The Object-oriented Tool MOTES$_{DM}$ and its Use for Diagnosis of Sewage Plant Failures

J. Schönwälder M. Hofmann H. Langendörfer
TU Braunschweig
Institut für Betriebssysteme und Rechnerverbund
Bültenweg 74/75
D-3300 Braunschweig
Tel.: [0049] 531 391 3248
UUCP: ...mcsun!unido!tubsibr!schoenw

Abstract

In this paper we present the tool MOTES$_{DM}$ which is used for knowledge based diagnosis of a sewage plant. MOTES$_{DM}$ supports hybrid knowledge representation that allows us to adequately express knowledge about the structure of the sewage plant, knowledge about well understood purification processes and associative knowledge for the diagnosis of the sewage plant. The knowledge representation formalisms make use of object-oriented concepts since classes and inheritance promote a modular design of knowledge bases. In our paper we describe the object-oriented knowledge representation, we discuss the implementation of our tool in Smalltalk-80 and we review the development of a prototypical knowledge base for a sewage plant in Salzgitter-Bad (FRG).

1 Introduction

The number of sewage plants using a mechanical and a biological purification stage to clarify the sewage is continually growing. These sewage plants are locations where rather complex biological and chemical processes take place. While the normal activities necessary for a regular management of a sewage plant can be accomplished by average unspecialized personnel, specialists like civil engineers are needed in case of failures.

Failures may be caused by biological or chemical processes as well as mechanical defects. Very often, the detection of the cause of a failure can only be done on the background of expert knowledge, while the removal of the failure's cause can be carried out by the filter plant's staff. Since in small- and medium-sized plants a civil engineer is usually not a member of this personnel due to the high costs of his or her employment, a lot of time oftenly passes until the cause of a failure is detected. This can lead to negative influences on the quality of the water leaving the sewage plant.

In 1988, the German Ministry for Research and Technology (BMFT) decided to promote a joint project between the institute for operating systems and computer networks and one institute for civil engineering of the University of Braunschweig (FRG).[1] Main goal of the project is the development of a system that can be used on small- and medium-sized sewage plants to aid the staff of the sewage plant in the following areas:

[1] The project is granted under number 02WA8828/7.

Figure 1: Concept of MOTES_{DM}

Figure 1 shows the concept of MOTES_{DM}. Measurings taken at the technical system are transmitted to the model of the system. The model collects this data and derives new data out of the known state of the systems. Therefore we use a constraint network that defines the correct behavior of the technical system. Faults in the technical system enter the model in form of data that do not fit with the correct behavior of the system. The propagation of such data causes contradictions in the constraint network. These contradictions activate a rule interpreter that uses associative knowledge encoded in rules to find the fault. In the following sections we will describe the representation formalisms in more detail.

[2] Modelling Technical Systems for Diagnosis and Monitoring

Figure 2: Structure of the effluent

2.1 Component Description

The knowledge about the structure of the technical system can be expressed using the component description language of MOTES_{DM}. The resulting model structure is used to store data representing the state of the technical system. We designed a component description language that distinguishes between components, component classes, aggregates and aggregate classes:

- Components represent elementary objects of the technical system. The state of a component is given by its attributes. Every component is an instance of a component class.

- Component classes are used to describe similar objects. Every component class may have several instances. Component classes can be specialized by making one component class a subclass of another component class. The subclass inherits all properties of the superclass.

- Aggregates are compound objects. They may enclose other components and aggregates. Every aggregate is an instance of an aggregate class.

- Aggregate classes describe similar aggregates. Aggregate classes can be structured hierarchically analogous to component classes.

With these four elements we can express three relationships. Between an aggregate and its parts exists a *part_of* relationship, between a component or an aggregate and its class exists an *instance_of* relationship, and between hierarchical arranged classes exists a *specialization_of* relationship. Thus we can use the abstraction mechanisms aggregation and generalization that are of great importance in any system that models relationships between (technical) objects [BB84][SS77].

The mentioned relationships allow the inheritance of constraint definitions (see 2.2) and attribute specifications. Attribute specifications determine the data type of an attribute, when and how the system may ask the user about the value of an attribute etc.. Attribute specifications are always bound to a class or an instance. They will be inherited along *specialization_of* and *instance_of* relationships towards the instances of all subclasses. Since class specializations can require the change of an inherited attribute specification we allow the overloading of attribute specifications for subclasses and even for individual instances.

