# AD2L - A Programming Language for Model-Based Systems (Preliminary Report)

**Thomas Havelka** and **Markus Stumptner*** and **Franz Wotawa**†
Technische Universität Wien,
Institut für Informationssysteme and
Ludwig Wittgenstein Laboratory for Information Systems,
Favoritenstraße 9–11, A-1040 Wien,
Email: {havelka,mst,wotawa}@dbai.tuwien.ac.at.

## Abstract

Model-based reasoning in general and model-based diagnosis (MBD) in particular has become a very important technique for providing solutions to real-world problems. However, modeling has become a problem making the use of MBD in companies difficult or even preventing it from being considered. Reasons are the lack of modeling principles and a general accepted language for modeling. In this paper we propose a programming language AD2L specially tailored for describing systems to be diagnosed. In addition, we discuss guidelines a modeling language should or even must fulfill, give some properties of the language, and an overview about further extensions increasing the applicability of AD2L.

## Introduction

Model-based reasoning in general and model-based diagnosis (MBD) (Reiter 1987; de Kleer & Williams 1987) in particular have become very important techniques for providing solutions to real-world problems. Some applications are described in (Wotawa 1999) including systems for the automotive domain (Cascio *et al.* 1999), electrical circuits (Snooke 1999; Flores & Farley 1999), and phone networks (Steimann, Fröhlich, & Nejdl 1999). Other domains are space exploration (Williams & Nayak 1997) and software debugging (Friedrich, Stumptner, & Wotawa 1999; Stumptner & Wotawa 1999) where model-based reasoning has been applied successfully. All applications have in common that they require a model to allow the derivation of diagnoses. This model is in most cases a mapping from the physical world to a logical description except in the case of software debugging. In the debugging domain a model is a description of the structure and semantics of the program to be debugged. It is central for MBD to have a model describing the structure and behavior of a system (Davis 1984; 1993).

Since the beginnings of model-based reasoning several techniques for representing models have been proposed. They mainly have in common that they are qualitative in nature, i.e., they do not use quantitative values. Such models are not only used in MBD but also in other fields. For example hardware designers speak about "low" and "high" or "true" and "false" instead of the exact voltage levels. In (Dague 1995) an overview of qualitative modeling is given. Although, the basic modeling principles seem to be established, there is almost no accepted and widely used model description language available. Every reasoning system based on specific models use its own languages. In addition, apart from (Iwasaki *et al.* 1997) almost no work in the direction of providing tools for handling models and model libraries has been done. In (Iwasaki *et al.* 1997) a WWW-based modeling system for sharing knowledge about physical systems is described. They use CML (Compositional Modeling Language) for describing models that can be translated to the Knowledge Interchange Format (KIF) (Genesereth & Fikes 1992). CML combines languages used for describing systems using Qualitative Process Theory (Forbus 1984) and the Qualitative Physics Compiler (Crawford, Farquhar, & Kuipers 1990). Other approaches for sharing diagnosis knowledge include (Heck, Laengle, & Woern 1998) where KQML (Finin, Labrou, & Mayfield 1997) is used as communication language. Recent approaches for model interchange are mostly based on XML. We did not take this approach, because we consider XML to be primarily a language for information exchange, which does provide support for defining semantics specific to modeling for diagnosis. On the other hand, it is straightforward to convert AD2L to an XML-based format.

All previous approaches that rely on a logical description of the model are well suited for presentation purposes. However, they are not so good, when modeling is to be done by not so experienced users. We face this problem in industry, where people are not familiar with the concepts of MBD and logic description languages (including Prolog). Although they see advantages of MBD compared with other approaches, they are sceptical concerning the realization of the advantages, e.g., reuse of models. While teaching students (especially from the electrical and mechanical engineering fields) the fundamentals of model-based diagnosis might alleviate the problem in the long term. A major step forward on the road to more general acceptance could be to uncouple the representation issue from the theoretical roots of the field and provide a dedicated representation that is more in line with the background of practitioners who might be "put off" by the appearance of pure logic. Advantages of

---

* Currently at Universität Linz, Data and Knowledge Engineering Group, Altenbergerstraße. 69, A-4040 Linz.

