE** being executed once again for each of these queries. For a query on the recursive fringe, the symbolic variable representing its solution is returned immediately. The variable elimination phase is performed symbolically.⁹ That is, the solution to the query is defined as the expression $\sum_{\phi} \prod_{f \in \mathbf{f}} f$. Some of the f will be numeric CPFs, while others will be symbolic expressions. In general, each factor entry is a polynomial in the symbolic expressions, and products and marginalizations of factors are defined in terms of products and sums of polynomial expressions. The result is a single factor, defining the conditional probability distribution over σ given τ as a polynomial expression in the symbolic variables. This expression is then returned as the solution to the query. At the conclusion of the extra iteration, we will have in the cache a symbolic expression defining the solution to each query. These expressions

⁸The set of unshielded queries can be detected by marking each query in the cache with the number of the last iteration for which it was solved. All queries that were solved in the iteration when **Changed** ended up *False* are unshielded.

⁹The process is similar to Li and D'Ambrosio's Symbolic Probabilistic Inference (SPI) algorithm [62].

define a system of fixed point equations, which can then be solved directly.

7.6 Conclusion

In this chapter, we developed a powerful extension to our relational probabilistic modeling language, by allowing the language to express recursive models. As we have shown, these types of models are very natural in many domains. We provided a semantics for recursive models that generalizes the semantics for the non-recursive case, and proved a strong existence theorem showing that all knowledge bases satisfying a natural and reasonable assumption have a model under this semantics.

We also developed approximate inference algorithms for reasoning about recursive probability models. As usual, we obtained major benefits from exploiting the object structure. In addition to the standard benefits of encapsulation and reuse of computation between different objects, we found dramatic benefits resulting from reuse of computation between different iterations of the approximation algorithm.

We believe that the added expressive power of a recursive language will be vital in many domains. On the other This extra power does not come without a price. We are no longer guaranteed that a KB will have a unique model. It is up to the knowledge engineer to design the KB in such a way that it has a unique model, or at least is sufficiently well-defined so that many queries have a unique answer. Designing recursive probability models is a skill much akin to programming. Very little is understood at this point about the behavior of these models.

Some important open questions are:

- Are there simple but non-trivial conditions that guarantee that a recursive probabilistic KB has a unique model?
- Are there simple but non-trivial conditions that guarantee that inference for a query on a KB with OOBN-style bindings reaches a recursive fringe?
- Is there a way to analyze the general form of a specification of a model so as to understand the complexity of inference for the model?

- Closely related to the last question: are there methodologies for designing recursive probabilistic representations that lead to efficient inference.

We hope that answers to some or all of these questions will be provided in the coming years. A good theoretical understanding of the properties of complex probabilistic models will go a long way towards making probabilistic knowledge representation a practical technology.

Chapter 8

Implementation and Applications

We have implemented a system called SPOOK (for “System for Probabilistic Object-Oriented Knowledge”) for building and reasoning with the richer probabilistic languages defined in this dissertation. In this chapter we describe the SPOOK system, along with three example applications: military situation awareness, computer system diagnosis, and modeling a university. In conjunction with the situation awareness application, we present experimental results that show how the advantages of using the **SVE** algorithm compared to the **KBMC** approach.

8.1 The SPOOK System

SPOOK is a system for representing and reasoning with object-based probabilistic models. A high-level diagram of the SPOOK architecture is shown in Figure 8.1. The SPOOK system itself consists of four main components: a core module, that contains all the data structures necessary for representing SPOOK models; a user interface, through which the user can create SPOOK models and ask queries about them; and two inference engines, one using the knowledge-based model construction approach and the other using the **Structured Variable Elimination** algorithm. Both SPOOK inference engines use a Bayesian network reasoning system to perform the underlying BN computations. This system, called Phrog, is a high-performance

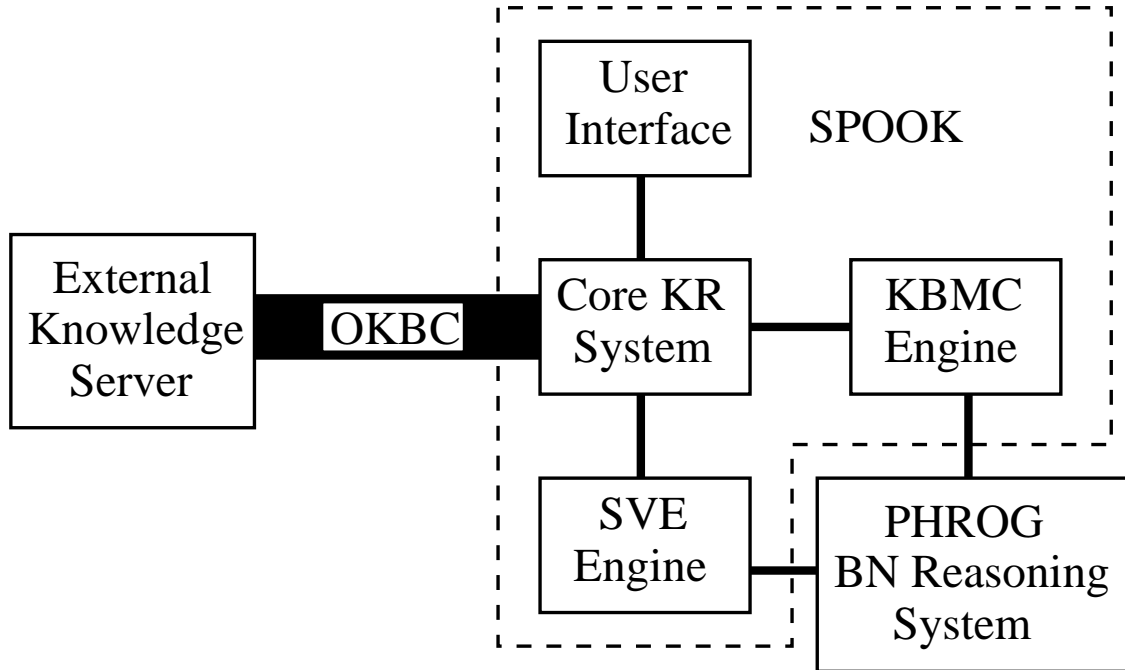


Figure 8.1: The SPOOK system architecture.

BN reasoning engine developed at Stanford.¹ SPOOK can also integrate with external knowledge servers using a protocol called OKBC [18], as will be described below.

The core modules of SPOOK were developed by Brian Milch, Ken T. Takusagawa, Ryan Shaw and myself. We have implemented much of the functionality described in this thesis. In particular, we have implemented relational probability models described in Chapter 5, including the version of the **SVE** algorithm described in Section 5.5. We have also implemented multi-valued attributes and number uncertainty, as described in Chapter 6, including the combinatoric algorithms for inference with number uncertainty described in Section 6.2.3. Things not yet implemented include reference uncertainty, the version of **SVE** that integrates RPMs with OOBN-style bindings, and the **Iterative SVE** algorithm for recursive probability models.

SPOOK was designed as a *frame representation system (FRS)*, or just *frame system* for short. Frame systems are a common kind of object-based knowledge

¹The Phrog system was developed by a number of people, in particular Uri Lerner and Lise Getoor.

representation system. The basic unit of discourse in a frame system is a *frame*. A frame has a set of *slots*, each of which may have *slot values* or *fillers*. Formally, a slot represents a binary relation on frames; if the filler of slot A in frame X is frame Y , then the relation $A(X, Y)$ holds. A slot in a frame may have associated *facets*. A facet is a ternary relation: if the facet value of facet F on slot A in frame X is Y , then the relation $F(X, A, Y)$ holds. A facet is often used to represent information about the values of slots, without specifying the slot values themselves. For example, a standard facet is `VALUE-TYPE`, which specifies a value restriction on the values of a slot.

One frame may inherit from another in a frame system. Slots, facets, slot values and facet values may all be inherited. The exact inheritance mechanism is system-dependent. Frame systems normally distinguish between instance-class relationships and subclass-class relationships, but there is no strong dichotomy between classes and instances. Any frame that has instances is considered a class, and a class may itself be an instance of another class (a *metaclass*). The slots of a class frame may be *own slots*, which describe a property of the class itself, and *template slots*, which are slots inherited by all instances and subclasses of the class. The facets associated with template slots are *template facets*, and are also inherited. An instance or subclass may override the values of inherited slots or facets.

The language of relational probability models maps very naturally into that of an FRS. Class and instance objects are of course frames, and attributes are slots. The probabilistic knowledge is represented using facets. We can view the probability model associated with a slot as a natural extension of the `VALUE-TYPE` facet. Whereas the `VALUE-TYPE` restricts the possible set of values of a slot, the probability model in addition defines a probability distribution over that set of values.

The specific protocol for defining probability models in facets is as follows. Probability models are associated with template slots of class frames. Each such slot will have a `VALUE-TYPE` facet. If the `VALUE-TYPE` is an explicitly enumerated set, the slot is simple; otherwise the `VALUE-TYPE` must be a class frame, and the slot is complex. In an ordinary FRS system, the `VALUE-TYPE` facet serves the purpose of making sure that no value of an inappropriate type is ever asserted for the slot. Here,

we take advantage of that to make sure that no value will ever be asserted that is not envisioned by the probability model.

A simple slot has two additional facets: `PARENTS` and `DISTRIBUTION`. The value of the `PARENTS` facet lists the parents of the slot, represented as a list of lists of slots, each list of slots denoting a slot-chain. The value of the `DISTRIBUTION` facet is an instance of the `conditional-distribution` class, and defines the conditional probability function of the slot.

The use of a frame to represent the CPFs makes it possible to develop an ontology of probability distributions and conditional distributions which can then be used in developing a particular probability model. We use a two-tiered system for representing CPFs, with the first tier consisting of instances of the `probability-distribution` class, and the second tier consisting of instances of the `conditional-distribution` class. There are many ways to represent both probability distributions and conditional probability distributions; different representations can be supported using subclasses of `probability-distribution` and `conditional-distribution`. In SPOOK, we have focused on discrete probability distributions, defined by the `discrete-distribution` subclass of `probability-distribution`, and on CPFs represented as tables, defined by the `conditional-probability-table` subclass of `conditional-distribution`. However, our use of an ontology allows our protocol to extend naturally to other representations of probability distributions and CPFs. For example, `probability-distribution` may have a subclass `continuous-distribution`, which may itself have a subclass `gaussian-distribution`.

The `discrete-distribution` class has two slots. The `RANGE` slot is an enumerated list of values. The value of the `PROBABILITIES` slot is a function.² The function maps values in the `RANGE` to real numbers in the interval $[0, 1]$. We require that the values of the probability function sum to 1, and ask that the FRS enforce this constraint if it is able to do so.

The `conditional-probability-table` class has three slots. The `RANGE` slot is an enumerated list of values. The `PARENT-VALUES` slot is a list each of whose elements

²We assume that the FRS being used as the external knowledge server supports reified functions, so that they can be the values of slots. Many but not all FRSs have this capability. For an FRS that does not support this feature, an explicit representation must be used, such as a list of $\langle \text{value}, \text{probability} \rangle$ pairs.

is an enumerated list of values, representing the range of values of each parent. The value of the `DISTRIBUTIONS` slot is a function that maps a tuple of values from the cross-product of the values of the `PARENT-VALUES` to an instance of the `discrete-distribution` class. As above, the function can be specified in the FRS as a set of ground facts, one for each combination of parent values. We require that every value of the `DISTRIBUTIONS` function has the same value of `RANGE` as does the conditional distribution, and ask that the FRS enforce this constraint if it is able to. We also require that for any simple slot, the value of its `DISTRIBUTION` has the same value for its `RANGE` as does the simple slot, and again ask the FRS to enforce this constraint.

