Object-Oriented Modeling in Model-Based Diagnosis.

Object-Oriented Modeling in Model-Based Diagnosis

Karin Lunde[*]
R.O.S.E. Informatik GmbH, Schloßstr. 34, D-89518 Heidenheim

October 13, 2000

1 Introduction

In the life cycle of a product, engineers have to perform many analysis tasks, from design layout, over system simulation and risk analysis, to diagnosis during product operation. Most analysis tools are specialized for just a few of these analysis tasks. This forces the engineer to work with a whole tool suite using various model formats (often even different models) for the same product. But modeling a technical system is a kind of art — it requires both expert domain knowledge and the ability to structure a problem and to find out the appropriate level of abstraction for it. It is a creative, time-consuming and expensive task. To increase efficiency in product development, it would be desirable to have analysis tools which reduce the modeling effort as far as possible, by providing means to reuse existing models, and by offering multiple analyzes of a product on the base of a single product model.

For programming languages like Java or Smalltalk, the advantages of object-oriented concepts are well-known. They can be employed to support hierarchical structuring of problems, to allow wide reuse of code, and maintenance of large and evolving software systems. The same applies to object-oriented modeling languages. They are especially well-suited to model complex and multi-domain systems. A very promising attempt to introduce a coherent object-oriented modeling language based on the experience of previous languages is Modelica [Mod99].

In this paper, the experience gathered with the prototype of a Modelica extension for model-based diagnosis is presented. The paper starts with the introduction of the special application field of model-based diagnosis. Basic principles are illustrated by the example of their realization in the model-based analysis tool RODON. Its object-oriented modeling paradigm is explained and compared to modeling in Modelica. Although both modeling philosophies are very close to each other, there are some additional requirements of model-based diagnosis. These are explained and illustrated by examples.

2 The Model-Based Analysis Tool RODON

The functional analysis tool RODON is a model-based simulation, monitoring and diagnosis tool which integrates known engineering

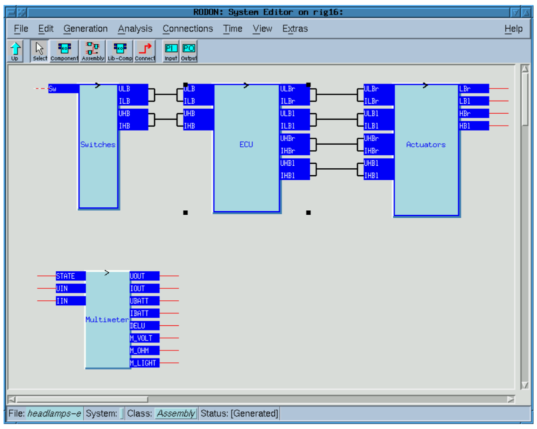
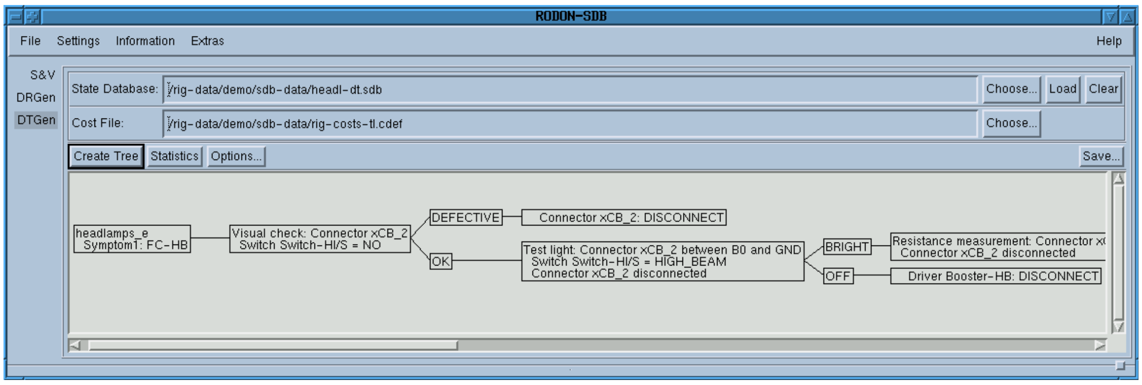


Figure 1: A RODON model of car headlamps.

