A New Approach to Model-Based Diagnosis Using Probabilistic Logic

Nikita A. Sakhanenko, Roshan R. Rammohan, George F. Luger
Dept. of Computer Science, Univ. of New Mexico
Albuquerque, New Mexico

Carl R. Stern
Management Sciences, Inc.
Albuquerque, New Mexico

Abstract

We describe a new approach to model construction using transfer function diagrams that are consequently mapped into generalized loopy logic, a first-order, Turing-complete stochastic language. Transfer function diagrams support representation of dynamic systems with interconnected components. We demonstrate how these diagrams provide interfaces to a context-sensitive probabilistic modeling system (COSMOS). As a result, interfaces as well as the notion of context underlying COSMOS are successfully used for model-based diagnosis. This paper describes transfer function diagrams and how they are incorporated into COSMOS. We illustrate our approach with a practical example taken from a "pump system" modeling problem.

Introduction

In this paper we describe a powerful new approach to model construction for diagnostic reasoning. Transfer function diagrams provide an intuitive, yet expressive way to capture domain knowledge, which is then mapped into a stochastic, first-order representation. Coupled with a context-sensitive probabilistic modeling system, this approach supports probabilistic model-based reasoning. Our example application is diagnosis of a mechanical pump system. However, the same approach can be adapted to other problems in probabilistic model-based reasoning, including behavioral modeling and prediction, fault recovery and repair, and high level control.

The probabilistic framework of graphical models makes this representation suitable for many noisy situations. It is also often easier to understand graphical models than raw joint probability distributions. On the other hand, graphical models do not focus on the knowledge engineering problem. We propose transfer function diagrams as a knowledge engineering representation that is automatically mapped into probabilistic models described in stochastic, first-order language (Pless *et al.* 2006). These diagrams serve as *interfaces* to "internal" probabilistic models.

These interfaces are utilized in our model-based framework called COSMOS (Context Sensitive Probabilistic Modeling System). A COSMOS hyper-model of a dynamic system consists of an ensemble of related probabilistic networks representing various operation modes along with a mechanism for switching between active networks based on detected context changes. COSMOS embodies an incremental failure-driven learning and repair mechanism based on abductive inference and EM parametric learning. Together these mechanisms generate new models better adapted to track system behavior in newly encountered environments. COSMOS is described in (Sakhanenko *et al.* 2007).

The probabilistic models comprising a COSMOS hyper-model differ from basic graphical models in that they contain explicitly defined interfaces. These interfaces represent the inputs and outputs to the model, the way in which the dynamic system interacts with its external environment. We rely on the notion of contexts and interfaces because we believe that for many dynamic systems and actual applications it would be unrealistic and computationally intractable to try to represent in a single model all the possible ways in which the dynamic system might interact with an external environment. Instead we exploit the notion of context, associating individual contexts with individual stable modes of interaction between the dynamic system and its current environment. Each stable mode of interaction is captured in a fixed model interface and a fixed range of probabilistic functional dependencies between the internal states and attributes of the network model and the values of the interface variables.

In applying this approach to model-based diagnosis we find a significant benefit from the flexibility provided by COSMOS context sensitive hyper-models. One of the well-known challenges for model-based diagnosis is the problem of interface-altering faults (IAFs). These arise when the components of a designed system begin interacting in ways that were not anticipated or intended by the system designer. Such faults, including current leakage and crosstalk, are common in electronic systems. They also occur in mechanical systems, e.g., when fluid leakage from a conduit results in corrosion or higher-than-normal friction in a nearby rotating device. IAFs present a challenge to those model-based diagnostic methods that derive the component interfaces solely from the as-designed model or the schematic. Such approaches afford no mechanism for deriving the altered inter-component interfaces resulting from faults. COSMOS' mechanisms provide automatic model switching and/or model repair when IAFs are detected, substituting new network models with appropriately modified inter-component interfaces and functional dependencies.