Complex slots also have facets in addition to the `VALUE-TYPE` to describe their probability model. The `INVERSE` facet is fairly standard in FRSs; its value (if it exists) should be a slot of the `VALUE-TYPE` of the complex slot. A complex slot also has an `IMPORTS` facet, and may also have an `EXPORTS` facet. The values of each of these are lists of slots. These facets have the same meaning as in Definition 5.3.4.

Number uncertainty can be handled very simply in the protocol. FRSs have standard `MIN-CARDINALITY` and `MAX-CARDINALITY` facets, indicating the minimum and maximum number of allowed fillers for the slot. A slot for which the values of these facets differ has number uncertainty. A slot with number uncertainty has an additional `NUMBER-SLOT` facet, whose value is an associated number slot. The number slot is a simple slot of the same frame, with the appropriate numeric range. The number uncertainty is expressed in the probability model of the number slot.

SPOOK was developed as a basic FRS, with particular emphasis on the functionality needed to represent probability models. There are a number of highly developed FRSs in existence, including Ontolingua [25], LOOM [64] and CYC [61]. These systems provide many features not available in SPOOK, such as support for extensive libraries of shared knowledge, and various non-probabilistic reasoning mechanisms. In particular, CYC and Ontolingua have full first-order expressive power. In order to allow the user to take advantage of these capabilities, and to augment existing knowledge bases with probabilistic knowledge, SPOOK can use an external knowledge representation system as its knowledge server.

SPOOK communicates with its external knowledge server using the *Open Knowledge Base Connectivity (OKBC)* protocol developed by SRI and the Knowledge Systems Laboratory at Stanford [18]. OKBC is a protocol for communication between knowledge representation systems. The knowledge model of OKBC is frame-based, so it fits easily with the frame-based specification of SPOOK models. One of the features of OKBC is that it allows a system to specify its capabilities in the protocol, so SPOOK can inquire of a server whether or not a particular server has the capability to represent SPOOK models. In particular, SPOOK can inquire whether it allows all the facets required by SPOOK to be defined, and whether template facets are inherited. Both of these functionalities are required for SPOOK to work. We have successfully integrated SPOOK with the Ontolingua server [25] developed by Stanford's Knowledge Systems Laboratory.

We have recently learned that OOBNS have been implemented in the commercial HUGIN probabilistic reasoning system [46]

8.2 Example: Military Situation Awareness

Military situation awareness is the task of reasoning about the status and activities of units in a battlespace, based on intelligence reports about the units in the space. This task is naturally suited to probabilistic reasoning. There will typically be only very limited information about the different units, which must be used as evidence to infer the properties and configuration of the units. However, as shown by Mahoney and Laskey [66], the situation awareness domain is a very challenging one for traditional BN technology. The reason is that there are typically many units in a battlespace, with highly flexible configurations. The configurations of enemy units are in fact generally not known.

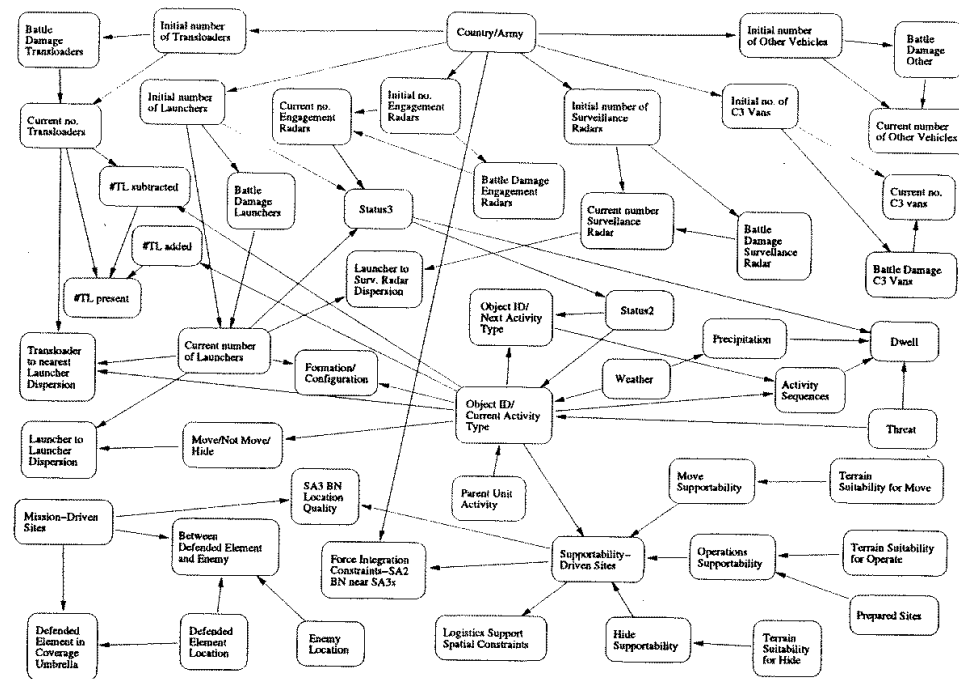
Our model deals specifically with missile battalions, the batteries within those battalions, and the individual basic units — vehicles, radar emplacements, missile launchers, etc. — within the batteries. A scenario consists of multiple battalions, some of which may be at the same location. A battalion typically has four batteries, each with about 50 basic units. Let us consider trying to model this domain with a

flat BN. With, say, four or five variables for each basic unit, a flat BN for a battalion model will typically contain over a thousand nodes. The sheer size of this network is a major obstacle to its construction. In addition, the resulting BN will be too rigid for practical purposes. The configuration of a battalion is highly flexible, with the exact number of units of each type varying considerably between different battalions.

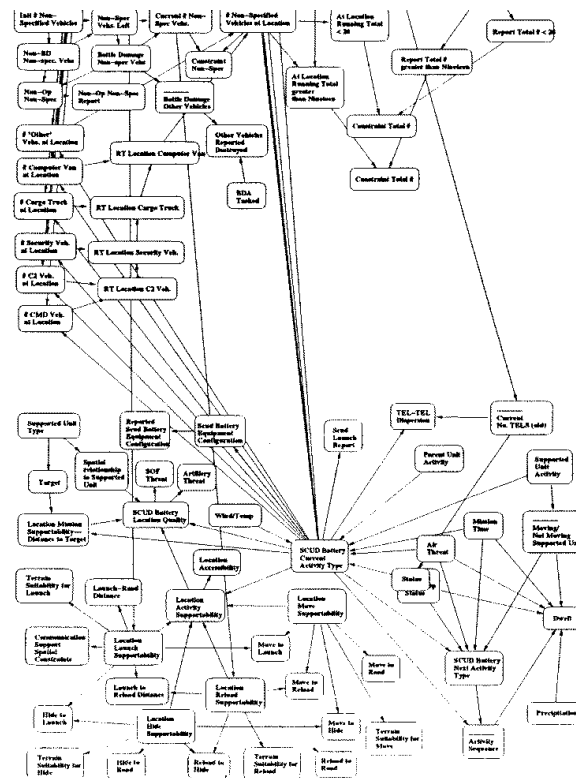
These difficulties have led to an alternative approach, in which several different BNs are used, one for each aspect of the model. Figure 8.2 (a) shows a Bayesian network for an SCUD battalion, while Figure 8.2 (b) shows a network for a SCUD battery.³ There are similar networks for other types of units, such as SA2 battalions and batteries. Although a SCUD battalion contains SCUD batteries, the battalion model does not replicate all the details of the battery model; rather, it summarizes the status of all the batteries with nodes, indicating the initial number of batteries, the number of damaged batteries, and the current number of batteries present. Similarly, the battery model does not contain detailed models of all its individual units, but rather summaries of the units of different kinds. These summaries serve two purposes: to keep the network reasonably simple; and to account for changing model configuration by making the initial number of subunits a variable.

A major disadvantage of this approach is that it is very difficult to reason between the different networks. The only way to reason from one network to another is to reach conclusions about the state of variables in one network and assert them as evidence in the other network. For example, the only way to transfer conclusions from a battery to a battalion is to condition one of the summary nodes in the battalion model; going from one battery to another requires conditioning the battalion model, reasoning about the battalion, and then conditioning the other battery model. This type of reasoning has no sound probabilistic semantics. In fact, it can yield incorrect results when the conclusions from reasoning in one network are not close to deterministic, and is particularly problematic when the conditioning is performed on multiple random variables. Furthermore, this type of reasoning between fragments must be

³Grateful thanks to Suzanne Mahoney, KC Ng, Geff Woodward and Tod Levitt of IET Inc. for these models.



(a)



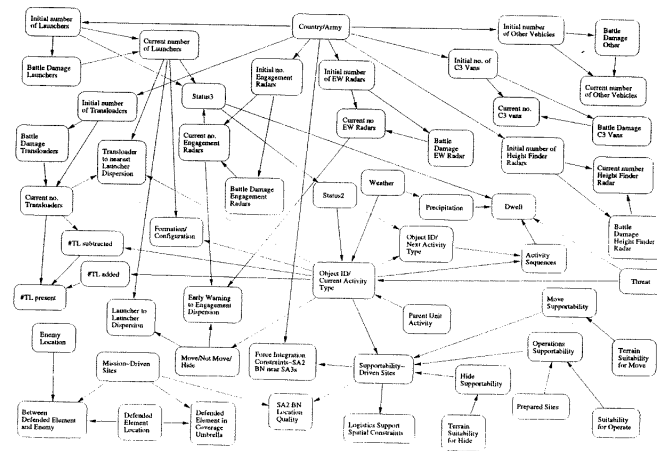
(b)

Figure 8.2: (a) SCUD Battalion Bayesian network. (b) SCUD Battery Bayesian Network.

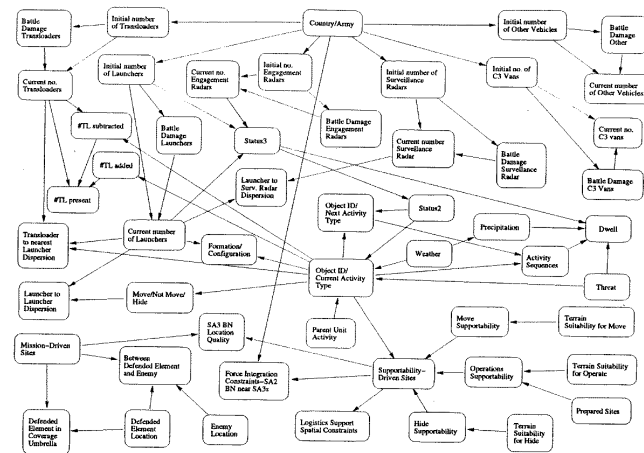
[illegible]

Another disadvantage is that multiple BNs do not allow us to take advantage of redundancy within a model and similarities between models. For example, the battalion model in Figure 8.2(a) contains many similar substructures, summarizing groups of basic units of different kinds, as illustrated by the circled regions in Figure 8.3. In addition, different battalions may all have substructures describing their locations, as shown in the bottom right corner of the figure. Furthermore, some battalions will have very similar structures, and their models will contain many of the same substructures. This is illustrated in Figure 8.4, showing the models for SA-2 and SA-3 battalions. In the multiple BNs approach, the only mechanism for exploiting these redundancies is cut-and-paste. This makes it very hard to maintain these models, because each time one of the reused components is changed, it must be updated in all the different networks that use it.

OOBNs solve the problems inherent in the multiple BN approach. In fact, the hierarchical nature of a military organization falls naturally into the OOBN framework. By allowing a battalion to contain a battery as a sub-object, we can easily



(a)



(b)

Figure 8.4: (a) SA2 Battalion Bayesian network, (b) SA3 Battalion Bayesian network.

have the battalion model encompass the complete models of the different batteries in it, which in turn contain complete models of their subunits, without making the battalion model impossibly complex. We can then reason between different objects in the part-of hierarchy in a probabilistically coherent manner. In addition, by allowing us to define a class hierarchy, OOBNs allow us to exploit the redundancy in the model. For example, we capture the commonalities shown in Figure 8.4 by creating an **SA-Battalion** superclass of **SA2-Battalion** and **SA3-Battalion**. In addition, we can exploit the redundancy between the different groups in a battery by representing them all using a **group** class.