[*]K.Lunde@rose.de



State Database: /rig-data/demo/sdb-data/headl-dt.sdb

Cost File: /rig-data/demo/sdb-data/rig-costs-tl.cdef

headlamps_e
Symptom1: FC-HB

Visual check: Connector xCB_2
Switch Switch-HI/S = NO

DEFECTIVE — Connector xCB_2: DISCONNECT

OK — Test light: Connector xCB_2 between B0 and GND
Switch Switch-HI/S = HIGH_BEAM
Connector xCB_2 disconnected

BRIGHT — Resistance measurement: Connector x
Connector xCB_2 disconnected

OFF — Driver Booster-HB: DISCONNECT

Figure 2: An automatically generated decision tree for model-supported diagnostics by the service.

methods with AI technology (see [Sei97]). It provides the following analyzes using the same knowledge base:

- requirements analysis (design layout)
- design verification
- risk analysis
  - fault tree analysis
  - failure mode and effect analysis
  - sneak circuit analysis
  - tolerance analysis
- process monitoring
- model-supported diagnosis
  - decision trees
  - diagnostic rules
- model-based diagnosis (on or off board)

2.1 Declarative modeling

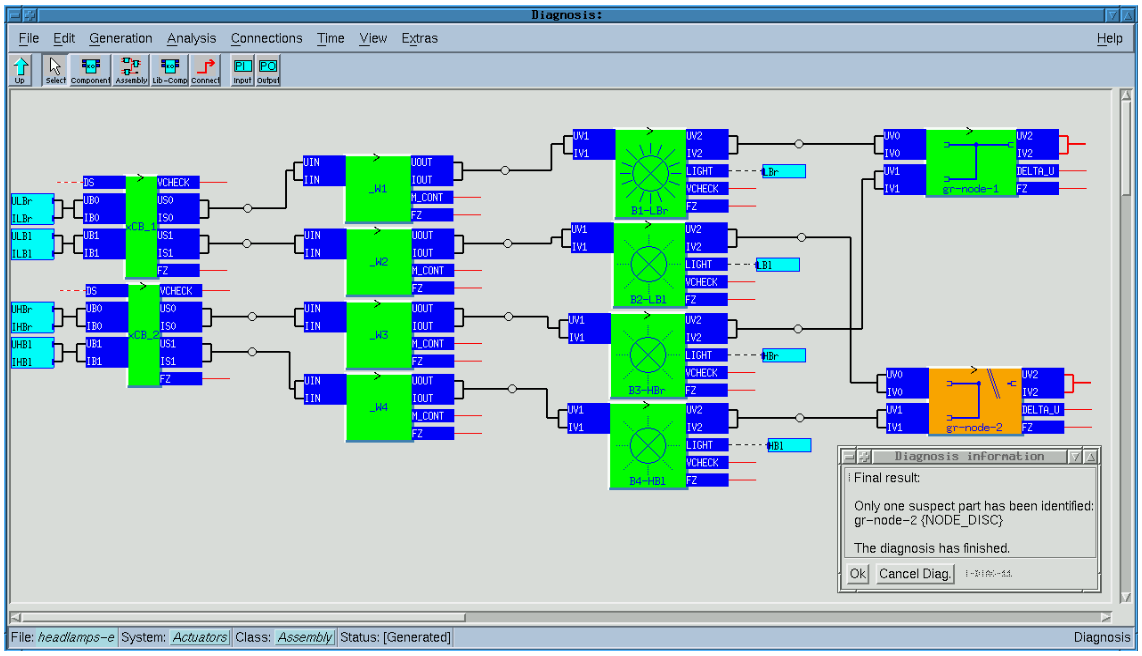
To build the necessary knowledge base, a technical system is mapped to a hierarchical model consisting of a number of functional units which are connected with each other. The behavior of each functional unit (component) is described in terms of physical laws, in the form of constraints. A constraint defines a relation between model variables. Thus, the component behavior is formulated declaratively, in contrast to many conventional simulation tools like Simulink or Matrix$_X$, where all system equations are essentially assignments, and all component ports have to be either inputs or outputs. The declarative (or non-causal) modeling concept allows to model component behavior independently of the context where the component shall be used. This leads to more simple and flexible model libraries, which are reusable in many contexts. At the same time, it remains possible to formulate certain constraints as assignments if necessary, for instance in subsystems where a behavior description by signal flow is an appropriate abstraction.