Model-based diagnosis and COSMOS

In this paper we extend the method and set of algorithms originally proposed by Srinivas for mapping device schematics into probabilistic models for diagnosis and repair (Srinivas 1995). Srinivas' method is based on a set of earlier approaches to model-based diagnosis, in particular consistency-based approaches that exploit design models and schematics (Davis & Hamscher 1992; deKleer & Williams 1989). We initially discuss these earlier approaches because they anticipate many of the issues and challenges later faced by Srinivas' probabilistic approach to model-based diagnosis.

Model-based diagnosis in the consistency-based variants is based on the use of *behavioral models* derived from *design models*. A behavioral model is a functional representation in which system behavior, including the resultant values of output variables, can be predicted purely from the internal states of the model and the values of the input variables.

Implicit in the specification of a system or component's input and output variables is the concept of an interface. While interfaces are an explicit part of many types of design models, especially electrical schematics, the explicit specification of interfaces is typically not part of the model specification for standard probabilistic graphical models such as Bayesian Networks, Markov Models, etc. (Pearl 1988). However one of the important contributions of Srinivas' translation scheme for mapping design models into Bayesian Networks is to show how interfaces in design models can be effectively mapped into Bayesian Network representations.

Consistency-based model-based diagnosis uses design models for diagnosis. A fault is defined as a deviation from the as-designed behavior. Consistency-based diagnostic algorithms try to localize the fault to a specific malfunctioning component but cannot identify the underlying character or cause of the fault. Therefore several authors have proposed adding fault models to consistency-based models in order to improve the efficiency of the algorithm and provide a more detailed characterization of the fault. In a common version of this approach, a fault is specified as an alternative internal state/operational mode of a system component resulting in a modification to the as-designed functional input-output behavior of the component.

In addition to providing a more detailed characterization of fault behaviors, fault models serve to improve the efficiency of the diagnostic algorithm by specifying relative likelihood of various faults, thereby providing valuable information for heuristic search strategies. However a fundamental limitation of the approach Design Models + Fault Models, is that the specification of fault modes as alternative internal states or operational modes of the component does not alter or extend the as-designed component interface. This representation is unable to manage cases where the fault in fact does change the interfaces between components by producing unanticipated interactions between components that diverge in functional signature from those specified in the as-designed component interfaces.

Model-based diagnosis in COSMOS supports a fundamentally more flexible approach to system modeling better able to address Interface-Altering Faults (IAF). COSMOS hyper-models incorporate an ensemble of graphical networks each capturing the behavior of the device in a different context. Like the extended conflict-based diagnostic approaches described above, COSMOS hyper-models incorporate network models for both normal and fault behaviors. The selection from a range of alternative models is correlated with the identification of context, where context may correspond to a normal operating state and region of the device or to a particular fault mode or to an environmental condition that is out-of-range with respect to the device's normal operating regions.

In contexts corresponding to fault states or out-of-range conditions, observed behavior may conflict with the intended or designed behaviors. These deviations from as-designed behaviors might be internal behavioral deviations that can be successfully localized to a single component fault mode. However they might also be new behaviors arising from unanticipated and unintended interactions between components. COSMOS hyper-models are capable of successfully adapting to the latter kind of non-localized changes or faults by learning new models incorporating both altered intra-component behaviors and interfaces. We now describe the elements of explicit interfaces of the COSMOS hyper-modeling system and the way it supports this flexible form of model-based diagnosis.

In the next section we briefly describe the logic-based probabilistic language used in COSMOS. We then give an example to illustrate our transfer-function approach specifying interfaces to probabilistic models. The mapping from the interfaces to probabilistic models is given in the following section. We then discuss related work and conclude.

Generalized Loopy Logic

The COSMOS hyper-modeling system is implemented in Generalized Loopy Logic. The logic-based representation gives us the flexibility to dynamically link-in knowledge as necessary. Generalized Loopy Logic (GLL) is the extension of a basic stochastic logic language (Pless *et al.* 2006). GLL is a logic-based, first-order, Turing complete stochastic modeling language. Sentences of GLL are Prolog-like Horn clauses with variables having stochastic distributions. To perform inference, GLL rules are mapped into a Markov random field. Loopy belief propagation (Pearl 1988) is used for inferencing (hence the name "Loopy"). As opposed to its basic predecessor, GLL can also use other iterative inferencing schemes such as generalized belief propagation and Markov chain Monte-Carlo. During the mapping to a Markov random field, two rules with the same head are usually combined using a product combining rule. GLL extends that further by employing other functions than product when combining. One of the major features of GLL is its natural support for parameter learning via a variant of EM algorithm. In addition, GLL supports dynamic models by using recursion and controlling the depth of unfolding of recursive rules when mapping into a Markov random field.