† Authors are listed in alphabetical order.

a widely accepted language would include the possibility to interchange models between researchers and companies, or between companies directly, the increase of reuse, and the certainty for companies that the model description can be used for a long time, thus saving the investments for modeling and providing an argument for using MBD.

The language AD2L described in this paper is a proposal for such a modeling language. AD2L (Piccardi 1998) has been developed as part of a project with the goal of interchanging system descriptions over the Internet, and has been extended and adapted for industrial needs afterwards. The language definition is independent of the underlying diagnosis engine and provides language constructs directly representing model-based concepts, e.g., components and connections. Additional other concepts from programming language design have been incorporated such as packages and strong typing. This allows for building model-libraries and avoids errors at runtime, that are central requirements of industry.

Similar approaches have been considered in the past, such as the language COMODEL (Dilger & Kippe 1985), but have not found general use in industrial applications. AD2L on the other hand was developed in collaboration with industry.

This paper is organized as follows. In the next section we discuss certain guidelines a general modeling language should fulfill. After that we present AD2L and argue why we think it is appropriate as such a general language. We also give a simple example to show how models in that language will look like. In the fourth section we take a look at qualitative models, which have been the focus of much interest during the years and show that qualitative models can be easily described using AD2L. In the last section we show some shortcomings that the language in its current version does still have and give an outlook on how we want to improve it.

## Guidelines for Modeling Languages

The task of modeling has mainly been guided by the problem at hand and by the tools available, i.e. the underlying diagnosis engine. We argue that more thought should be given to the modeling process itself and propose that general guidelines for modeling must be developed. In the following we give a list of guidelines we think should be included in such a general standard. These ideas may not be complete, but they should suffice as a starting point for discussion.

Reusability

As systems grow more and more complex so does the task of modeling them. Therefore it is reasonable to reuse parts of models, once they have been developed for other systems, whenever they are applicable.

This is only possible on a large scale when there is a standard for describing these systems. If the descriptions are formulated in different paradigms, exchange is impossible. Component libraries are useless when the diagnosis engine is changed and the models are no longer applicable.

Understandability

A goal of a standardized modeling language should be easy usage, not only by the implementor of the language itself or the diagnostic/configuration system, but by the domain experts themselves. It most cases they will have a *basic* understanding of programming languages and of logic, but should not be forced to learn a complex specification language.

Independence from the underlying Theorem Prover

Most languages have been defined for a certain theorem prover. This is mostly due to the fact that the underlying diagnosis engines use a different kind of reasoning. A general description language must be easily translated to any of these systems.

Well defined Semantics

The semantics of the language must be well defined and easy to understand. The knowledge engineer should be able to formulate his ideas in an easy way. On the other hand it should be easy to deduce from a system description how the system actually behaves.

Different Models

It should be possible to describe components at different levels of abstraction. An adder could be described as a component that has numerical values as its in- and outputs, as well as arrays of boolean values or even arrays of numbers representing electrical currents. Depending on the task at hand, the diagnosis engine can use the level of description that will help the most to solve the problem.

Diagnosis Knowledge

A modeling language must provide language constructs for representing knowledge about the systems to be diagnosed. Apart from components and connections a model designer has to specify repair actions and costs, costs for observations, knowledge about observability, tolerance criteria (intervals), physical impossibilities (Friedrich, Gottlob, & Nejdl 1990), and others. In addition, the language must allow for describing hierarchical models.

Meta-Knowledge

In addition to the description of the system itself it should be possible to describe knowledge about the diagnosis task. This includes information about which diagnosis algorithm to use, the level of description of components to be invoked, etc.