A declarative modeling approach reflects also the fact that in physical systems, a steady state is an equilibrium determined by the interaction between all components. For instance, modeling a complex electrical circuit by signal flow is nearly impossible. Moreover, if the behavior of one component in the circuit changes (e. g. in case of a defect), the model had to be re-designed because of the different signal flow. This shows that for model-based diagnosis, a declarative formulation of model behavior is essential.

Risk analyzes as well as diagnostics benefit from the definition of additional behavioral modes modeling the component behavior in typical failure states. How behavior modes can be used in model-based diagnosis will be described in Section 2.3.

2.2 Simulation in RODON

In real technical systems, some model parameters may be subject to manufacturing tolerances, others may be inaccurate because of measurement errors. For diagnostic purposes, it is especially important that the simulation results of a model match the behavior of the real system, because discrepancies are interpreted as defective behavior. Hence, it is dangerous to define those uncertain parameters by sharp real num-



Final result:

Only one suspect part has been identified:
gr-node-2 {NODE_DISC}

The diagnosis has finished.

Figure 3: A diagnosis: in case that gr-node-2 is disconnect, a leakage current causes the high beam lamps to shine dimmed, although they are switched off.

bers, more or less arbitrarily, as is usually done with conventional simulation tools.

To reflect these uncertainties, in RODON the basic value type is not the real number but the interval. More precisely, since interval operations may result in multiple intervals, the values of RODON variables are represented by interval sets, and all operations are carried out by interval arithmetics (see [HHKR95], [AH74]).

Since product specifications often contain tolerances or ranges, this value representation also suggests the use of RODON as a tool in requirements analysis or design verification.

The simulation algorithm of RODON seeks to constrict the values of all model variables as far as possible, without losing any solution. It bases on a local constraint propagation technique which was adapted for interval constraint propagation by E. Hyvönen (see [Hyv91]). The iterative algorithm starts by assigning undetermined values to all variables except for those whose values are given by the user (or in case of diagnosis: where input from the real system is available). An agenda mechanism is initialized with a list of all constraints. Now, the problem solver takes the constraints from the agenda one by one and evaluates them. If the value of some variable changes after evaluation, the problem solver collects all constraints which depend on this variable and puts them on the agenda again. They have to be evaluated anew, to propagate the new value through the system. This iterative process does not stop while the agenda contains constraints, that means, while some values keep changing (with respect to some accuracy, which is adjustable by the user). This algorithm is called local constraint propagation because changes are propagated through the system by passing them to the immediate neighbors in the constraint net.

Local constraint propagation has a number of advantages. For instance, it allows to solve constraint systems which are under- or overdetermined. This is especially important in model-based diagnosis, where a system has to be overdetermined in order to allow conclusions about defective components.

2.3 Model-based diagnosis

Suppose one is given a description of a system, together with an observation of the system's behavior which conflicts with the way the system is meant to behave. The diagnostic problem is to determine those components of the system which, when assumed to behave abnormally, will explain the discrepancy between the observed and correct system behavior [Rei87].

The observed system behavior is given by process data, i. e. by a set of values measured by the real system in operation. A diagnosis starts with a simulation of the nominal system behavior together with the given process data. The resulting equation system has to be overdetermined. If it has no solution, a conflict occurs during local propagation, which means that one or more components must be defect (i. e. behave abnormally).

In case of a conflict, RODON analyzes the conflict and generates hypotheses (candidates) which may explain the abnormal system behavior. A candidate is a minimal set of violated assumptions, where an assumption can have a form like ”the component XY behaves normally”.

For the efficient generation of hypotheses it is not sufficient to know that a simulation ended with a conflict. Without additional information about where the conflict occured candidate generation would be a search in a gigantic search space. Simulation by local constraint propagation provides us with this information. With each value set, RODON manages the assumptions the value set depends on. This is done by means of a truth-maintenance system (TMS). If a conflict occurs, RODON is able to backtrack the assumptions corresponding to the conflicting values. They can be used to narrow the search space to candidates containing these assumptions.