Our approach

Example: a pump system

In this section we introduce a simple mechanical system that we want to model. We use this example throughout the paper to illustrate how the domain expert captures the knowledge about the system, how that knowledge is then transformed into a stochastic model, and how we can use the model to solve diagnostic tasks.

Consider a pump system schematically depicted in figure 1. A water pump sucks liquid up from a reservoir

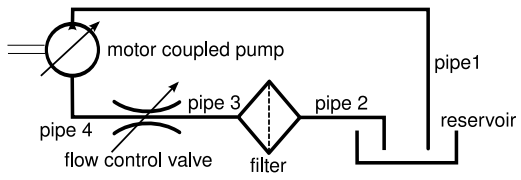

Figure 1: The diagram representing the pump system.

through a pipe (`pipe1`) and ejects the liquid into another pipe (`pipe4`). The pump is driven by an electrical motor. The liquid, that can contain emissions, is cleared by a filter and disposed back into the reservoir. The flow control modulates the liquid flow.

To diagnose the system, we install a number of sensors that detect current pressure, flow, and the emission state of the liquid at different locations, as well as indicating parameters such as the rotation rate of the pump and vibration near the motor. One important task is to detect when the filter gets clogged leading to possible cavitation in the system.

In the next sections we show how this system is modeled in GLL using intermediate transfer function diagrams as interfaces, making diagnostic tasks easier to perform.

Extended capabilities of transfer function diagrams

In this section we propose an input representation in the form of transfer function diagrams. We argue that this representation is quite similar to a functional model, typically consisting of interconnected components, created during the design process of an engineering system. Not only does this representation provide an intuitive way of engineering design making the task of modeling a system easier, but it also keeps the design process tractable by using a variant of an object-oriented approach. Modularity of the input representation provides clarity of the design and reusability of its components. Our approach extends Srinivas' mapping of functional schematics into probabilistic models (Srinivas 1995).

A transfer function diagram is a set of interconnected components. A component receives a set of inputs (**I**) and emits a set of outputs (**O**). There is also a set of internal state variables (**S**) of a component. Note that we assume that all the variables are discrete. For each output of the component there is a function computing the output that takes a subset of inputs and a subset of internal states as its arguments: $\forall O \in \mathbf{O}$, $\exists F$, $\exists\{I_1, \ldots, I_l\} \subseteq \mathbf{I}$, $\exists\{S_1, \ldots, S_m\} \subseteq \mathbf{S}$ such that $F : I_1 \times \ldots \times I_l \times S_1 \times \ldots \times S_m \rightarrow O$.

Figure 2 illustrates a component from a transfer function diagram modeling the pump system introduced earlier. The component represents a pipe. The boxes inside of the component correspond to functions, whereas ellipses correspond to component's internal states.

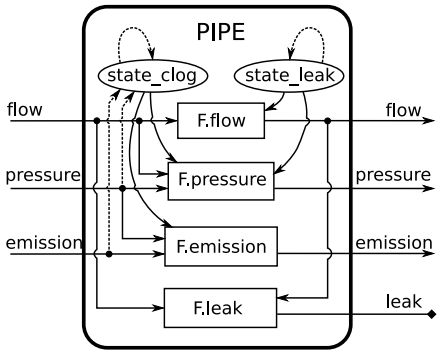

Figure 2: A component from the transfer function diagram representing a pipe of the pump system.

Allowing multiple outputs via objects

Typically, each component in a design model, e.g., functional schematics (Srinivas 1995), corresponds to a single function and, hence, emits only one output. This is very limiting, since usually components of engineering systems have multiple outputs, e.g., the pipe in figure 2 has four. Thus, we need to represent multiple outputs for this component.