Usability for Configuration

We have recently been involved in work dealing with the interface between configuration and diagnosis (Stumptner & Wotawa 1998). Configuration is an active research area, with a certain overlap to diagnosis in its subject. In particular, configuration systems, like diagnosis, use component-connection models for representation. Currently however, configuring a new system and diagnosing the result are generally treated as two separate tasks, each of which requires a separate description of the components used. In times when product cycles tend to become shorter and shorter this is a fact most unwelcome to the designers and maintainers of technical systems. A language applicable both to configuring and diagnosing systems could reduce much of this overhead. (Most of these comments would apply to design as well as to configuration.)

Important for industrial purposes is that a modeling language should be easy readable and learnable, even for non-experts in this field. Unlike conventional programs, erroneous models will not result in crashes of the diagnosis system, but still errors in the system description can be hard to find. Therefore we provide a typed language (i.e., all components and all connections are typed) and allow for statically checking type correctness. The language AD2L currently does not fulfill all requirements but the most important to be applicable in an industrial context. For example meta-knowledge, e.g., relations between models, cannot be described. However, repair costs for components can be stated allowing to trade of between adding observations and doing repair (Friedrich & Nejdl 1992).

Not all of these requirements are equally important. Having a language that is independent of the diagnosis engine and allowing to describe models declaratively is important in the case where the model should be used by different companies or where knowledge transfer is a requirement, e.g., between diagnosis agents. A declarative model has the advantage that compiling the model into different representations is easy. In several cases it is better to use fault trees, i.e., decision trees, for diagnosis. One case is for example when a solution has to be computed very quickly or whenever memory or processor requirements are not met. This is the case for on-board diagnosis in the automotive domain (Cascio *et al.* 1999). Not so important but very helpful in certain domains is the use of generic parameters. For example, using this feature, only a model for a $n$-input AND gate would have to be written, and the generic parameter $n$ could be set to create components with arbitrary number of inputs based on the generic definition.

## The AD2L Language

Based on the ideas described above we have developed the language AD2L for describing systems. In the following we describe the syntax and semantics of AD2L. The source code of a digital full adder implemented in AD2L can be found in the appendix. We will use this example for describing the main parts of the programming language. Note that due to space restrictions it is not possible to give a complete description of all the details of the language.

In AD2L a system to be diagnosed is described by its components and their connections. There are special language constructs for describing components, the connectivity, and their behavior. The system as a whole is considered as the top-level component. There are no special statements for distinguishing systems from hierarchical components. There are two kinds of components: atomic and hierarchical components. Atomic components are specified by their behavior, hierarchical components by their internal structure: other components and the connections between them. Each kind of component has a set of ports, to connect the component to its environment. For each port the direction of value-passing and the type of values passed is defined.

```
<AtomicComponent> →
  <ComponentHeader>
  [ <VariableDefinitions> ]
  [ <AxiomDefinitions> ]
  <BehaviorDefinitions>
  end component

<HierarchicComponent> →
  <ComponentHeader>
  <SubcomponentDefinitions>
  <ConnectionDefinitions>
  end component

<ComponentHeader> →
  component <ComponentIdentifier>
  [ <AD2LComment> ]
  <GenericDefinitions>
  [ <ConstantDefinitions> ]
  [ <PortDefinitions> ]
```

### Description of behavior

For each component several modes of behavior can be specified. Each of these modes is defined by a set of rules specifying a relation between values at different ports of the component. Each mode is associated with a probability, a comment specifying what happens in this mode as well as costs and an action, to specify repair-actions and costs to be taken, if the component is stuck in this mode.