However, the language of OOBNs is insufficient to model the situation awareness domain to our satisfaction, and we need the more expressive language of relational probability models. If we want to model the effect of a unit's location on the unit, we need to represent the relationship between the unit and its location. In our model, this was the only relationship that did not fall into the part-of hierarchy, but richer models of the battlespace domain require more sophisticated relationships, such as that between a unit supporting another unit. In addition, our domain requires multi-valued attributes and quantifiers. A battalion contains several batteries, and each battery contains several units of different types. The higher level objects do not depend directly on the individual lower level objects, but only on aggregate properties of the set of objects, expressed through quantifiers. The ability to create named instances and hook them together via relations is also important in our domain. For example, we want to be able to describe situations in which two battalions share the same location, so as to reason from one battalion to another via their common location. Finally, the battlespace domain contains a great deal of structural uncertainty, in particular uncertainty over the number of subunits of a unit.

Our SPOOK model of the battlespace domain includes a natural class hierarchy, with **Military-Unit**, **Environment**, **Location** and **Weather** as root classes. Our model also has a natural part-of hierarchy, with **Battalion** classes containing **Battery** classes, which contain **Group** classes, which in turn contain **Basic-Unit** classes. While we only modeled the domain up to the battalion level, we could easily extend our model to higher-level groups in the military hierarchy. The **Battalion**, **Battery**, **Group**, and

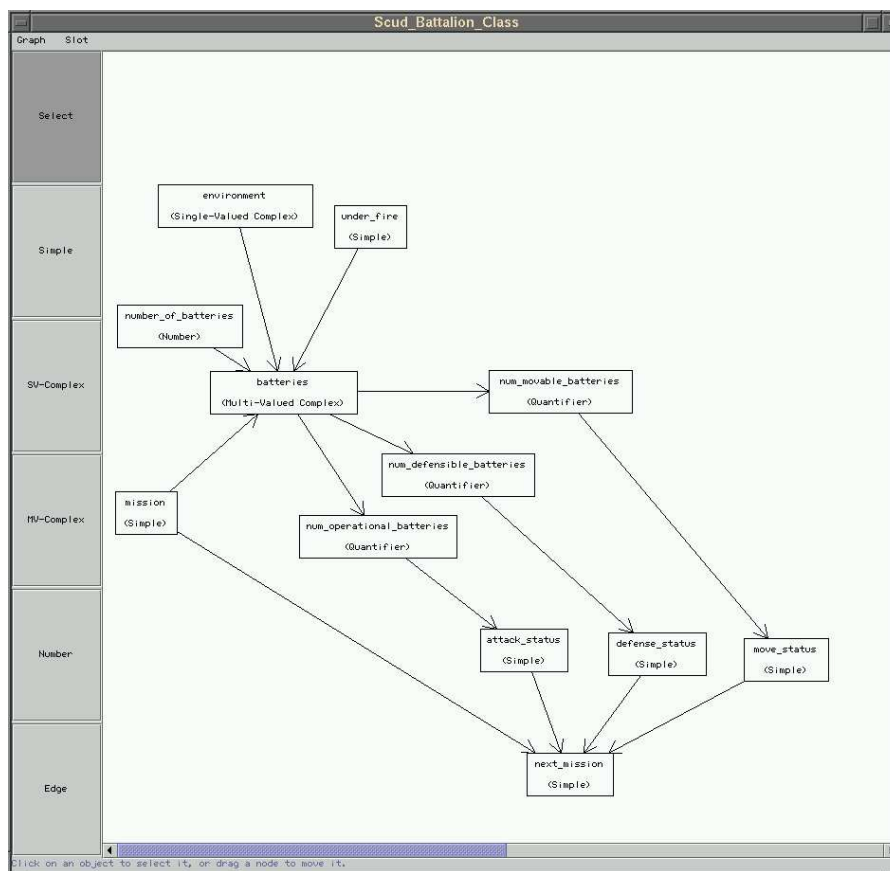


Figure 8.5: SCUD-Battalion class model.

Basic-Unit families are all part of the **Military-Unit** hierarchy.

Figure 8.5 is a screenshot from the SPOOK system, showing the model for a SCUD battalion. The model contains simple attribute for the current and next mission of the battalion, and whether or not it is under fire. The next mission depends probabilistically on the current mission, and on its current ability to operate, defend itself and move. The model contains a single-valued complex attribute for the environment, encapsulating the location, weather and terrain. (The location describes general, fixed properties of the location, such as whether or not it is hilly, whereas the terrain describes temporary aspects that can also be affected by the weather, such as whether or not it is muddy.) It also contains a multi-valued complex attribute representing the batteries in the battalion, with an associated number attribute. The battalion

influences the batteries via the environment and whether or not it is under fire, and is influenced by them through the quantifiers indicating the number of batteries that are capable of performing operations, defending themselves, and moving.

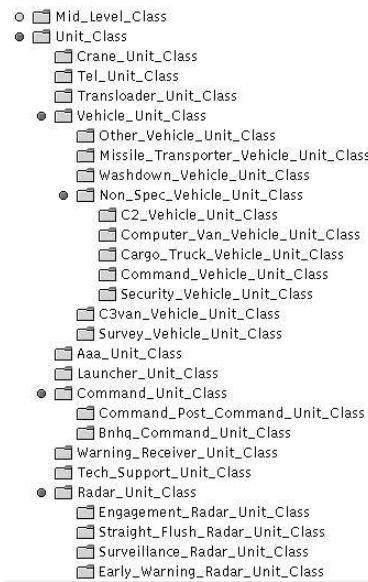


Figure 8.6: Class hierarchy of basic units in the battlespace model.

Batteries do not contain basic units directly, but instead contain a **Group** object for each type of basic unit. For instance, a battery has (among others) groups of missile launchers, command vehicles, and anti-aircraft artillery units. Each **Group** has a multi-valued attribute relating it to the individual units, as well as a number attribute and a set of quantifier attributes that summarize the status of the units. The purpose of adding the extra **Group** layer to the hierarchy is to allow the same quantifiers to be reused across many different groups of individual units. The **Basic-Unit** class represents the basic military units, and has a rich family of subclasses, illustrated in Figure 8.6 (another SPOOK screen-shot).

An object of class **Basic-Unit** has simple attributes **reported**, **operational**, **damaged** and **reported-damaged**. These attributes are influenced by the location of the battalion — specifically, the location’s support for concealment and defense — and by the battalion being under fire. We represent these influences in SPOOK by specifying, for example, `in-battery.in-battalion.at-location.defense-support` as a parent of **damaged**.

The number of damaged units in turn influences the battery's **operational** attribute, and a quantifier slot that counts the number of operational batteries in a battalion influences the battalion's **current-activity**. Subclassing gives us the ability to provide models for certain types of units that are similar to the general unit model but not exactly the same. For instance, **Missile-Launcher** has an additional **activity** attribute that indicates whether it is launching, reloading, or idle.

In our current model, all units in a battalion share a common environment, which is referred to by the **in-environment** attribute of the battalion. The environment is composed of **Location** and **Weather** objects, which between them determine the current support of the environment for various activities such as moving, hiding and launching missiles.

One might want to associate a different environment with each battery or unit, making locations of lower-level objects related probabilistically to higher level objects. Adding a hierarchy of environments, in parallel with the hierarchy of units, would not have been difficult if the environments were generic. That is, each individual unit could be associated with an environment with certain properties, depending probabilistically on the properties of the environment of the containing object. On the other hand, without a hierarchy of environments, one can easily associate the environments with specific map locations, so that every battalion is at a particular location on the map. In addition, one can model structural uncertainty over the locations of the particular battalions without difficulty.

However, combining a hierarchy of environments with named map locations is very challenging, if the lower level units are also associated with map locations. The difficulties are two-fold. First of all, creating a correct model for the spatial distribution of the units in the hierarchy is difficult. One needs to model the different possible configurations of the batteries in a battalion, and of the subunits in a battery, and integrate that model with the terrain models. Secondly, one would like to make independence assumptions, stating that the configurations of the subunits in the different batteries are independent of each other, but it is far from clear that these assumptions are correct.

We see therefore that although our framework goes a long way towards representing the situation awareness domain, and certainly makes it easier to represent than existing frameworks, it is not up to the task of representing the domain in all of its detail. Nevertheless, even without creating an extremely detailed model, one can still obtain a rich and useful representation of the domain.

To give an example of the power of reasoning at multiple levels of the hierarchy and between different objects, we present a series of queries we asked the model. First we queried the prior probability that a particular SCUD battery was hit, and found it to be 0.06. We then observed that the containing battalion was under heavy fire, and the probability that the battery was hit went up to 0.44. We then observed, however, that none of the launchers in the battery had been reported to be damaged, and the probability that the battery was hit went down to 0.28. We then explained away this last observation, by observing that the environment has good support for hiding; the probability that the battery was hit went back up to 0.33.⁴ This example combines causal, evidential and intercausal reasoning, and involves battery and battalion objects, individual launcher objects, the launcher group, and the environment object.

8.3 Experimental Results

In our experimental setup, we constructed SCUD battalion models of different sizes. We varied the size of the model by varying the number of basic units of each kind within a battery. Each model consists of a single battalion with four batteries, each containing 11 groups of different kinds of units. The number of units in each group varied from 1 to 10. The model also contains objects for the environment, location and weather. The size of the constructed BN grows linearly in the number of units per group, and varies from 750 to 5500 nodes.

⁴The model encoded the assumption that good hide support helps units to avoid detection, but does not make it less likely for them to be hit, which is why observing good hide support made the probability of hit go up. If the model had also encoded a dependence of a unit being hit on the existence of good hide support, observing good hide support would have provided direct causal evidence that a unit had not been hit, counteracting the explaining away effect.

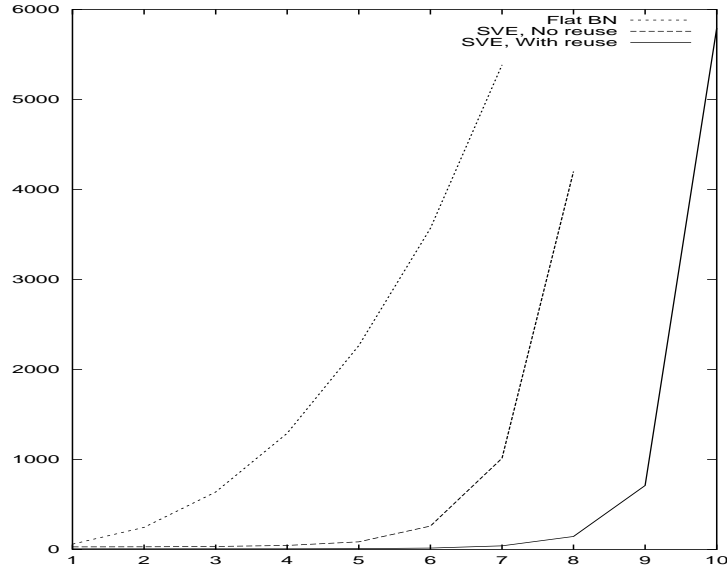


Figure 8.7: Comparison of unstructured and structured inference algorithms.

In our experiments, which were performed on a Sun Ultra-2 machine, we compared the performance of the object-based **SVE** algorithm with that of the KBMC algorithm, which constructs a flat BN and then performs inference in that BN. In order to measure separately the benefits from exploiting interfaces and from reusing computation, we tried two different versions of the object-based algorithm, with and without reuse. We also compared the naive and combinatoric approaches to dealing with multivalued attributes described in Section 6.2.3. We compared the different algorithms on a query on **Battalion.Next-Mission**, which depends (indirectly) on the status of most of the individual units in the model. Figure 8.7 shows the running time of KBMC, **SVE** with no caching, and **SVE** with caching, on models of different sizes. The x -axis shows the maximum number of units in each group within a battery, while the y -axis shows running time in seconds.