A possible simple solution to this problem is to represent one component with many outputs as a set of components with one output each. However, in this case transfer function diagrams become very large and confusing for the knowledge engineer, which essentially diminishes the value of the diagrams as an intuitive input representation. Another straightforward way of handling the issue of multiple outputs is to represent them as a single Cartesian product. This, however, forces every output of the component to depend on every input, even if it is not necessary. As a result, the stochastic model constructed from such a diagram is overly complicated by redundant information.

We propose an object-oriented methodology for handling the issue of multiple outputs. Each component can be treated as an object containing multiple attributes representing different features of the component. The component `pipe` in figure 2, for example, contains several attributes such as the amount of emissions in the pipe, the pressure, etc. Each attribute is modeled by an appropriate function, e.g., the emission condition in the pipe is represented by a function that takes input emission, input pressure, and a current state of a pipe being clogged as its arguments. Note that each function box encapsulated inside of the component can be modeled by another transfer function diagram. The object-oriented representation accepts multiple functions, thus we have different outputs of the exterior object. This alternative gives us representational clarity.

Note also several other advantages of the object-oriented representation of transfer function diagrams. By specifying components of the diagram via objects we allow for reuse of model fragments. Moreover, we can replicate the inheritance mechanism from the object-oriented programming by combining some attributes of different objects into another object. After mapping into a probabilistic model, the stochastic parameters of the inherited attributes can be learned simultaneously by taking advantage of GLL parameter learning mechanism (see later).

Adding indicator variables

We distinguish two types of variables in a model: *operating* parameters and *indicator* parameters. Operating variables are those participating in operation of the system, e.g., engine speed, flow rate, etc. On the other hand, indicator variables, such as vibration near the motor, are not related to functioning of the system.

AI diagnostic representations such as transfer function diagrams focus only on operational behavior of the system and do not use indicator variables. We argue that by adding indicator parameters to the representation we gain a lot of information. Most notably, the inclusion of indicator variables supports assigning probability distributions that describe the operating state of the components.

When modeling a component of a system, we create a latent variable representing a state of the component that is not observed directly. However, we can classify the state based on its effects modeled by indicator variables. An indicator parameter provides evidence describing the current state of a subsystem that is essential for diagnosis. Note that the relationship between indicator variables and other parameters of the representation is not deterministic and is reminiscent of the relationship between observable and hidden variables in a hidden Markov model. An indicator variable is a component's output that is not used as an input to any other component. In figure 2 the component `pipe` contains one indicator variable: `leak`. It is computed by a function that takes the flow before the pipe and after the pipe as its arguments and represented as an output that goes nowhere.

We categorize indicator variables as two types: direct (sensory) and indirect (functional). When we have a sensor monitoring some aspects of the system (such as vibration near the motor in our example), it is directly represented by a sensory indicator variable. When needed to monitor some internal states of a subsystem for which no sensors are available, we use a function of inputs and outputs to model an indirect (functional) indicator variable. The output `leak` in figure 2 is a functional indicator variable computed as a ratio between the input and the output. We can see a functional indicator variable as a *virtual* sensor that provides the information whether a specific function (such as a ratio) within the subsystem is consistent with the data. Moreover, the task of the virtual sensor can be to indicate which function is currently describing the behavior of a subsystem most accurately. In the case of the indicator variable `leak` in figure 2, it represents a virtual sensor that tells whether the I/O ratio of the flow is equal to the degree of the leak of the pipe. Note that virtual sensors allow representation of the consistency of the data at the initial representation stage as opposed to factoring this information directly into stochastic models.

Consider a single complex indicator variable that takes inputs from all components of the system. Such a variable produces some general diagnostic information about the system. In this case, however, we encounter a significant complexity issue: if the system consists of a large number of components, then the indicator variable has a very large number of inputs (fan-in/fan-out problem). In order to address this problem we use a divide-and-conquer approach by introducing to the model more indicator variables each of which is associated with a small subsystem.

Coping with dynamic systems

One of the limitations of the functional schematics approach (Srinivas 1995) is the lack of an efficient way of representing dynamic systems. Srinivas identified how functional schemas can be adapted to specify time and dynamic feedback and proposed to map the extended diagrams into a variant of dynamic Bayesian networks. We also emphasize the importance of being able to monitor dynamic systems. As seen in the pump example earlier, a knowledge engineer must represent temporal relations between components as well as within components to diagnose such situations as cavitation in the system.