The concept of behavior modes is an extension of both (de Kleer & Williams 1989) and (Struss & Dressler 1989). In that an *unknown fault mode* can, but must not, be included into the system description. It should be noted that the absence of an unknown fault mode - which allows any behavior - means that the behavior modes must be complete in the sense of describing all possible behaviors, i.e. every combination of port values. In AD2L rules are used for declaring the behavior of a component mode.

```
<Rule> →
  [ <ImplicationSentence> ] =:
     <RightHandConjunctiveSentence> .
  | <ImplicationSentence> =: False .
  | <ImplicationSentence> =:=
     <RightHandConjunctiveSentence> .

<ImplicationSentence> →
  <ConjunctiveSentence> -: <ImplicationSentence>

<ConjunctiveSentence> →
  <Sentence> , <Sentence>

<Sentence> →
  ( <ImplicationSentence> )
  | <PredicateSentence>
  | <QuantifiedSentence>

<PredicateSentence> →
  <Predicate>

<QuantifiedSentence> →
  [ forall | exists ] <VariableList> : <ImplicationSentence>
```

<RightHandConjunctiveSentence> →
<RightHandSentence> , <RightHandSentence>

<RightHandSentence> →
( <ImplicationSentence> )
| <PredicateSentence>
| <RightHandQuantifiedSentence>

<RightHandQuantifiedSentence> →
[ **forall** ] <VariableList> : <RightHandConjunctiveSentence>

In the full adder example given in the appendix, the and-gate has three behavior modes (one normal behavior mode and two fault modes). One fault mode assumes that the output is stuck at 0 (or false), and the other assumes that the output is stuck at 1 (or true).

One can distinguish two kinds of rules. The basic rule , denoted by the operator `=:` specifies a condition part on the left side and an action part on the right side. Whenever the conditions on the left side hold, the actions on the right side are executed, e.g. the specified values are propagated to the ports.

In the other, bidirectional type of rule, denoted by `=:=`, both sides define actions as well as conditions. The rule `leftside =:= rightside` can be considered as abbreviation for the rules `leftside =: rightside` and `rightside =: leftside`.

The meaning of the rule

Val(i1,true), Val(i2,true) =:= Val(o,true).

is that whenever both inputs are true, true will be propagated to the output. Whenever the output of the gate is true, this value will be propagated to both of the input ports.

On the other hand, the rules

Val(i1,false) =: Val(o,false).
Val(i2,false) =: Val(o,false).

specify propagation into one direction only.

## Predicates

The basic elements of rules are predefined predicates which describe possible facts about the system and can be true or false. The set of these predicates is fixed:

**Val** is a predicate with two parameters. The first one specifies the port and the second one the value associated with that port.

**Cond** is a predicate containing one argument which is a boolean expression. The predicate itself is true whenever the expression evaluates to true. *Remark:* This predicate was introduced to provide a shorthand in a frequently occurring situation (syntactical sugar).

**Conn** is a predicate with two parameters. The first one specifies a port of the component and the second one the component the port is connected to. If the port is connected to more than one component, the predicate is true for all of these components.

**Unconn** is a predicate with one argument, specifying a port. The predicate is true whenever there is no component connected to the port.

**Type** is a predicate with two arguments. The first one specifies a component and the second one a component type. The argument is true whenever the type of the component and the specified type match.

To illustrate the usage of the later predicates one example of using these predicates will be given:

**exists** INPORTS: Conn(INPORTS,X),
TYPE(X, old_model) =: Val(warning,true).
**forall** INPORTS: Unconn(INPORTS,X) =: Val(free,true).

This particular group of rules describes a component that can be connected to several other components. The first rule sets a `warning` port to true whenever an `old_version` component is used. The second rule sets a `free` port to true announcing that no other component is connected to the component itself.

## Value Predicates

The other kind of predicates are defined by the user as elements of enumeration types. These are passed as values between the ports of the components and must not appear in the rules directly. They are just allowed within the predefined `Val`-predicate. By means of these predicates more complex data can be passed amongst the components. Using this kind of predicate it is possible to do temporal reasoning via rules like this:

Val(in, tmp_val(X, T)) =:= Val(out, tmp_val(X, T+1)).

where a connection having as value the predicate `tmp_val(X, T)` means that at time `T` the value is `X`. The underlying diagnosis or constraint system is responsible for dealing with the proper handling of temporal facts.