From the graph, we see that all versions of the object-based algorithm outperform the KBMC algorithm by a large margin, and that the algorithm with reuse outperforms the algorithm without reuse. For example, with four units per group,

the object-based algorithm with reuse takes 9 seconds, without reuse takes 46 seconds, while the KBMC algorithm takes 1292 seconds.⁵ In this particular experiment, we did not have evidence about many different individual units in the battlespace, and the scope for reuse was great. If we had evidence about the individual units, we would not be able to reuse inference between them, and we would only obtain the gains from exploiting the interfaces between the objects. Nevertheless, the graph shows that those gains alone are quite significant. In addition, the interconnectivity structure of the objects in the model was very simple — nearly a tree structure, as in OOBNs. It is likely that with a more complex structure the results would not have been quite as impressive.

The reason for the great disparity between the inference times for the flat BN and for the object-based algorithm without reuse, is that the BN reasoning algorithm is failing to find optimal junction trees in the flat BN. The largest clique constructed for the flat BN contains 18 nodes, whereas the largest clique over all of the local BN computations for the structured algorithm contains only 8 nodes. The BN inference engine uses the standard minimum discrepancy triangulation heuristic to construct the junction tree. We see that at least for a standard BN implementation, exploiting object structure and the small interfaces between objects is vital to scaling up inference. While algorithms do exist for computing optimal triangulations [90], these tend to be quite expensive; and most implementations of Bayes nets do not use them; furthermore, these algorithms do not address the issue of reuse.

The curves in Figure 8.7 for structured inference, both with and without reuse of computation, show the results for the algorithm using the naive approach to reasoning with multi-valued attributes, in which a separate single-valued attribute is created for each of the fillers. In Figure 8.8 we show the improvement obtained from exploiting symmetry and using combinatorics. The model used for the experiments of Figure 8.8 also used number uncertainty. The curves are exactly as we would expect. Without combinatorics, the curves exhibit an exponential blowup as the number of possible units in each group increases. With the combinatorics, the growth is linear in the

⁵In our original experiments, the junction tree construction code was not fully optimized. We have run some more experiments on an optimized version of the code, and found that the KBMC algorithm runs approximately 30% faster. This does not change the qualitative nature of the results.

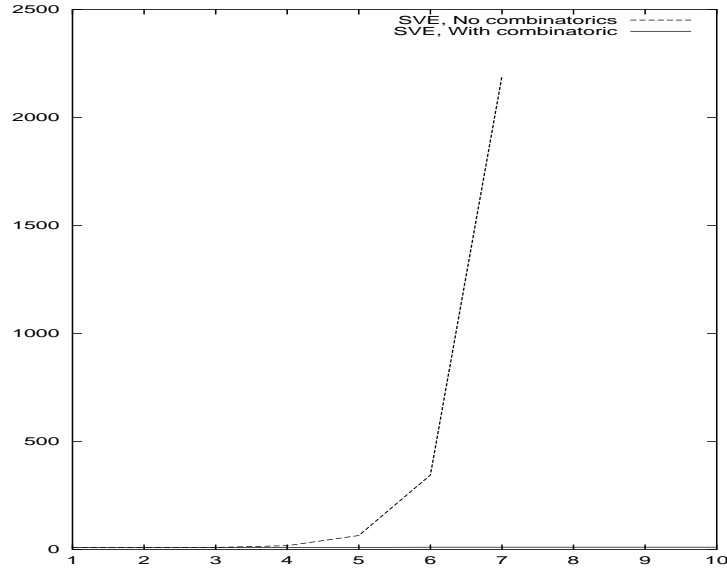


Figure 8.8: Comparison of naive and combinatoric approaches to inference with quantifiers.

number of possible units.⁶ The running time topped out at 11 seconds for a maximum of 10 units per group.

8.4 Example: Computer System Diagnosis

For our next example, we describe a model for diagnosis of a computer system. This domain falls very naturally into the OOBN framework, as does diagnosis of complex manufactured systems in general, such as locomotives or satellites. Manufactured systems are constructed out of components, which map naturally onto the OOBN part-of hierarchy. Computer systems have both “hard” components such as hard drives, and “soft” components like the operating system, but the principle is the same. We developed this example in a fair amount of detail in Chapter 4. The model describes a basic standalone PC system. The knowledge engineering was based on [68].⁷ Here we consider various design issues encountered while designing the

⁶The curve is hard to see since it is almost parallel to the x -axis and close to it.

⁷We do not claim to be knowledgeable about PC maintenance and repair. Any egregious errors in the model should be attributed to our lack of expertise.

model.

One design issue involves the organization of the part-of hierarchy: should it be organized along physical or functional lines? For example, should the video controller be encapsulated inside a functional **Display** subsystem, that also includes the monitor, even though physically the video controller is found on a circuit board inside the computer? From the point of view of localization of probabilistic influence, both design decisions make sense, since both functional and physical influences flow around the system. For example, the quality of the observed display depends on the state of the monitor and the video controller, suggesting that the controller should be encapsulated within the **Display** subsystem. On the other hand, overheating inside the computer can affect the video controller, suggesting that the controller should be encapsulated according to its physical location inside the computer. Choosing to organize the hierarchy along functional lines would require us to pass physical information through the interfaces, and vice versa. We chose to use a functional hierarchy, since the number of functional variables that would have to be passed around is greater than the number of physical variables. As can be seen from Example 4.2.5, the physical variables **Age** and **Temperature** get passed from the **Computer** class to its contained **Hard-Drive**.

Another, related, design issue is to determine the degree of encapsulation that is beneficial. More encapsulation can lead to more locality of inference, but too much encapsulation can require passing a lot of information through interfaces. An example of over-encapsulation would be to encapsulate the power supply unit inside the motherboard, where it is physically located. The reason is that the entire system depends on the stability of the power supply, not just components on the motherboard. On the other hand, we found it useful to encapsulate cables within other components. Most cables are more readily associated with the component at one end than the one at the other. For example, the cable connecting the mouse to the computer is associated with the mouse, while the cable connecting the printer to the computer is associated with the printer.

Once we have determined the basic design of the computer system model, we can create families of subclasses for the different types of components. An example of a

family of mouse classes was presented in Example 4.6.2.

After creating the computer model, we need to determine how to use it in a diagnosis situation. As described in Section 4.6, a way to do this in the OOBN framework is to add various types of events to the computer class model, such as file read and write events, and print events. Each event depends on the status of some of the components of the system. For example, a print event may depend on the status of the printer and the operating system. This approach is not very flexible, since it does not allow us to consider arbitrary collections of events and use them to diagnose a situation. It is here that we run into the limitations of OOBNs, and must use the more general relational framework. We create a high-level abstract **Event** class, with a single complex **Of-Computer** attribute, of type **Computer**. The event class has concrete subclasses for the different types of events. For example, the **Print-Event** subclass of **Event** has an **Outcome** attribute that may depend on **Of-Computer.Has-Printer.Connected** and **Of-Computer.Has-OS.Printer-Driver-Installed** among other things. Using this approach, a scenario can be created involving any number of events.

The model can be further refined by introducing **Application** objects. An **Event** object can now be associated with a particular application, rather than with the computer as a whole. Adding this extra level to the model can help determine whether a fault is with the way the computer system is set up or with a particular application. For example, if a **Print-Event** associated with a word-processing application works successfully, while one associated with a music printing application fails, then with high probability there is a problem in the way printing has been set up in the music application.

Using a relational model also allows us to diagnose multiple computers connected over a network. In particular, **Event** objects may refer to more than one system. For example, a **Networked-Print-Event** depends on both the computer issuing the print command and the printer on the network, while a **Client-Server-Event** requires that both the client and server objects be functioning correctly. Similarly, a **File-Read-Event** on a networked file system depends both on the computer with which the data is stored and on the computer into which the data is read.

Adequately modeling an event involving multiple systems requires modeling whether

or not the network connection between them is alive. A simple way to model this is to introduce a `Connection` object for each pair of systems on the network. This scheme is wasteful, since the number of `Connection` objects required is quadratic in the number of computers in the network. Furthermore, correlations between different `Connection` objects are lost. If the connection between machines A and C passes through machine B , then whether or not the connection between A and C is alive depends on the connections between A and B and between B and C .

A much better approach would be to follow the actual physical structure of the network, and create a `Connection` object for each physical connection. For any event involving two computers A and B , the outcome of the event will depend on whether there exists a path of live connections between A and B . Formally, if we define the relation *Live* to consist of those pairs of machines that have a live physical connection between them, we want to know if the pair (A, B) lies in the transitive closure of the *Live* relation. Unfortunately, transitive closure cannot be expressed in our representation language. Even if we were to introduce a transitive closure primitive, we would be faced with an extremely difficult inference problem. Suppose we were to observe that A and B are not in the transitive closure. This provides us with evidence that one or more links that could have connected them is down. But how do we determine the posterior probability over the liveness of each of the physical links in an efficient manner? There is no problem in principle in calculating this probability, but the inference is very expensive, because the probability of failure of all the different physical links becomes correlated.

8.5 Example: Modeling a University

For our final example, we present an application of a very different flavor: modeling students and courses in a university. This example illustrates the use of a combined logical and probabilistic representation language for reasoning about ordinary, everyday situations. We start with a BN describing a single student taking a single course, shown in Figure 8.9. This BN is very similar to that of Example 3.3.2, with extra nodes representing the quality of the teacher and the difficulty of the final.

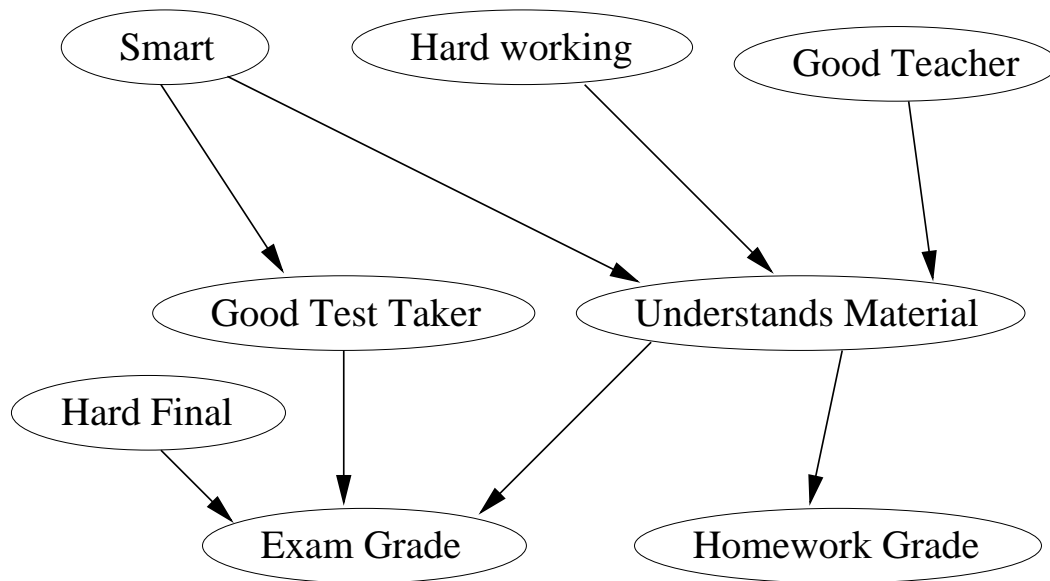


Figure 8.9: A Bayesian network describing a single student taking a single course.

Next, we extend the network to model the same student taking two different courses. A BN for this situation is shown in Figure 8.10 (a). The nodes **Good-Teacher**, **Hard-Final**, **Homework-Grade**, **Final-Grade**, and **Understands-Material** have been replicated for the two different courses. (Node names are abbreviated for convenience.) The **Smart**, **Hard-Working** and **Good-Test-Taker** attributes pertain to the student, and are shared between the two different courses.