In order to explicate the temporal dynamics of the system, we explicitly specify all states of each component. For example, we identify two states of component `pipe`, `state_clog` and `state_leak`, representing the amount of clogging in the pipe and the presence of leaks (see figure 2). The temporal change of a component's state is, then, captured by a functional dependency on the values from the previous time steps. These dependencies are depicted with dotted arrows, e.g., in figure 2 three dotted arrows point to `state_clog` which means that the current state depends on the state's value and two inputs (`pressure` and `emission`) from the previous time step.

In figure 2 all the temporal connections (dotted arrows) within the component `pipe` represent local dynamics. However, we can also use temporal links outside of a single component representing more global dynamics between components that are not directly connected to each other.

Once the transfer function diagram is complete, its components are mapped to GLL rules. Since a GLL program represents classes of probabilistic models, switching to GLL rules provides the modeling system with additional power for capturing dynamic processes. Recursive rules of GLL, for instance, lend themselves nicely to representing a potentially infinite structure where some variables have a self-dependency over time. In the next section we describe this mapping to GLL rules in more detail.

Combining transfer-function diagrams with GLL

Once a domain expert has specified every system component within a transfer-function diagram, it is converted into a GLL program for further inferencing. In this section we specify the mapping rules guiding this conversion.

First, each function of every component in the diagram is mapped into a GLL sentence as shown in figure 3. Every

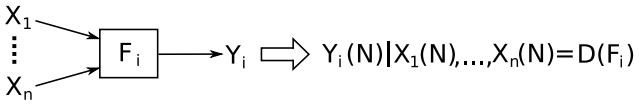

Figure 3: Mapping of a transfer function into a GLL rule.

input and the output of a function corresponds to a variable in GLL. Note that N stands for the current time step of the system, thus function F_i has an instant effect (the output is in the same time step as the input). Additionally, $D(F_i)$ stands for the probability distribution corresponding to function F_i, provided by an expert. Note that the functions producing indicator variables are handled similarly.

Second, each state of every component in the diagram is mapped into a GLL rule according to figure 4. Note that

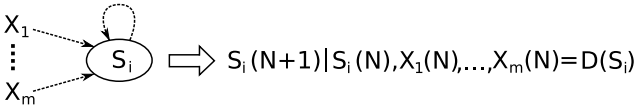

Figure 4: Mapping of a component's state into a GLL rule.

temporal influences on the state are easily described by a recursive GLL rule. We use $D(S_i)$ to denote the probability distribution corresponding to the function representing the temporal change of state S_i.

Third, connections between neighboring components are included in the corresponding GLL program according to figure 5. If the output O_i of component C_i is an immediate

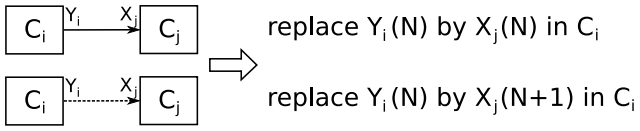

Figure 5: Mapping connections between components into a GLL program.

(in the same time step) input I_j to component C_j, then for the GLL rule representing the function of C_i producing O_i, replace $O_i(N)$ with $I_j(N)$. Similarly, when the output O_i is an input I_j at the next time step, then we replace corresponding $O_i(N)$ with $I_j(N+1)$. Note, again, how recursive rules of GLL easily capture the time change in the system.

During the mapping of the transfer function diagram into a GLL program the deterministic function, specified by a domain expert as a matrix, is straightforwardly transformed into a probability distribution table with zeros and ones. Moreover, the noise and the rate of change can be simulated by adding a probabilistic bias to the deterministic function during the mapping. It is possible in GLL to omit specification of a probability distribution of a sentence by marking it as learnable. The GLL system uses EM-based learning mechanism to infer the distribution from data. Therefore, by using functions from the expert knowledge as an initial approximation of the system and, then, utilizing the learning capabilities of GLL, the model of the system is further refined to closer represent the domain. This is another advantage of combining transfer function diagrams with GLL.