As an example figure 2 shows the part of the model-structure we have defined for the effluent of the sewage plant. Components

Aggregate addClass: 'Drain'
specializationOf: ''
withPartClasses: 'Confluence MeasuringApparatus Water '
withAttributes: 'nh_4nThreshold no_2nThreshold turbidnessThreshold highWaterQuantity'
elucidation: 'A Drain is a part where streams coming from two basins leave the plant.'

Aggregate add: 'Effluent'
instanceOf: 'Drain'
withParts: 'EffluentConfluence EffluentMeasuringApparatus WaterInEffluent'
elucidation: 'The water leaves the sewage plant through the effluent.'

Figure 3: Definition of *Drain* and *Effluent*

are framed in rectangles, aggregates in ovals. Classes are shadowed. Figure 3 shows the definition of the aggregate class *Drain* and its instance *Effluent* by our component description language.

2.2 Constraints

Constraints are a useful programming paradigm because they allow programmers to state declaratively a relation that is to be maintained. Applications of constraint based systems reach from simulation systems to user interface toolkits [FMB90]. In our tool we use constraints for describing the behavior of components and aggregates. A constraint consists of a set of attributes and a relation upon these attributes. In MOTES_{DM}, we use constraints to compute values for unknown attributes from a set of given attributes and to test the consistency of values.

By sharing attributes among different constraints we get a constraint network in which we propagate data received from the technical system. There are a lot of known propagation techniques for constraint networks with different properties [Mes89]. Currently we use the value propagation technique which works similar to the Blue algorithm of ThingLab [FMB90]. The advantage is that we can handle contiguous variables and that every local consistent solution is also global consistent. The drawback of this method is that not all global consistent solutions are detected [VV87].

According to the object-oriented approach of MOTES_{DM} we distinguish between constraints and constraint classes. Every constraint is an instance of a constraint class. A constraint class defines a generic relation between abstract attributes. Instances of this class establish this relation upon a set of concrete attributes. The introduction of this concept allows the inheritance of constraint definitions along *specialization_of* and *instance_of* relationships.

Further we distinguish between predicate constraint classes and constructive constraint classes. Instances of constructive constraint classes allow the computation of unknown attribute values from known values. Constructive constraint classes are defined by a set of functions that calculate unknown values. Instances of predicate constraint classes can not be used to derive unknown attribute values. Instead, they are used to describe relations that may hold or not. They are defined by an expression that returns a logical value.

Constraint addClass: 'Adder'
variables: 'a b c'
functions: #('b+c' 'a-c' 'a-b')
elucidation: 'This constraint class defines the generic adder $a = b + c$.'

Confluence **addConstraint:** 'ConfluenceQuantityConstraint'
instanceOf: 'Adder'
variables: #('output' 'inputLeft' 'inputRight')
elucidation: 'The output of a confluence is the sum of both input streams.'

Figure 4: Constructive constraints

Figure 4 shows the definition of the constraint class *Adder* and the definition of *ConfluenceQuantityConstraint*. The constructive constraint class *Adder* is defined by three functions that simply return the value of the third variable if two variables are known. In our implementation, functions are defined by Smalltalk-80 expressions. Thus we can build rather complex constraints. The definition of *ConfluenceQuantityConstraint* will be inherited along *specialization_of* and *instance_of* relationships since *Confluence* is an aggregate class. Therefore this constraint must hold for the instance *EffluentConfluence* in figure 2.

Constraint addClass: 'GreaterThan'
variables: 'a b'
predicate: 'a >= b'
elucidation: 'This constraint is satiesfied if $a >= b$ holds.'

Confluence **addConstraint:** 'ConfluenceGreaterThanLeft'
instanceOf: 'GreaterThan'
variables: #('output' 'inputLeft')
elucidation: 'The output stream is always bigger than the left input stream.'

Confluence **addConstraint:** 'ConfluenceGreaterThanRight'
instanceOf: 'GreaterThan'
variables: #('output' 'inputRight')
elucidation: 'The output stream is always bigger than the right input stream.'

Figure 5: Predicate constraints

In figure 5 we have defined the predicate constraint class *GreaterThan*. The definition of *ConfluenceGreaterThanLeft* and *ConfluenceGreaterThanRight* makes sure that the direction of flow can not be reversed for all instances of class *Confluence*.