Figure 8.10 (b) shows a BN for the contrasting situation of two students taking the same course. In this network, the **Smart**, **Hard-Working**, **Good-Test-Taker**, **Understands-Material**, **Homework-Grade** and **Exam-Grade** nodes are replicated, while the **Good-Teacher** and **Hard-Final** are shared. Observe that the **Understands-Material**, **Homework-Grade** and **Exam-Grade** nodes are replicated in both networks. These attributes are associated with student/course pairs, whereas the other attributes are associated either with a student or with a course.

It is clear that in order to generalize to multiple students taking multiple courses, a relational model is appropriate. We need four classes: a **Student** class with simple attributes **Smart**, **Hard-Working** and **Good-Test-Taker**, a **Course** class with simple attribute **Hard-Final**, a **Registration** class with simple attributes **Understands-Material**,

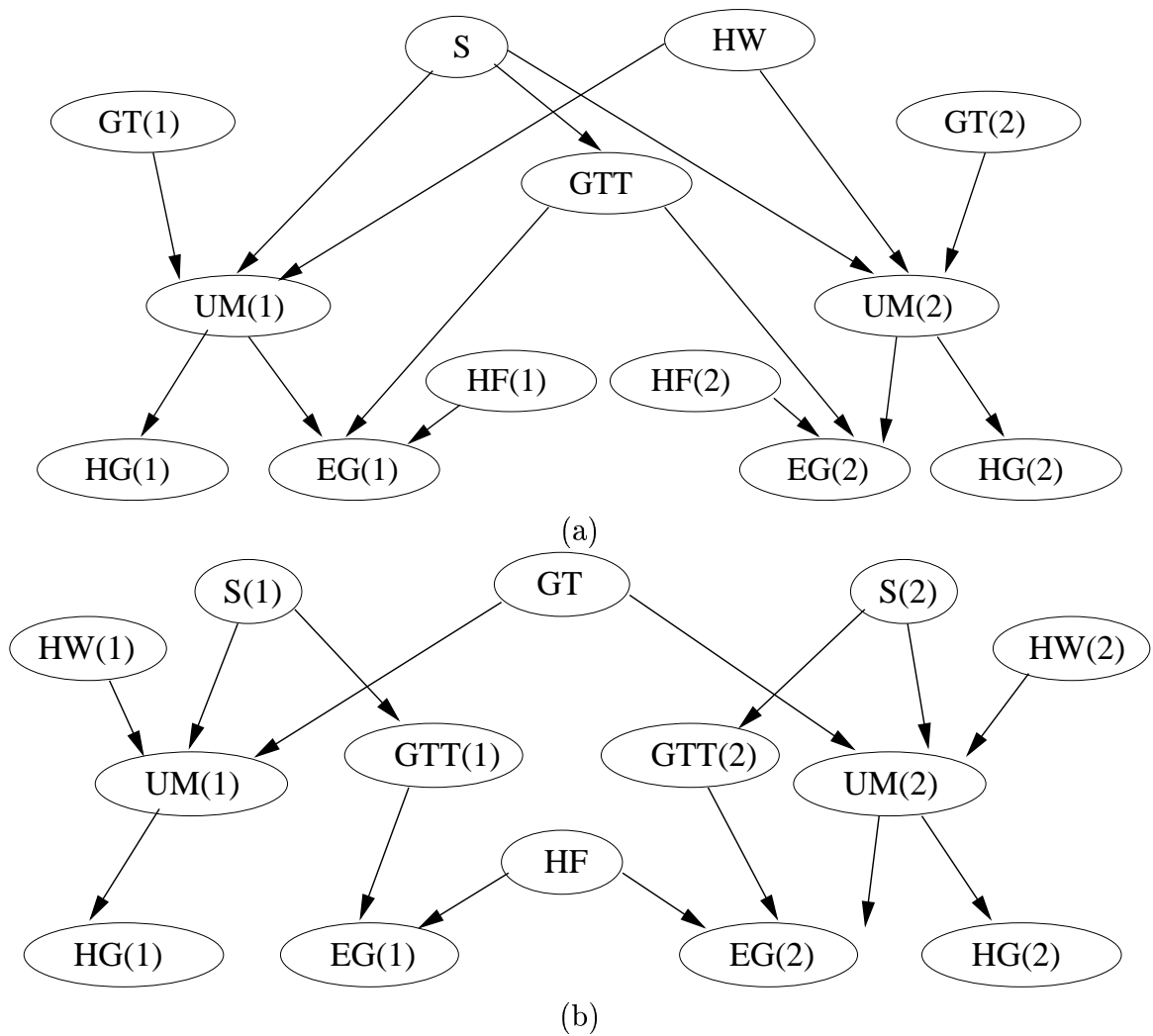


Figure 8.10: (a) BN for single student in two courses. (b) BN for two students in single course.

Homework-Grade, and Exam-Grade, and a Professor class, with simple attribute Good-Teacher. Modeling the teacher in a separate class from the course allows us to model multiple courses taught by the same person. An instance of the Registration class corresponds to a particular student taking a particular class. The Registration class therefore has the complex attributes Of-Student and In-Course, both of which are single-valued. The Student class has the multi-valued complex attribute Taking, which is an inverse of Of-Student, while the Course class has the multi-valued Registered attribute, which is an inverse of In-Course. The Course class also has the single-valued Teacher attribute, while the Professor class has the multi-valued Teaching attribute, which is an inverse of Teacher.

The local probability models for each of the simple attributes can be derived directly from the network of Figure 8.9. For example, the Understands-Material attribute of the Registration class depends probabilistically on Of-Student.Hard-Worker, Of-Student.Smart, and Of-Course.Teacher.Good-Teacher. Its CPF will be exactly the same as in the network of Figure 8.9. This example shows how a standard BN can be converted easily into a relational probability by associating each of the attributes with particular classes. Once this has been done, one can take a relational database — in this case, a database of students registered in courses — and construct a probability model for the entire domain, using the closed world semantics of Section 6.3.

Alternatively, once we have converted the model into the relational language, we can use other features of RPMs such as structural uncertainty. For example, one could associate number attributes with the Taking, Registered and Teaching attributes. Defining subclasses is natural in this domain — for example, the Seminar subclass of Course would have a different distribution over $\#[\text{Registered}]$ from that in its base class. In addition, the model can easily be integrated into models describing other aspects of university life. For example, the Professor class will have other attributes describing the research performed by the professor and the papers written. This part of the model will be nearly independent from the teaching model, but perhaps not completely, because Good-Teacher and $\#[\text{Papers}]$ may both depend probabilistically on Time-Spent-On-Research.

Knowledge representation in the combined logical-probabilistic language is in general quite natural and easy. However, there are some things that are challenging to represent. For example, a student's grade point average depends on her performance in all classes. It cannot be defined using the simple quantifiers we presented in Section 6.2. Rather, it is the average of the grades achieved in the different classes. Representationally it is not difficult to include to introduce average as a quantifier in the language, but inference with it becomes very hard. If one observes that a student's GPA is 3.5, and wants to know the probability that the student received an *A* in a course, one has to consider all the possible ways the student's grades in the different courses could combine to produce a GPA of 3.5.⁸ A more difficult challenge is to model the grade of a student taking a class graded on the curve. Here, the numeric grade of the student (contained in the `Registration` object) must be compared with the median grade of all students taking the course to produce a letter grade. The median can be an attribute of the `Course` object, computed as an aggregate from the numeric grades of all the students. It is again challenging to perform inference with this aggregate operator. It is possible that variational methods [51], which are also known as "mean-field methods" because they exploit the law of large numbers to reason about the average effect of a large number of influences, will work well for the types of aggregates discussed here.

8.6 Discussion

We have presented examples of relational probability models from several different domains. While we feel that the language does allow us to describe many interesting aspects of the domains, there are some things that are challenging, as seen in the examples. In this section we try to analyze what aspects of a system make it difficult to deal with in our language. Some of the challenging types of situations are as follows:

⁸Ironically, the problem is easier if the student takes many courses, because of the Gaussian approximation to the sum of a large number of i.i.d. random variables. (The grades are i.i.d. given the student's smartness, work ethic and test-taking skills.)

Many inter-connected instances We have already discussed this situation in Section 6.3, in the context of the sports teams example. The same type of problem shows up in the students/courses example. If there are many named students and many named courses, all the courses taken by a student will be correlated. As a result, the cost of exact inference for this model will be huge.

Uncertainty over complex combinatoric structures This is the situation encountered in Section 8.2, in which we saw that associating each unit in the military hierarchy with a specific map location is very difficult. Our approach to reasoning with structural uncertainty relies on the fact that the structural variables are independent of each other, or at least conditionally independent given a small number of other variables. If this assumption is violated, our approach is not feasible.

Mutual exclusion relationships These are notoriously difficult to deal with in standard BNs, and just as troublesome in our language, in particular in conjunction with reference uncertainty. In fact, if we have the constraint that no two battalions may be at the same location, then no independencies between the structural variables modeling the locations of the different battalions hold. Inference in this model quickly becomes infeasible even without a hierarchy of environments.

Numeric relationships We have seen this problem in the context of the students/courses example. Dealing with numeric relationships is difficult, because there are so many combinations of values for the arguments of a numeric formula that produce the same result. Observing a value for the result requires computing a joint probability distribution over the values of the arguments. The mean and median aggregates encountered in Section 8.5 are difficult for this reason.

Logical concepts Computing the probability of a logically defined concept such as transitive closure is very difficult, as we discussed in Section 8.4. The reason is similar to the previous case — there are often many different ways in which to satisfy a logical formula. For example, asserting that there is no path on

the network between machines A and B requires us to adjust the probability that each link between them is down. In order to do that properly, we have to compare the probabilities that there would be a path if the link was up and if it was not.

All of the problems discussed here are problems of inference, not of representation. Once we have the relational framework, it is generally not difficult to represent these more complex types of situations. On the contrary, the relational framework makes it very easy to envision and express very rich probability models for which inference is very difficult. The issue of performing probabilistic inference in very rich models may become increasingly important, as our ability to represent these models increases. Since exact inference will generally be impossible, the challenge will be to find a way to do inference in a reasonable amount of time, with a reasonable amount of accuracy. A judicious choice of domain-specific independence assumptions, together with the right approximation algorithms, could lead to the right answer in many cases.

For example, Pasula et al. [78] have addressed the problem of matching the observations of different sensors in a freeway traffic surveillance domain. In this domain, sensors are positioned at multiple positions in a freeway system. Each sensor reports the appearances of vehicles at its location. The challenge is to match up the sensor reports from different locations, and determine which ones correspond to the same vehicle. Pasula et al. have found that, with appropriate independence assumptions such as trajectory independence of the different vehicles, a Markov Chain Monte Carlo algorithm [72] works well in this domain. It remains to be seen whether any general answers can be found to the problem of probabilistic inference in very rich models, or whether domain-specific approaches will always be required.

Chapter 9

Related Work

9.1 Axiomatic Approaches

There are a number of directions from which researchers have approached the topic of integrating logical and probabilistic representations. Coming from the viewpoint of classical symbolic AI, probabilistic approaches are seen as a way of addressing many of the problems with logical representations, particularly the problem that real world knowledge is rarely the hard and fast kind that can be modeled using purely logical languages. If probability is to be used to address fundamental KR issues, it must be integrated with the classical logical languages which are seen as necessary components of KR.

Much of the early work in this direction focused on understanding the semantics of a combined logical and probabilistic representation. Bacchus [4] developed a framework in which probabilistic statements are understood as statistical statements about entities in the world. In this framework, there is a single relational possible world, which comes equipped with a probability distribution over the domain elements. Halpern [35] offered an alternative semantics, in which a probability distribution is defined over a set of possible worlds, each of which is an interpretation of the relational language. Probabilistic statements are understood as expressing degrees of belief, in terms of the probability distribution over possible worlds. We have adopted this approach in this thesis. Halpern also showed that the two different types

of semantics can be combined.