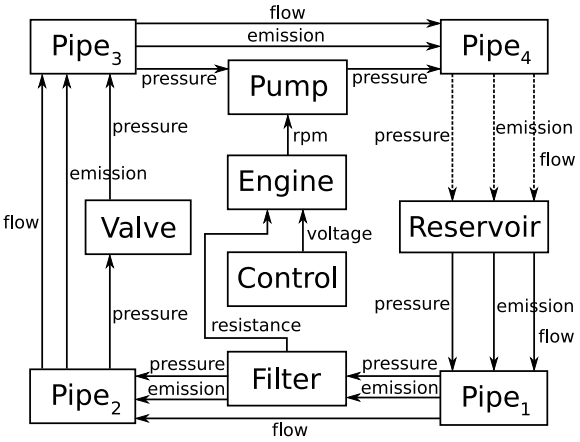

Figure 6: The general transfer function diagram of the pump system.

In the pump system example, the transfer function diagram (figure 6) is mapped into a GLL program:

```
filter_state<-{ok,soso,bad}
engine_vibration<-{no,low,med,hi}
..snip..
filter_state(N+1)|filter_in_emission(N),
                  filter_state(N)=
[[[1,0,0],[.95,.05,0],..snip..,[0,0,1]]]
engine_vibration(N)|engine_in_electro(N),
                    engine_state(N),
                    engine_in_resistance(N)=
[[[[1,0,0,0],..snip..,[0,0,.05,.95]]]]
```

After specifying initial conditions of the system, we perform various forms of analysis, e.g., we can submit a query `filter_state(4)?`. GLL inferences over the model represented by stochastic rules instantiated with initial parameters. If the initial parameters correspond to a normal operation of the system, then the result of the query is

```
(filter_state 4) [.923, .073, .004]
```

indicating that at time 4 the filter will be clean with probability .92. As we increase the emission in the pump system, we can observe the state of the filter change to more clogged, e.g., when emissions are high the result of the query is

```
(filter_state 4) [.0, .239, .761]
```

meaning that the filter is dirty with probability .76.

Related work

For the knowledge engineering component of modeling, two frameworks are often used: logic-based and probabilistic. First-order logic captures relationships between the entities in a domain (Poole 1988) and has clear *declarative semantics* independent from its *operational semantics*. While this allows domain experts to focus purely on the application, logic-based systems are unsuitable for representing uncertainty and cumbersome in generalizing evidence. Probabilistic graphical models (Pearl 1988) handle uncertainty

and noise, and support stochastic inference. However, being propositional in nature, they are not suitable for expressing first-order relations as well as time-dependent or recursive structures. Recent research (Ngo & Haddawy 1997; Kersting & DeRaedt 2000; Getoor *et al.* 2001; Pless *et al.* 2006; Richardson & Domingos 2006) combines these two frameworks to overcome these limitations. These systems have a sound declarative semantics independent of the inference algorithm and an ability to represent uncertainty. Yet, representing complex models with first-order rules itself can be daunting. However, complex systems can be described via functional schematics and automatically mapped to Bayesian networks (Srinivas 1995).

Other methods for tackling the knowledge engineering problem are decomposition and knowledge induction. Data-centric methods for the induction of causal links in graphical models (Lam & Bacchus 1994; Twardy *et al.* 2004) are limited in scalability and reliability in sparse-data situations. Knowledge engineering still relies heavily on contributions from domain experts (Pradhan *et al.* 1994). Approaching these problems with decomposition-based methods, as is offered through object-oriented design, aids usability.

Object-Oriented Bayesian Networks (Koller & Pfeffer 1997) allow complex domains to be described in terms of inter-related entities. This approach allows the encapsulation of variables within an object enabling the reuse of model fragments in different contexts. Similar object-oriented approaches focus on the modularization of the knowledge representation (Langseth & Bangso 2001; O. Bangsø 2004; Laskey & Mahoney 1997) detailing how large networks can be woven together from smaller, coherent components.