This diagnostic approach is based on the GDE (general diagnostic engine) introduced by de Kleer ([dKW87], [dKW89]). It is characterized by

- an incremental diagnostic procedure
- use of behavioral modes for candidate verification
- a truth-maintenance system (TMS) to reuse knowledge in multiple contexts
- a scalable candidate generator
- the ability to diagnose multiple faults as well as unspecified faults

3 Object-Oriented Modeling in RODON

The tool goes back to ideas of W. Seibold [Sei92] in the early 80’s, and has developed over the past 10 years. From the very beginning, an object-oriented modeling approach was used to define hierarchies of model component classes. It agrees with the Modelica philosophy [Mod99] in central points, namely

- hierarchical modeling
- component-oriented modeling
- declarative modeling
- quantitative modeling
- multi-domain modeling
- support by object-oriented model libraries

This remarkable similarity suggests the idea to support Modelica as a modeling language in RODON. To gain experience for the integration of Modelica into RODON, we implemented a prototype of a model representation module, and combined it with a new version of the RODON diagnostic engine.

While structure and topology representation of Modelica models could be adopted without any changes, the behavior representation had to be slightly modified.

In the following subsections, we will explain the alterations made for our application. All examples are written in the Modelica dialect we currently use with our prototype implementation. Its behavior section is fitted to our special needs. Note that the language specification is not fixed entirely.

3.1 Modeling behavior modes

Let us start with a simple example modeling an electrical wire class with two behavioral modes: the nominal behavior, and the most common failure modes disconnect, and shortToGround. An appropriate class hierarchy could be the following.

```
package Rose.Electrical
connector Port
```

```
type Current = Interval(unit = "A", quantity = "current");
type Resistance = Interval(unit = "Ohm", quantity = "resistance");
type Voltage = Interval(unit = "V", quantity = "voltage");
type FM = Discrete(min = 0, quantity = "failure mode");
```

Figure 4: Type definitions in RODON.

```
  Voltage u;
  flow Current i;
end Port;

partial model TwoPort
  Port p, n;
  FM fm (max = 1);
behavior
  if (fm==0 | fm==1)
    Kirchhoff(p.i, n.i);
  if (fm==1) p.i = 0;
end TwoPort;

model IdealWire2
  extends TwoPort (fm (max = 2));
  protected Interval iGnd;
behavior
  if (fm==0 | fm==2) p.u = n.u;
  if (fm==2) p.i + n.i + iGnd = 0;
  if (fm==2) p.u = 0;
end Wire2;
end Rose.Electrical;
```

The model `TwoPort` is a general electrical component providing basic physical laws for the current in case of nominal behavior (`fm==0`) and the failure mode disconnect (`fm==1`). The alternative behavior for different behavioral modes is defined using conditional constraints, where the discrete failure mode variable `fm` serves as a kind of switch.

The model `IdealWire2` extends this base class by adding a constraint for the voltage (it is an ideal wire without resistance) as well as constraints defining the behavioral mode shortToGround (`fm==2`).

Recall that all variables in a RODON model are represented by sets of values. This is reflected by the type definitions in Fig. 4. The basic type `Interval` has all attributes of the Modelica type `Real`, but its value attribute is a set of intervals. Accordingly, the value attribute of the basic type `Discrete` is a set of integer values.

An important semantic difference to the **`equation`** section in Modelica is that in our **`behavior`** section, any variable can participate in any number of constraints. Our iterative simulation algorithm does not rely on the fact that the equation system contains exactly the same number of state variables and equations. It can handle over- or underdetermined equation systems easily.

3.2 Modeling alternatives

3.2.1 `if`-clauses in RODON

The example above illustrated one way to model alternative behavior by means of a syntactical element known from Modelica — the **`if`**-clause. However, there are some semantic differences between **`if`**-clauses in RODON and Modelica. In RODON, the clause

```
if (<condition>) <relation>
```

means that when the problem solver takes this constraint from the agenda, it checks first the condition. Only if the condition is decidable and true, the relation is evaluated, and the consequent value changes are propagated as usual. This approach allows the use of many **`if`**-clauses in a model without slowing down the simulation process significantly.