A basic impasse between traditional logical approaches and probabilistic approaches as embodied in Bayesian networks is that logical approaches tend to be *axiom-based*, in that only knowledge that is known for certain is expressed, while probabilistic approaches tend to be *model-based*, in that all possible states of the world are considered and a single probability distribution is specified over all the possible states. A logical KB typically has many possible models, while a BN has a unique model, which is itself a probability distribution over many possible models. Logical inference attempts to draw conclusions that hold in all possible models of a KB. In contrast, probabilistic inference attempts to determine the unique answer to a query, in terms of the unique probability distribution over possible worlds. Early attempts to integrate probability and logic generally took the minimalist approach. For example, Nilsson's probabilistic logic [76], which integrated probability with propositional logic, allowed probabilistic statements with inequalities. The semantics of Nilsson's language allowed a space of models, each of which is a probability distribution over possible worlds. The first-order probabilistic languages of Bacchus and Halpern are similar in this regard.

Much of the recent focus in KR has been on restricted subsets of first-order logic, such as description logics [11, 64, 81]. Accordingly, researchers attempted to create probabilistic variants of these logics. For example, Shastri [89] integrated probabilities into semantic networks, while Heinsohn [41] presented a probabilistic description logic. Both Shastri's and Heinsohn's logics use an axiomatic approach, augmenting an existing logical language with probabilistic knowledge, and drawing whatever conclusions could be drawn from that knowledge.

It was only with the success of Bayesian networks that the power of the model-based approach to probabilistic reasoning was appreciated. Experience also showed that the axiomatic approach was quite limited, in that without specifying a unique probability distribution over the domain it is hard to draw any interesting and meaningful conclusions from a knowledge base. One way to circumvent the limitations of the minimalist approach, while still allowing a logical language to be augmented with a small amount of probabilistic knowledge, is to allow many different models of

a KB, and then to choose one according to some criterion. A number of researchers have followed this path, often using the principle of maximum entropy [91] to select a particular probability distribution from the space of distributions consistent with a KB. Paris and Vencovska [77] implemented this approach for a highly restricted logic, while Bacchus, Grove, Halpern and Koller [6, 33] developed it for a much more expressive language. Jaeger used this approach to develop a probabilistic description logic [48].

Although the maximum entropy principle has some theoretical justifications, it also has some severe limitations, and using it to choose a probability distribution can have unexpected and unfortunate results [37]. Experience with Bayesian networks has shown that it is better to specify a complete distribution directly, using independence statements to make the representation feasible, rather than to rely on a principle such as maximum entropy to choose a distribution.

9.2 Model-Based Approaches

9.2.1 Structure Bayesian Networks

Another direction from which to approach the topic of expressive probabilistic representations is research in probabilistic reasoning methods and Bayesian networks. Researchers in this area have investigated various ways in which to extend the power and applicability of BNs. One has been to introduce hierarchy into the representation, as we did in OOBNs. The *multiply sectioned Bayesian networks (MSBNs)* of Xiang, Poole and Beddoes [100] are an inference framework based on the hierarchical clustering of network nodes. The basic idea of MSBNs is to create a “hypertree” of junction trees, with each node in the hypertree itself being a junction tree. Each pair of adjacent nodes has a separator between them, which is also a junction tree. The inference framework of MSBNs is quite similar to that of **SVE** for OOBNs. In particular, we exploit the same idea that they do of organizing the inference in a tree structure, and using the interfaces between a node and its parent in the tree to communicate between them during inference. In addition, their idea of representing

the interfaces in a decomposed form as a junction tree is a very good one; we show how to exploit this idea in our framework in the version of **SVE** in Section 4.5.5.

Srinivas [94] presented an approach to model-based diagnosis based on a hierarchy of BNs. His ideas contain the germ of some of our work on OOBNs. In his framework, each component in a component hierarchy has an associated probabilistic model, just as our OOBN objects do. However, his language is quite limited in the ways the objects in the hierarchy can communicate with each other. A component does not share any variables with its containing object. Rather, each component is represented in its containing object as an abstract node, that mimics several possible modes of behavior of the component. As a result, his approach is suitable for fault models, but not for general models of hierarchical systems.

9.2.2 Knowledge-Based Model Construction

A major thrust of research in the probabilistic reasoning community has been to use *knowledge-based model construction* to construct BNs tailored to specific situations from a general knowledge base. The basic approach of KBMC, pioneered by Breese [16], is to create a knowledge base containing the rules needed to build BNs for a particular domain. A particular situation is described by ground facts in the knowledge base. The ground facts and BN construction rules work together to produce a particular BN for a specific scenario, which can then be used to answer queries about that scenario. A review of early work using KBMC can be found in [98].

There have been a number of variants on the KBMC approach, differing mainly in the language used to express the BN construction rules, the types of rules allowed, and the method used to implement the knowledge base. For example, Goldman and Charniak [32] present a network construction language that uses forward-chaining rules to specify how to construct the network, and special combination rules to specify how the CPFs of nodes in the network are computed. Their framework, based on a truth maintenance system, was applied to natural language tasks such as word-sense disambiguation and pronoun resolution. Bacchus [5] allows a much more general first-order probabilistic logic. Poole [83], meanwhile, uses a more restricted language,

based on probabilistic Horn rules, in which probabilities are associated with different possible explanations of a fact.

Poole’s framework is closely related to that of *probabilistic logic programming (PLP)*. In this approach, developed by Ngo, Haddawy and others [74, 75], probabilities are associated with the rules in a logic programming language such as Prolog. Each rule is interpreted as a causal relationship, with an associated “noise” factor indicating the probability that the cause has the normal effect. The different causal rules that can have the same effect are combined using a combination rule, with noisy-or being a typical choice. Glesner and Koller [30] have applied PLP to dynamic systems.

At the core of the PLP framework is a backward-chaining process, that is very similar to the process used to prove facts in logic programming. The process constructs a Bayesian network, in which each node is a ground term, i.e., a predicate with all of its arguments instantiated. The process begins by placing each of the query terms in the network. Whenever a term is placed in the network, it is matched with the heads of all the causal rules. If it matches the head of a rule, then each of the terms in the body of the rule is added to the network, if not already there.

Several issues must be addressed in order to make sure that the process produces a coherent network. First, the causal rules must be acyclic, so that no two ground terms can depend on each other. Second, the process must terminate. This is achieved by limiting the values of predicate arguments to ground terms in the knowledge base — the assumption is similar to the closed world assumption normally made in logic programming. Third, there can be no variables in a rule body that are not bound in the rule head. The reason is that a ground term must be produced for each possible binding of the variable, which typically can be any atom in the knowledge base. This is in contrast to standard logic programming, in which reasoning is performed on terms containing variables, and not solely on ground terms. A workaround to deal with the third issue is to use fully-known, non-probabilistic context predicates, that can be used to bind variables not in a rule head. For example, in the rule

$$\text{Has-Gene}(X, G) \text{ :- Parent}(X, Y), \text{ Has-Gene}(Y, G).$$

the `Parent` predicate functions as a context predicate, specifying the structure of the

family tree.

There are a number of differences between the KBMC framework in general, and the PLP framework in particular, and our approach. Syntactically, in the KBMC approach, the representation is centered around facts and rules, whereas our approach emphasizes objects, their properties, and the relationships between them. Our language is therefore more closely related to object-centered languages such as frame-based systems and object-relational database languages. Our language supports the ability to create classes and subclasses and to use an inheritance hierarchy. Another syntactic difference is that numbers in languages that use KBMC are typically associated with causal rules, and combination rules must be applied to translate these numbers into conditional probabilities, whereas for us the numbers are directly specified in CPFs, as in Bayesian networks.

The expressive power of the various KBMC languages varies. The more expressive ones, such as that of Goldman and Charniak, come with no guarantees as to whether the BN construction process terminates, or whether it is acyclic. On the other hand, the probabilistic logic programming languages are more limited in their expressive power than our language. In particular, they do not allow structural uncertainty: the context predicates used to bind variables in the body of a rule can be viewed as determining which objects are related to other objects. They must be fully known in the probabilistic logic programming framework.

From the point of view of semantics, the KBMC approaches can generally be viewed as procedural. The usual approach is to attempt to construct the BN, and if the BN is successfully constructed it defines a probability distribution over the variables of interest. The probabilistic logic programming languages rely on the closed world assumption to show that the BN construction is successful. In Section 6.3, we offered a semantics based on the closed world assumption as one possible interpretation of our language, but for the most part we have followed alternative semantics that does not make that assumption.

A major difference between our approach and that of all the KBMC languages is in the way probabilistic inference is performed. In the KBMC approach, inference is performed by constructing a BN and using standard BN inference algorithms. In

contrast, inference in our framework is performed within the structured models. As we have shown, a structured inference algorithm is able to exploit encapsulation of variables within objects and reuse of inference between objects, both of which have major computational benefits. In addition, none of the KBMC languages deal with issues involving inference with structural uncertainty.

9.2.3 Network Fragments

A more recent approach, related to KBMC but similar in spirit in some ways to ours, is the *network fragments* of Laskey and Mahoney [59, 65]. They provide network fragments for different aspects of a model, and operations for combining the fragments to produce more complex models. Network fragments exploit the same types of domain structure as do OOBNs. Because they allow complex fragments to be constructed out of simpler ones, models can be composed hierarchically. Similarly, because they allow the same fragment to be reused multiple times, they exploit the redundancy in the domain.

The main difference between the two approaches is that ours focuses on building structured models, while theirs focuses on exploiting the domain structure for the knowledge engineering process, but the constructed models themselves are unstructured. An analogy from programming languages is that network fragments are like macros, which are preprocessed and substituted into the body of a program before compilation. Our class models, on the other hand, are like defined functions, which become part of the structure of the compiled program. The advantages of the two approaches are comparable to those of their programming language analogues. Network fragments, like macros, are very flexible, since no assumptions need be made about the relationship between combined fragments. Our language, on the other hand, provides a stricter, more semantic approach to combining models. Like a structured programming language, it allows strong type-checking in the definition of models. The most important advantage of our approach compared to network fragments is that the models are themselves structured. As a result, we can exploit the domain structure for efficient inference, whereas in the network fragments approach a KBMC process

must be used, and inference is performed in the flat constructed BN. In addition, because the domain structure is an explicit part of our language, we can integrate uncertainty over the structure directly into the probability model.

9.2.4 Other Model-Based Approaches

There are a number of other approaches to building expressive probabilistic models that do not use KBMC. One recent approach is the *relational Bayesian networks* framework of Jaeger [49]. In his framework, a single BN is created in which each node represents an entire relation, rather than a ground term. The CPF of a node specifies how the entire value of the relation depends on the entire values of other relations. While the framework is quite expressive, it does not allow conditional independence relationships that hold between ground terms to be represented explicitly in the network. Since probabilistic influences generally flow between ground terms, representing these relationships explicitly is crucial to extending the advantages of BNs to relational domains.

Probabilistic Prolog [20] is a variation on the probabilistic logic programming idea. Here, probabilities are associated not with individual facts, but with clauses in the knowledge base, including Horn clauses. The probability of a fact is then interpreted as the probability that a set of clauses sufficient to prove the fact actually hold. Inference in this framework is not performed by constructing a BN, but rather by using standard Prolog theorem-proving methods.