Current status and future work

Work on COSMOS to this point has focused on developing formalism and algorithms, and building an initial implementation. This implementation has been tested and run on simulated data sets for a pump system. The target for this implementation is live data from an actual pump system testbench. This bench is equipped with a variety of electrical and mechanical sensors, including a number of vibration sensors, at various positions. Within the next few months we expect to test our system on actual testbench data to demonstrate the value of COSMOS' context sensitive model switching mechanisms in providing enhanced capabilities in diagnosis and prediction. In the medium and longer term, we are planning experiments to demonstrate COSMOS' capabilities with respect to model adaptation and learning. We hope soon to validate COSMOS' potential to automatically construct new models with interfaces that accurately track the unplanned destructive interactions that sometimes arise in the use of compromised devices.

Acknowledgements

We thank the NSF (115-9800929, INT-9900485) and the US Navy (SBIR N00T001, STTR N0421-03-C-0041) for earlier support in the design of the Loopy Logic language. The research presented in this paper is partially funded by an Air Force Research Laboratory SBIR contract (FA8750-06-C0016). We thank Daniel Pless and Chayan Chakrabarti for many helpful discussions.

References

Davis, R., and Hamscher, W. C. 1992. Model-Based Reasoning: Troubleshooting. *Readings in Model-Based Diagnosis* 3–24.

deKleer, J., and Williams, B. C. 1989. Diagnosis with Behavior Modes. 1324–1330. Proc. of IJCAI.

Getoor, L.; Friedman, N.; Koller, D.; and Pfeffer, A. 2001. Learning Probabilistic Relational Models. *Relational Data Mining* 307–335.

Kersting, K., and DeRaedt, L. 2000. Bayesian Logic Programs. 138–155. Proc. of 10th Int. Conf. on ILP.

Koller, D., and Pfeffer, A. 1997. Object-Oriented Bayesian Networks. 302–313. Proc. of the 13th Conf. on UAI.

Lam, W., and Bacchus, F. 1994. Learning Bayesian Belief Networks: An Approach Based on the MDL Principle. *Computational Intelligence* 10:269–293.

Langseth, H., and Bangso, O. 2001. Parameter Learning in Object-Oriented Bayesian Networks. *Annals of Mathematics and Artificial Intelligence* 32(1–4):221–243.

Laskey, K., and Mahoney, S. 1997. Network Fragments: Representing Knowledge for Constructing Probabilistic Models. 334–340. Proc. of the 13th Conf. on UAI.

Ngo, L., and Haddawy, P. 1997. Answering queries from context-sensitive probabilistic knowledge bases. *Theoretical Computer Science* 171(1–2):147–177.

O. Bangsø, J. Flores, F. V. J. 2004. Plug and play object oriented Bayesian networks. 457–467. LNAI 3040. Proc. of the 10th Conf. of the Spanish Assoc. for AI.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

Pless, D. J.; Chakrabarti, C.; Rammohan, R.; and Luger, G. F. 2006. The Design and Testing of a First-Order Stochastic Modeling Language. *International Journal on Artificial Intelligence Tools* 15(6):979–1005.

Poole, D. 1988. Representing knowledge for logic-based diagnosis. 1282–1290. Proc. of International Conference on 5th Generation Computing Systems.

Pradhan, M.; Provan, G.; Middleton, B.; and Henrion, M. 1994. Knowledge Engineering for Large Belief Networks. 484–490. Proc. of the 10th Conf. on UAI.

Richardson, M., and Domingos, P. 2006. Markov Logic Networks. *Machine Learning* 62(1–2):107–136.

Sakhanenko, N. A.; Rammohan, R.; Luger, G. F.; and Stern, C. R. 2007. A Contex-Partitioned Stochastic Modeling System with Causally Informed Context Management and Model Induction. Proceedings of IICAI-07.

Srinivas, S. 1995. *Modeling techniques and algorithms for probabilistic model-based diagnosis and repair*. Ph.D. Dissertation, KSL, CS Dept., Stanford University.

Twardy, C.; Nicholson, A.; Korb, K.; and McNeil, J. 2004. Data mining cardiovascular bayesian networks. Technical Report 165. School of CSSE, Monash University.