For set-valued variables, logical expressions have not merely two possible values, but three: true, false, or undecidable. A condition is decidable if it is either true or false for all combinations of values out of the variable's value sets. For instance, consider the constraint

```
if (2<x & x<5) <relation>
```

If `x` has the value set `[0 4]`, there are some elements of the set for which the condition is true, but there are other elements in the value set of `x` for which it is false. During the iterative propagation process, the value set of `x` may be further constricted so that at some moment in the propagation process, the condition will have a unique logical value. But at the present moment, the problem solver cannot know which logical value that will be.

Note that for set-valued variables, there are some subtleties when defining correct semantics for conditions. Conditions have to be evaluable monotonly, i. e. if a condition has been decided to be true or false once, it must not change this logical value during propagation. This is due to the fact that if a condition has once been true, the corresponding relation has been evaluated, and the resulting value changes have been propagated further. If later on the condition becomes false, we would have to withdraw all values which depend on the earlier evaluation of that relation. To keep track of all these dependencies requires a huge effort.

Therefore, only monotonously evaluable conditions are allowed. That means that a condition like `(x subset y)`, where `x` and `y` are interval variables, is not permissible, because it is true for `x=[0 10]` and `y=[-10 20]`, but during the iterative propagation process `y` may be constricted to `y=[-10 2]`, which renders the condition undecidable. An overview of all permissible logical relations for set-valued variables is shown in Fig. 5.

```
Interval x, y;
Real a;
constant Interval s;
```

x==a	x **subset** s	
x!=a	x!=s	x!=y
x<=a	x<=s	x<=y
x< a	x< s	x< y
x>=a	x>=s	x>=y
x> a	x> s	x> y

Figure 5: Permissible logical relations for value sets.

Another topic is the negation of logical expressions, which have a slightly different meaning for sets than for real values. For instance, the negation of the condition `(x==a)` is not `(x!=a)` but `(a∉x)`. In general, the relation `x!=y` has to be interpreted as `x∩y==∅`.

3.2.2 The `or`-clause

There are situations where it is desirable to have another kind of alternative behavior. Consider the following definition of a piecewise linear function:

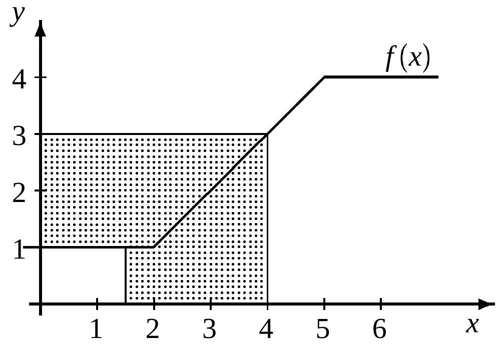


Figure 6: A piecewise continuous function.

$$f(x) = \begin{cases} x < 2 & 1 \\ 2 \leq x \leq 6 & x-1 \\ x > 5 & 5 \end{cases} \qquad (1)$$

For real-valued variables, this function can be modeled adequately by means of `if`-clauses:

```
if (x<2) y = 1;
if (x>=2 & x<=6) y = x-1;
if (x>6) y = 5;
```

But for interval-valued variables, the conditions of these **`if`**-clauses may be undecidable, and the variable `y` will not be constricted at all by these constraints, although it is clear that the range of `y` is `y=[1 5]`. For instance, if `x=[1.5 4]`, the evaluation of a constraint representing the equation $y = f(x)$ should result in `y=[1 3]` (see Fig. 6). In this case, not evaluating the corresponding relations due to their undecidable conditions `x<2` and `(x>=2 & x<=6)` means waste of information, which may be crucial in the propagation process.

For such situations the modeling language of RODON has a disjunction element. By means of a disjunction, (1) would be modeled like

```
or {{x < 2; y = 1;}
    {x = [2 5]; y = x-1;}
    {x > 5; y = 4;}}
```

The equations in the alternative cases of the disjunction are evaluated separately, and the result is the set union of the alternative cases. For set-valued variables, such a language element is essential.