Researchers from the inductive logic programming (ILP) community have also been concerned with integrating probabilities into logic programs. The goal of ILP is to induce a logic program from data. One of the major problems in ILP has been dealing with noisy data, and probabilities have been proposed as a means of addressing this problem. Muggleton [70] has presented *stochastic logic programs*, which are yet another variant on the probabilistic logic program theme. Stochastic logic programs, which are based on stochastic context free grammars, define a generative probability model that generates possible facts in a knowledge base. In some ways, stochastic logic programs are similar in flavor to the stochastic functional language we described

in [55], which forms the basis for much of our work. In particular, his work shares with ours the idea of using a generative model to define probability distributions over a rich space. However, while Muggleton shows how stochastic logic programs can be used to define rich probability models, he does not provide a method for performing probabilistic inference with these models.

Another extension of stochastic context free grammars has been developed by researchers working on Bayesian approaches to computer vision. In the Bayesian approach, the vision problem is presented as a problem in hierarchical probabilistic reasoning, with the representation of a scene at each hierarchical level depending probabilistically on the representation at the level above it. It is clear that in order to represent the complex types of relationships that hold between representations at different levels of abstraction, a rich probabilistic model is needed. Geman, Potter and colleagues [9, 29] have proposed *compositional systems*, an extension of stochastic context free grammars that can be used to define hierarchical probabilistic models. Like our work and Muggleton's, their framework uses the idea of specifying a rich probabilistic model through a generative process.

9.3 Miscellaneous

Other communities have also approached the issue of integrating logical and probabilistic representations. For example, researchers in the deductive database community have investigated a language in which probabilities are attached to the rules in the database [73]. This language, which is also called probabilistic logic programming, is quite different from the probabilistic logic programming variants described earlier. The approach is axiom-based, in that many probability distributions over the relevant facts are possible, and is more akin to the primarily logical approaches discussed in Section 9.1.

Wong et al. [93] have proposed using relational databases as a framework for implementing Bayesian networks. However, their framework is not really a combined logical and probabilistic representation language. Rather, a relational database is used to organize and perform the computations needed for inference in standard

BNs.

As this survey of related work shows, creating expressive probabilistic representation languages is a topic of interest in a variety of communities. We hope that our framework provides a strong combination of expressivity, semantic clarity and effective inference, that can provide the basis for work in a number of different areas.

Chapter 10

Conclusion and Future Work

10.0.1 Summary

In this thesis, we have presented a framework for probabilistic reasoning in complex systems. Our framework builds on the strengths of Bayesian networks: the notion of conditional independence is used to provide a natural, compact representation language; the language has clear probabilistic semantics, defining a probability distribution or probability measure over possible worlds; and the explicit representation of conditional independence relationships supports effective probabilistic inference algorithms. Our language also incorporates the advantages of relational representations: the ability to describe the world in terms of objects and the relationships between them; the ability to build a large model out of small modular components; and the ability to make general statements about objects, that can be reapplied in many situations.

We first described Object-Oriented Bayesian Networks, a probabilistic representation language for hierarchical systems. Every object in the hierarchy has an associated local probability model. Information about an object's environment is passed to it from its containing object. The environmental information can probabilistically influence the properties of its object, and the object can in turn influence other aspects of the environment. Local probability models are associated with classes of objects, and all instances of the same class have the same probability model. We showed that

OOBNs have clear probabilistic semantics, and define a unique probability distribution over the properties of all the objects in the hierarchy. We described an inference algorithm that exploits the object structure as well as the conditional independence structure traditionally exploited in BNs. Specifically, we exploited the fact that all the information that needs to be passed between an object and its containing object can be summarized in a relatively small interface, and the fact that reasoning can be reused between different objects of the same class.

We then described a more general language for relational probability models, that is capable of describing systems with a wide variety of relationships between objects, and systems with many interconnected objects. As in OOBNs, a local probability model is associated with each class of objects, with a property of an object depending probabilistically on other properties of the same object and on properties of related objects. These class models are augmented with a relational model, describing the objects in the system and the relationships between them. This approach of combining the probabilistic and relational components of the representation is very flexible, as it allows the same class probability models to be used in systems with very different configurations. We showed that the language of relational probability models has clear probabilistic semantics, in terms of defining a probability measure over possible worlds. We also described how the structured inference algorithm for OOBNs can be extended to the more general language, so as to allow us to exploit the object structure in these more general models.

We went on to consider extending the language of relational probability models, so as to allow an object to depend on multiple related objects in an aggregate manner. We showed that the reuse of class probability models, as well as symmetry, can be exploited to provide efficient inference with aggregate relationships. After discussing languages in which the number of objects related to any other object is known, we considered models with structural uncertainty, in which the number, identity or type of related objects is unknown. We showed that the structural uncertainty can be integrated directly into the probability model, thereby allowing the structural properties of a system and the individual properties of the objects within it to participate in the same probability model. As a result, BN inference techniques can be used to reason

about structural properties of a system configuration.

Next, we discussed a powerful extension to our framework, allowing recursive probability models, in which a property of an object can depend on an infinite chain of properties of other objects. We defined a semantics for this language in terms of the local conditional probability tables, and showed that every knowledge base has a model, but not necessarily a unique one. We defined a sound iterative approximation algorithm for recursive probability models, and showed how the object structure can be exploited to reuse computation between different iterations. We showed that using this method, the amount of work performed in each iteration eventually becomes constant for a large class of models.

Finally, we described an implemented system called SPOOK that allows a user to build and query relational probability models. We described three example applications, one to military situation awareness, another to diagnosis of a computer system, and the third to modeling a university. We presented experimental results that show that exploiting the object structure for probabilistic inference has beneficial effects, and that each of the aspects of structure exploited by our algorithm has a cumulative benefit.

10.0.2 Future Work

It is our hope that the advances presented in this thesis will lead to more widespread application of probabilistic reasoning systems. In order for this to happen, a number of issues need to be addressed. One such issue, that we touched on in Section 6.3, is the need for efficient inference algorithms for systems with many named, non-generic, inter-related objects. It appears that approximate inference algorithms will be needed, and it is an open question whether the best approach will be to apply existing BN algorithms or to develop new approximate algorithms that exploit the object structure.

We have not implemented all possible representational features in our language. There are a number of possible extensions, including the integration of more expressive logical reasoning formalisms, and more ways of describing structural uncertainty.

Some of these may turn out to be important in practice. For example, we believe that biased selection of an element from an intensionally specified set, described at the end of Chapter 6, is a common and naturally occurring phenomenon, and it will be important to model that kind of situation. The only way to find out which functionality is really important is to attempt to apply our framework to more real-world problems, and discover what the key limitations are in practice.

One very valuable feature of BNs is that they do not always have to be constructed by experts, but instead can be learned from data. In fact, they provide a very good method of combining expert knowledge with observed data. We would like our language to share this feature of BNs. We have done some work on learning relational probability models from data, including [56] where we showed how to learn the probabilistic parameters of such a model, and [26], where we learned aspects of the model structure. Learning RPMs has great potential, since it allows learning to be performed directly on a relational model, and the resulting learned model is also relational. This is in contrast to most traditional learning algorithms, including BN learning algorithms, which can only be applied to flat data, and for which the learned models are attribute-based. There is much work remaining to be done on learning our more expressive probability models.

Many real systems are dynamic, and a model of a dynamic system should take into account the system dynamics. As we described in Chapter 7, dynamic Bayesian networks extend standard BNs to deal with dynamic systems. In [27], we developed an extension of the OOBN language for representing dynamic models. Other work [13, 14] has shown that a structured representation of a dynamic system may lead to better approximate inference algorithms. We believe there is still a good deal of work to be done on the structured representation of complex dynamic models, in a way that cleanly and efficiently integrates the temporal and relational structure of a system.

Finally, we want to be able to use our probabilistic representation language not only for reasoning about an uncertain world, but also for making decisions in the face of that uncertainty. Influence diagrams [45] extend Bayesian networks to incorporate decision and utility nodes, and allow BN reasoning and inference algorithms to be used for making decisions. Influence diagrams suffer from the same limitations as

BNs that we have addressed in our thesis. We would like to include decisions and utilities into our structured probabilistic representation language, and exploit the system structure for more efficient decision making algorithms.

10.0.3 Conclusion

Knowledge representation is a field with a long and rich history. For the most part, it has developed in a purely logical framework, and has not been able to deal adequately with issues of uncertainty. Probabilistic reasoning methods, and Bayesian networks in particular, have been developed to address the issues of reasoning under uncertainty. However, the field has mainly focused on propositional representations, and issues of large scale probabilistic knowledge representation have for the most part been ignored.

There are many important lessons to be learned from both fields. From knowledge representation, we learn the importance of relational representations, that provide the flexibility and modularity needed to model large domains. We also learn the importance of making general statements, rather than explicitly having to model every single aspect of the world, and the importance of reasoning at the most general level possible — lifted inference in the first order logic setting.

From Bayesian networks we learn the value of a model-based rather than axiomatic approach to knowledge representation. Model-based approaches have two major advantages: they allow a model to provide a useful answer to any query, and inference is performed by well-specified operations on the model, rather than by search over the space of proofs. BNs also teach us the value of conditional independence as a key organizing feature of a model, and also the importance of having an inference algorithm that exploits the structure encoded in a model — the conditional independence structure in the case of BNs.

In this thesis, we have tried to develop a language that implements the lessons learned from both fields. We developed an object-based probabilistic representation language that is flexible and modular, and allows the modeler to make general statements about large classes of objects. At the same time, our framework remains model-based, just like BNs, and conditional independence is still a key organizing

feature of our models. Our inference algorithms take advantage of the different types of structure represented in the language: both the relevance information as encoded by conditional independence relationships, and the encapsulation resulting from the object structure. Our algorithms also exploit the fact that the language supports general statements by reasoning at the class level wherever possible. We believe that our language achieves a happy synthesis of logical and probabilistic representations, and provides a strong foundation for scaling up probabilistic reasoning to large real-world domains.

Bibliography

- [1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [2] S. Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability. *BIT*, 25:2–23, 1985.
- [3] S. Arnborg, D.G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal of Algorithms and Discrete Math*, 8:277–284, 1987.
- [4] F. Bacchus. *Representing and Reasoning with Probabilistic Knowledge*. MIT Press, Cambridge, MA, 1990.
- [5] F. Bacchus. Using first-order probability logic for the construction of Bayesian networks. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence (UAI-94)*, 1994.
- [6] F. Bacchus, A.J. Grove, J.Y. Halpern, and D. Koller. From statistical knowledge bases to degrees of belief. *Artificial Intelligence*, 87:75–143, 1997.
- [7] J. Banerjee, H.-T. Chou, J.F. Garza, W. Kim, D. Woelk, N. Ballou, and H.-J. Kim. Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems*, 5(1):3–27, 1987.
- [8] A. Becker and D. Geiger. A sufficiently fast algorithm for finding close to optimal junction trees. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence (UAI-96)*, 1996.

- [9] E. Bienenstock, S. Geman, and D. Potter. Compositionality, mdl priors, and object recognition. In *Advances in Neural Information Processing Systems 9 (NIPS-97)*, 1997.
- [10] G. Booch. *Object-oriented analysis and design with applications*. Addison Wesley, 1994.
- [11] A. Borgida and P. Patel-Schneider. A semantics and complete algorithm for subsumption in the CLASSIC description logic. *Journal of Artificial Intelligence Research*, 1, 1994.
- [12] C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller. Context-specific independence in Bayesian networks. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence (UAI-96)*, 1996.
- [13] X. Boyen and D. Koller. Tractable inference for complex stochastic processes. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI-98)*, 1998.
- [14] X. Boyen and D. Koller. Exploiting the architecture of dynamic systems. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, 1999.
- [15] R. Brachman, A. Borgida, D. McGuinness, P. Patel-Schneider, and L. Resnick. Living with CLASSIC: When and how to use a KL-ONE-like language. In John Sowa, editor, *Principles of Semantic Networks*. Morgan Kaufmann, 1991.
- [16] J.S. Breese. Construction of belief and decision networks. *Computational Intelligence*, 1992.
- [17] E. Charniak. *Statistical Language Learning*. MIT Press, 1993.
- [18] V.K. Chaudhri, A. Farquhar, R. Fikes, P. Karp, and J.P. Rice. A programmatic foundation for knowledge base interoperability. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, 1998.