## Predefined Constants/Variables

There are some predefined variables (to be used by quantifiers) and constants referring to the component itself in AD2L. The predefined variables are:

**COMPS** is the set of all components that are defined in the surrounding hierarchical component. In our example above if `COMPS` is used by an and-gate in one of the half-adders it would refer to the and-gate and1 and the XOR-gate xor1, if used in the definition of an or-gate in the or-gate or1 of the full adder it would refer to the half adder halfadder1 and halfadder2 and the or-gate itself.

**PORTS** is the set of all ports of a component. This could easily be defined by the user himself but makes the usage of AD2L easier.

**INPORTS, OUTPORTS, INOUTPORTS** is the set of all ports being specified as having the respective mode.

The following constants are available:

**IDENTIFIER** is the name of a component as specified by the surrounding composite component. If there is no such surrounding component, the value of IDENTIFIER is undefined.

**TYPE** is the type of a component.

### Axioms

Axioms are a set of rules that always hold a given component, i.e. independent of the behavior mode. By specifying axioms for a component, the concept of physical impossibility as proposed in (Friedrich, Gottlob, & Nejdl 1990) can be expressed. The syntax and semantics of these rules is the same as for the rules that describe the various behavior modes. As an example we describe the classical light-bulb that cannot be lit whenever there is no power supply. This is done by the following rules (the definition of ports and data types should be obvious from the context):

Val(light, "on") =: Val(power, "+").
Val(power, "0") =: Val(light, "off").

The inclusion of physical impossibility to the description of component behavior is intended to prevent diagnoses that can be derived from the system description itself, but are not possible in the real world. They give a certain advantage for diagnosing systems where the complete set of failure modes is not known or hard to describe.

### Forced Discrepancies

Sometimes there are constraints on the behavior of a component that are not independent of the behavior mode. A violation of one of these constraints indicates that the component cannot be in that mode. Take for example a pipe that, depending on the positioning of a switch, can transport either 0 or 5 liters of water (in a given period of time). When the amount of water passing through that pipe exceeds 5 liters, this is an indication that the switch is broken. This fact can be made explicit by the predicate without arguments `Fail`. Whenever this predicate can be concluded by the underlying theorem prover, a contradiction has been derived.

**component** pipe_with_switch

  **input** switch : **bool**.
  **output** out : **integer**.

  **behavior** normal
    Val(switch, true) =: Val(out,5)
    Val(switch, false) =: Val(out,0)
    Val(out,X), Cond(X $>$ 5) =: Fail.
  **end behavior**

**end component**

### Quantifiers

The rules may contain quantified expressions. These quantifiers are similar to those of first order logic, although the notation these quantifiers are used is somewhat different. The AD2L sentence

**forall** INPORTS : Val(INPORTS, true)

would (assuming that inports is given for a particular component) be written in first-order logic as:

$\forall x \in inports : val(x, true)$

The quantified sentence must contain the name of a (predefined or user-defined) variable which is associated with a set of elements (ports, components or values). The AD2L sentence that is bound by the quantifier is instantiated with each of the elements of the set and these instances are connected depending on the quantifier.

Thus the AD2L-sentence

**forall** INPORTS : Val(INPORTS, true) =: Val(Out, true).

is in fact an abbreviation for the sentence

Val(in1, true), Val(in2, true), Val(in3, true) =: Val(out, true).

if the component whose behavior is described above does have the input-ports `in1`, `in2` and `in3`.

On the other hand, the sentence

**exists** INPORTS : Val(INPORTS, true) =: Val(Out, true).

would correspond to the set of sentences

Val(in1, true), =: Val(out, true).
Val(in2, true), =: Val(out, true).
Val(in3, true), =: Val(out, true).

In addition to providing the capability for more concise behavior descriptions, quantifiers are necessary for defining the behavior of components whose the number of ports is defined by a generic value (see below).

### Description of structure

AD2L supports structured definition of components. Each component can be specified by a set of interconnected components itself. The structure is specified by a set of connections that link a number of ports of the subcomponents.