An important property of our constraint network is that new data may enter and overwrite existing data in the model of the technical system. When such an event occurs, we expect MOTES_{DM} to guarantee that all affected attributes are updated. Therefore every attribute value is stored with a time stamp, a set of assumptions used to compute the value and a set of attributes depending directly on the value. These informations are used to collect all attributes depending on a specific attribute value at a given time, to update these attributes and to reevaluate all affected constraints.

Rule addContext: 'Turbidness'
specializationOf: ''
predicate: '(WaterInEffluent turbidness valueAt: currentTime) > (Effluent turbidnessThreshold valueAt: currentTime)'
elucidation: 'Context Turbidness is active if the threshold value is exceeded.'

Rule add: 'WaterQuantity'
context: 'Turbidness'
premise: '(Effluent highWaterQuantity valueAt: currentTime)'
conclusion: 'OutputWindow cr; show: ''The high quantity of water could have caused the turbidness.'' '
elucidation: 'High quantity of water can cause turbidness.'

Figure 6: Contexts and Rules

2.3 Rules

The associative knowledge for model state interpretation is represented in rules. Rules are attached to contexts[3] in order to structure the rule base. A context is a set of rules with an additional predicate that defines if the context is active or not. Only rules in active contexts are considered by the rule interpreter.

Different levels of abstraction can be expressed by making one context a more specific context of another context. In general, one can define a hierarchy of contexts. A rule in a more specific context becomes active, if its more general contexts are active and if the predicate of the context evaluates to true.

[3] The name context originates from the expert system community. A context corresponds to a rule class as it is implemented.

Figure 7: Class hierarchy of MOTES_{DM}

Rules consist of a name, a premise, a conclusion and an elucidation. The premise is a logical expression that defines when a rule can be applied. The conclusion will be evaluated if the context of the rule is active and the premise of the rule evaluates to true. Thus our current rule interpreter is forward chaining.

The reason why we use a forward chaining rule interpreter is that there is no need to do goal directed search in our domain. Instead a data driven search is sufficient to determine possible causes of a fault by a given state of the model since we are interested in finding all possible causes of a failure. Other chaining strategies like backward chaining would be needed if we were looking for exactly one cause explaining the observed failure. But in our domain we do not have knowledge suited to evaluate competing diagnoses.

Figure 6 shows the definitions of the context *Turbidness* and the rule *WaterQuantity*. The context *Turbidness* becomes active if the turbidness of the water in the effluent exceeds the threshold defined in the aggregate *Effluent*. The rule *WaterQuantity* checks the attribute *highWaterQuantity* which is of type logic. If the value of this attribute is true the given text will be displayed in the output window.

3 Implementation

MOTES_{DM} is implemented in Smalltalk-80 [GR83] on a Sun 3/60 Workstation. The pure system consists of 45 classes grouped in six categories. While loading a knowledge base we generate application dependent classes. This means that the class hierarchies of a knowledge base are converted and added to the Smalltalk class hierarchy.

As an example figure 7 shows a part of the MOTES_{DM} hierarchy after loading a knowledge base containing all knowledge base items mentioned in this paper. The upper half of figure 7 contains the main classes implementing the knowledge representation formalisms and the lower half shows how the class hierarchies of the representation formalisms are integrated in the Smalltalk-80 hierarchy.

An important property of knowledge based tools in general is that variables, constraints and premisses of rules are often unknown. To implement such systems one likes to use a logic supporting the three values *true*, *false* and *unknown*. Therefore we decided to reimplement the Smalltalk-80 classes *True* and *False*, to add the class *Unknown* and to change the class *Boolean* to *Logic*. Our goal was to evaluate

Figure 8: Sewage plant Salzgitter-Bad

formula like

$$\neg(\mathit{false} \wedge \mathit{unknown}) \wedge (\mathit{true} \vee \mathit{unknown})$$

to *true* [Cod79] and to add new conditional statements like

```
ifTrue:     trueBlock
ifFalse:    falseBlock
ifUnknown:  unknownBlock.
```

Unfortunately the methods of the Smalltalk-80 classes *True* and *False* are in-line compiled in the Smalltalk-80 Bytecode. So we had to make alterations in some compiler classes that prevent the compiler to generate in-line code for our logic classes. After having recompiled the entire Smalltalk system to eliminate all in-line code for logical expressions we got a system running two times slower than the original system. Therefore we decided to recompile only those classes that are really using our extended logic.