Another typical example where a disjunction is useful is the calculation of a discrete indicator

```
constraint MyQuadraticSpline
  Interval x, y;
  extends Spline (final degree = 2, periodic = true,
                  final values = {{0.5, 1}, {1, 1.5}, {1.5, 3}});
end MyQuadraticSpline;

model UseSpline
  Interval a, b;
behavior
  MyQuadraticSpline(a, b)(periodic = false, boundary = {1.3});
end UseSpline;
```

Figure 7: An example of a simple constraint class.

value based on a continuous variable. Consider the following model of an electrical light bulb.

```
model Bulb extends TwoPort;
  Discrete light (max = 2);
  parameter Current pNom=20;
  parameter Voltage uNom=12.0;
  protected
    Resistance r;
    Power pc;
    constant Power pLow=[-0.1 0.1];
    constant Power pMedium
      ={[-0.3 -0.1][0.1 0.3]};
    constant Power pHigh
      ={[-2 -0.3][0.3 2]};
behavior
  r = uNom * uNom / pNom;
  pc = p.i * (p.u - n.u) / pNom;
  if (fm==0) Ohm(p.u, n.u, p.i, r);
  if (fm==1) light = 0;
  if (fm==0)
    or {{p.i= 0; light = 0;}
        {pc = pLow; light = 0;}
        {pc = pMedium; light = 1;}
        {pc = pHigh; light = 2;}}
end Bulb;
```

The variable `light` is an indicator for the mechanic whether the bulb is off (`light=0`), is dimmed (`light=1`) or bright (`light=2`). If by some reason the constraint net is underdetermined so that the consumed power of the bulb `pc=[0.2 1]`, the mechanic gets the information `light={0,1}`, i. e. the bulb is off or dimmed, which may be an important indication. An analogous definition by means of conditional constraints would result in `light={0,1,2}`, i. e. the whole range of this variable.

3.3 Defining constraint classes

Very much like function classes in Modelica, RODON allows the user to define constraint classes. This may be useful for constraints which are used very often, like Ohm's law or Kirchhoff's law in the electrical domain. But it is also convenient for constraints which are laborious to define, e. g. characteristic curves of engines which are given in table form rather than in closed form. As for the function classes in Modelica, constraint classes define an argument list and provide type checking when the constraint class is used.

In the example shown in Fig. 7, the constraint class `Spline` is a basic constraint class provided by the RODON modeling language. It has pre-defined attributes `degree, periodic, boundary` and `values`. There are some other basic constraint classes. Constraint classes may be parameterized by means of the usual modification mechanism.

4 Conclusion

The striking correspondence of the modeling philosophies in Modelica and RODON as well as the neat class model of Modelica are strong arguments in favour of Modelica as the future modeling language of RODON. But there are some additional requirements caused by our different simulation algorithm and the diagnostic approach, namely

- basic data type: interval set
- multiple behavioral modes
- alternative behavior: disjunctions

- alternative behavior: conditional constraints, semantics of conditions for set-valued variables
- constraint classes
- coping with over- or underdetermined constraint nets
- simulation by local constraint propagation

A few of them can possibly be met by providing special libraries. However, our experience shows that there are features which require an extension of the Modelica language, or the definition of a separate dialect for behavior description.

References

[AH74] G. Alefeld and J. Herzberger. *Einführung in die Intervallarithmetik*. Bibliographisches Institut AG, Zürich, 1974.

[dKW87] J. de Kleer and B. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:1297–1307, 1987.

[dKW89] J. de Kleer and B. Williams. Diagnosis with behavioral modes. In *Proceedings of the IJCAI'89*, pages 1324–1330, 1989.

[HHKR95] R. Hammer, M. Hocks, U. Kulisch, and D. Ratz. *C++ Toolbox for Verified Computing*. Springer, 1995.

[Hyv91] E. Hyvönen. *Constraint Reasoning with Incomplete Knowledge*. PhD thesis, Helsinki University, 1991.

[Mod99] Modelica Association. *Modelica - Language Specification*, December 1999.

[Rei87] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57 – 95, 1987.

[Sei92] W. Seibold. Grundlage der Diagnose technischer Systeme — sechs Thesen. *ist – Intelligente Software-Technologien*, 2(3), 1992.

[Sei97] W. Seibold. First time right from layout to repair. In *Proceedings of the 30th ISATA Symposium, Florence (Italy)*, pages 483–492, June 1997.