- [19] P. Cheeseman. An inquiry into computer understanding. *Computational Intelligence*, 5(1), 1989.
- [20] E. Dantsin and V. Valkovsky. Abductive reasoning in probabilistic prolog. In *Second workshop of the INTAS-93-1702 project: Efficient Symbolic Computing*, 1996.
- [21] T. Dean and K. Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3), 1989.
- [22] R. Dechter. Bucket elimination : a unifying framework for probabilistic inference. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence (UAI-96)*, 1996.
- [23] D.L. Draper and S. Hanks. Localized partial evaluation of belief networks. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence (UAI-94)*, 1994.
- [24] H.B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.
- [25] A. Farquhar, R. Fikes, and J. Rice. The Ontolingua server: A tool for collaborative ontology construction. Technical report, Stanford KSL 96-26, 1996.
- [26] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1999.
- [27] N. Friedman, D. Koller, and A. Pfeffer. Structured representation of complex stochastic systems. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, 1998.
- [28] S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions and the Bayesian restoration of images. In M. A. Fischler and O. Firschein, editors, *Readings in Computer Vision: Issues, Problems, Principles, and Paradigms*. Morgan Kaufmann, 1987.

- [29] S. Geman, D. Potter, and Z. Chi. Composition systems. Technical report, Department of Applied Mathematics, Brown University, 1998.
- [30] S. Glesner and D. Koller. Constructing flexible dynamic belief networks from first-order probabilistic knowledge bases. In Ch. Froidevaux and J. Kohlas, editors, *Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty (ECSQARU)*. Springer Verlag, 1995.
- [31] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
- [32] R.P. Goldman and E. Charniak. A language for construction of belief networks. *IEEE Transactions on Pattern Recognition and Machine Intelligence*, 15(3):196–208, 1993.
- [33] A.J. Grove, J.Y. Halpern, and D. Koller. Random worlds and maximum entropy. *Journal of Artificial Intelligence Research*, pages 33–88, August 1994.
- [34] P.R. Halmos. *Measure Theory*. Springer Verlag, 1983.
- [35] J. Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 46, 1990.
- [36] J.Y. Halpern. Let many flowers bloom: a response to “An inquiry into computer understanding”. *Computational Intelligence*, 6:184–188, 1990.
- [37] J.Y. Halpern and D. Koller. Representation dependence in probabilistic inference. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, 1995.
- [38] D. Heckerman. A tutorial on learning with Bayesian networks. In M. I. Jordan, editor, *Learning in Graphical Models*. MIT Press, Cambridge, MA, 1998.
- [39] D. Heckerman and J.S. Breese. A new look at causal independence. Technical report, Microsoft Research MSR-TR-94-08, 1994.

- [40] D.E. Heckerman. An empirical comparison of three inference methods. In *Proceedings of the Fourth Workshop on Uncertainty in Artificial Intelligence (UAI-88)*, 1988.
- [41] J. Heinsohn. Probabilistic description logics. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence (UAI-94)*, 1994.
- [42] M. Henrion. Propagation of uncertainty in Bayesian networks by probabilistic logic sampling. In John F. Lemmer and Laveen N. Kanal, editors, *Uncertainty in Artificial Intelligence 2*, pages 149–163. Elsevier, 1988.
- [43] E. Horvitz, J. Breese, D. Heckerman, D. Hovel, and K. Rommelse. The Lumiere project: Bayesian user modeling for inferring the goals and needs of software users. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI-98)*, 1998.
- [44] E. Horvitz, S. Srinivas, C. Rouokangas, and M. Barry. A decision-theoretic approach to the display of information for time-critical decisions: The Vista project. In *Proceedings of SOAR-92*, 1992.
- [45] R.A. Howard and J.E. Matheson. Influence diagrams. In *Readings on the Principles and Applications of Decision Analysis*, pages 719–962. Strategic Decisions Group, Menlo Park, California, 1981.
- [46] HUGIN. Hugin newsletter. <http://www.hugin.dk/newsletters/nl070799.html>, july 1999.
- [47] T.S. Jaakkola and M.I. Jordan. Variational probabilistic inference and the QMR-DT network. *JAIR*, 1999. in press.
- [48] M. Jaeger. Probabilistic reasoning in terminological logics. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR-94)*, 1994.
- [49] M. Jaeger. Relational Bayesian networks. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence (UAI-97)*, 1997.

- [50] F.V. Jensen. *An Introduction to Bayesian Networks*. Springer Verlag, 1996.
- [51] M.I. Jordan, Z. Ghahramani, T.S. Jaakkola, and L.K. Saul. An introduction to variational methods for graphical models. In M.I. Jordan, editor, *Learning in Graphical Models*. Kluwer Academic Publishers, 1998.
- [52] S. Karlin and H.M. Taylor. *A first course in stochastic processes*. Academic Press, New York, 2 edition, 1975.
- [53] U. Kjærulff. dHugin: A computational system for dynamic time-sliced Bayesian networks. *International Journal of Forecasting*, 11:89–111, 1995. Special Issue on Probability Forecasting.
- [54] D. Koller, A. Levy, and A. Pfeffer. P-Classic: A tractable probabilistic description logic. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, 1997.
- [55] D. Koller, D. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, 1997.
- [56] D. Koller and A. Pfeffer. Learning probabilities for noisy first-order rules. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, 1997.
- [57] D. Koller and A. Pfeffer. Object-oriented Bayesian networks. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence (UAI-97)*, 1997.
- [58] D. Koller and A. Pfeffer. Probabilistic frame-based systems. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, 1998.
- [59] K. Laskey and S.M. Mahoney. Network fragments: Representing knowledge for constructing probabilistic models. In *Proc. UAI*, 1997.

- [60] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society*, pages 157–224, 1988.
- [61] D.B. Lenat and R.V. Guha. *Building Large Knowledge-Based Systems: Representation and Inference in the CYC Project*. Addison-Wesley, 1990.
- [62] Z. Li and B. D'Ambrosio. Efficient inference in Bayesian networks as a combinatorial optimization problem. *International Journal of Approximate Reasoning*, 11(1):55–81, 1994.
- [63] F.M. Donini M. Buchheit and A. Schaerf. Decidable reasoning in terminological knowledge representation systems. *Journal of Artificial Intelligence Research*, 1:109–138, 1993.
- [64] R. MacGregor. The evolving technology of classification-based knowledge representation systems. In J. Sowa, editor, *Principles of semantic networks*, pages 385–400. Morgan Kaufmann, 1991.
- [65] S.M. Mahoney and K. Laskey. Constructing situation specific Bayesian networks. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI-98)*, 1998.
- [66] S.M. Mahoney and K.B. Laskey. Network engineering for complex belief networks. In *Proc. UAI-96*, pages 389–396, 1996.
- [67] J.C. McCarthy. Programs with common sense. In *Teddington Conference on the Mechanization of Thought Processes*, 1958.
- [68] M. Minasi. *The complete PC Upgrade and Maintenance Guide*. SYBEX, 8 edition, 1997.
- [69] M.A. Morjaia, F.J. Rink, W.D. Smith, G. Klempner, C. Burns, and J. Stein. Commercialization of EPRI's generator expert monitoring system (gems). In *Expert System Application for the Electric Power Industry*, Phoenix, 1993. EPRI. Also: GE techreport GER-3790.

- [70] S. Muggleton. Stochastic logic programs. *Journal of Logic Programming*, 1999. Accepted subject to revision.
- [71] K. Murphy, Y. Weiss, and M. Jordan. Loopy belief propagation for approximate inference: an empirical study. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI-99)*, 1999.
- [72] R.M. Neal. Probabilistic inference using Markov chain Monte Carlo methods. Technical report, Dept. of Computer Science, University of Toronto CRG-TR-93-1, 1993.
- [73] R. Ng and V.S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1993.
- [74] L. Ngo and P. Haddawy. Answering queries from context-sensitive probabilistic knowledge bases. *Theoretical Computer Science*, 1996.
- [75] L. Ngo, P. Haddawy, and J. Helwig. A theoretical framework for context-sensitive temporal probability model construction with application to plan projection. In *Proc. UAI-95*, pages 419–426, 1995.
- [76] N.J. Nilsson. Probabilistic logic. *Artificial Intelligence*, 28(1):71–87, 1986.
- [77] J.B. Paris and A. Vencovska. On the applicability of maximum entropy to inexact reasoning. *International Journal of Approximate Reasoning*, 3:1–34, 1989.
- [78] H. Pasula, S. Russell, M. Ostland, and Y. Ritov. Tracking many objects with many sensors. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1999.
- [79] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.
- [80] J. Pearl. *Causality: Models, Reasoning and Inference*. Cambridge University Press, 2000.

- [81] C. Petalson. The BACK system : an overview. In *SIGART bulletin*, volume 2(3), 1991.
- [82] A. Pfeffer, D. Koller, B. Milch, and K.T. Takusagawa. SPOOK: A system for probabilistic object-oriented knowledge representation. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI-99)*, 1999.
- [83] D. Poole. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64(1):81–129, 1993.
- [84] D. Poole. The use of conflicts in searching Bayesian networks. In *Proceedings of the Ninth Conference on Uncertainty in Artificial Intelligence (UAI-93)*, 1993.
- [85] L.R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2), 1989.
- [86] M. Sahami, S. Dumais, D. Heckerman, , and E. Horvitz. A Bayesian approach to filtering junk e-mail. In *AAAI Workshop on Learning for Text Categorization*, July 1998.
- [87] L.K. Schubert. Comments on 'An inquiry into computer understanding' (by P. Cheeseman, with his reply). *Computational Intelligence*, 4(1):67–9, 1988.
- [88] R.D. Shachter and M. Peot. Simulation approaches to general probabilistic inference on belief networks. In *Fifth Workshop on Uncertainty in Artificial Intelligence (UAI-89)*, 1989.
- [89] L. Shastri. Default reasoning in semantic networks: a formalization of recognition and inheritance. *Artificial Intelligence*, 39(3), 1989.
- [90] K. Shoikhet and D. Geiger. A practical algorithm for finding optimal triangulations. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, 1997.
- [91] J.E. Shore and R.W. Johnson. Axiomatic derivation of the principle of maximum entropy and the principle of minimum cross-entropy. *IEEE Transactions on Information Theory*, 26(1):26–37, 1980.

- [92] H. Simon. *The Sciences of the Artificial*. MIT Press, 2nd edition, 1981.
- [93] C.J. Butz S.K.M. Wong and Y. Xiang. A method for implementing a probabilistic model as a relational database. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI-95)*, 1995.
- [94] S. Srinivas. A probabilistic approach to hierarchical model-based diagnosis. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence (UAI-94)*, pages 538–545, 1994.
- [95] H. J. Suermondt and G. F. Cooper. Probabilistic inference in multiply connected belief networks using loop cutsets. *International Journal of Approximate Reasoning*, 4:157–224, 1990.
- [96] B. Tessem. Interval probability propagation. *International Journal of Approximate Reasoning*, 7:95–120, 1992.
- [97] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1988.
- [98] M.P. Wellman, J.S. Breese, and R.P. Goldman. From knowledge bases to decision models. *The Knowledge Engineering Review*, 7(1):35–53, November 1992.
- [99] S. Wolfram. *The Mathematica Book*. Cambridge University Press, 4 edition, 1999.
- [100] Y. Xiang, D. Poole, and M.P. Beddoes. Multiply sectioned Bayesian networks and junction forests for large knowledge based systems. *Computational Intelligence*, 9(2):171–220, 1993.
- [101] N. L. Zhang and D. Poole. On the role of context-specific independence in probabilistic inference. In *Proc. IJCAI*, pages 1288–1293, 1999.