4 The Prototype

While developing MOTES_{DM} we designed a knowledge base for the sewage plant in Salzgitter-Bad (figure 8). Therefore we acquired knowledge from a colleague of the Institute for Sanitary Engineering at our university. The result of our activities was a heap of knowledge and different knowledge sources that had to be structured. The distribution of data and the various elements of knowledge caused severe problems during our project. Therefore we are now investigating the facility of an integrated management of knowledge sources and support of knowledge acquisition methods by using a hypertext system [HSL90].

After having finished the first knowledge acquisition cycle we used the tool MOTES_{DM} to implement the knowledge. The foundation of the knowledge base is a structural description of the sewage plant in Salzgitter-Bad that is assembled out of 136 components and aggregates that are instances of 55 component or aggregate classes. Our aggregates can be devided in aggregates representing the physical structure of the plant and aggregates used to describe functional units.

On top of the component description we defined constraints that

- describe dependencies between water and sludge quantities in the sewage plant,
- define the rotor steering mechanism in the biological basin,
- describe chemical properties of sewage,
- establish stability conditions for biological purification processes.

Knowledge about ammonia concentrations and effluent sludge is packed in suitable contexts and appertaining rules. We used contexts to represent diagnoses and appropriate subcontexts for refined diagnoses. Therefore we could keep the rule base very small.

We tested our prototype using data that were measured on the sewage plant Salzgitter-Bad during summer 1989. The ability to write Smalltalk-80 processes allowed us to simulate the asynchronous behavior of the measuring instruments. The prototype detected faults and generated possible causes.

5 Conclusion

With the development of this prototype, we have reached two goals: On the one hand we could demonstrate that our approach to use artificial intelligence techniques is reasonable in the present task. On the other hand we were able to design an easy modifiable knowledge base. This is due to separation of structural knowledge from associative knowledge and the use of object-oriented concepts that increase abstraction.

Through the implementation process we found that Smalltalk-80 is a useful tool for prototyping approaches. The Smalltalk-80 language in combination with the integrated programming environment allowed us to make even conceptual changes to the system without getting lost in a mass of details. This ability is sometimes summarized as fearless programming [DM87].

The development of our prototype finished. Although MOTES$_{DM}$ showed to be useful for the development of a prototype for the sewage plant in Salzgitter-Bad, there are still a lot of things to be done in order to get a more universal tool suited for other industrial applications. For instance we have to develop a user interface that is easy to use. Another important issue is the integration of a database system. Further we would like to improve the constraint system so that it supports time dependent constraints and backtracking for attributes with finite domains. The rule interpreter should be improved to support backward chaining and arbitrary combinations of forward and backward chaining in order to get a tool suited for a broader spectrum of applications.

References

[BB84] D.S. Batory and A.P. Buchmann. Molecular Objects, Abstract Data Types and Data Models: A Framework. In *Proc. VLDB '84*, pages 172–184, Singapur, 1984.

[Cod79] E.F. Codd. Extending the Relational Database Model. *ACM Transactions on Database Systems*, 12, 1979.

[DM87] J. Diederich and J. Milton. Experimental Prototyping in Smalltalk. *IEEE Software*, pages 50–64, May 1987.

[FMB90] B.N. Freeman-Benson, J. Maloney, and A. Borning. An Incremental

Constraint Solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80. The Language and its Implementation.* Addison Wesley, 1983.

[HSL90] M. Hofmann, U. Schreiweis, and H. Langendörfer. An Integrated Approach of Knowledge Acquisition by the Hypertext System CONCORDE. In *Proc. ECHT '90*, Paris, November 1990.

[Mes89] P. Meseguer. Constraint Satisfaction Problems: An Overview. *AI Communications*, 2(1):3–17, March 1989.

[SHL91] J. Schönwälder, M. Hofmann, and H. Langendörfer. Fault Diagnosis of a Sewage Plant. In *Proc. 7. IEEE Conference on Artificial Intelligence Applications*, Miami, February 1991. IEEE Computer Society Press.

[SS77] D.C.P. Smith and J.M. Smith. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems*, 2(2):105–133, June 1977.

[VV87] A. Voss and H. Voss. Formalizing Local Constraint Propagation Methods. Arbeitspapiere der GMD 248, GMD, May 1987.