### Connections

Connections are identified by a name. Each connection links one output or inout port with several other ports. Connections are important during diagnosis because they are associated with a certain value. Determining these values gives additional information about the state of the system. These measurements are associated with a certain cost, sometimes it is not possible to determine the values at all. This information is specified together with the definition of the port.

### Observations

Sometimes a set of Observations (taken at different times, with different input values) can help differentiating between diagnoses. To facilitate switching between or combining sets of observations, observations in AD2L can be named.

### Data Types

In AD2L there are five pre-implemented data-types: integer, real, boolean, string, and character. In addition to these, the user can generate new data-types on his own. Basically there are two ways of generating new data-types:

1. Enumeration Types are (ordered) sets of numbers, strings, characters and user-defined predicates (see below).

2. By changing the equivalence relation for an existing data type. These ”tolerances” can be used for number types as well as enumeration types. For numbers, intervals are defined, with values considered equal if they fall within each other’s interval. These intervals can be specified by fixed numbers (e.g. [+/- 10]) as well as a percentage of the numbers themselves. For enumeration types equality is defined explicitly, by specifying which of the elements are considered as equal.

### Indices and Generic Values

An additional feature of AD2L is the use of arrayed ports and components.

```
input in[1-8] : bool.
```

specifies an array of boolean input ports, which can also be linked to an array of components:

```
subcomp[1-8] : subcomptype.
. . .
inline[1-8] cost 5 : in[1-8] -> subcomp[1-8](in).
```

The statement above defines 8 connections named `inline[1]` to `inline[8]` that connect the corresponding inports `in[n]` to the port `in` of the corresponding component `subcomp[n]`.

Together with the concept of generic values, this construct provides a powerful means of describing complex systems. Generic values are variables used in the description of a component whose value is determined when the component is instantiated. That way a component with a variable number of ports can be described. Generic values can be used to create a variable number of subcomponents inside a component. Take for instance the following description of an and gate.

```
component andgate
  generic insize : integer = 2.
  input in[insize]: bool.
  output o: bool.

  default behavior nab
    comment ”The normal behavior”
    action ”Do nothing, for this is the correct behavior”

    forall Val(INPUTS,true) =:= Val(o,true).
    exists Val(INPUTS,false) =: Val(o,false).
  end behavior

end component
```

This gate has a variable number of input ports. The actual number is determined when the component is used.

```
subcomponent and1 : andgate (insize = 5).
```

If nothing is specified the default value (2) is used.

## Qualitative Reasoning

In many domains, full quantitative models of a system are too complicated to define and simulate. Therefore, most of the time qualitative models are used instead. In the following we show a simple model for qualitative reasoning implemented with AD2L. The example is based on the models of a control circuit with motor as used by Malik and Struss (Malik & Struss 1996).

That example deals with a direct-current motor controlled in a feedback loop. The speed of the axis of the motor $\omega$ is measured by a revolution counter. That measurement $\omega_m$ is fed into the controller, which adjusts the voltage $v$ for driving the motor to arrive at the desired speed $d$. The controller itself has an internal constant $c_C$.

The (normal) behavior of the controller is determined by the following equation

$c_C \times v = 2 * d - \omega_m$.

The corresponding qualitative equation is

$[c_C] \otimes [v] \oplus [\omega_m] = [d]$.

In AD2L the controller would be described by the following program

```
type qual : {”+”, ”0”, ”-”}.

component controller
  input : omega, d : qual.
  output v : qual.
  inout cc : qual.

  behavior nab
    – The blocks of the behavior each describe the propagation
    – of one qualitative variable: d, omega, v and c

    Val(c,”0”), Val(omega,X) =: Val(d,X).
    Val(v,”0”), Val(omega,X) =: Val(d,X).
    Val(c,CV), Val(v,CV), Cond(CV != ”0”), Val(omega, Y),
      Cond(Y != ”-”) =: Val(d,”+”).
    Val(c,C), Val(v,V), Cond(C != V),
      Cond(C != ”0”), Cond(V != ”0”),
      Val(omega, Z), Cond(Y != ”-”) =: Val(d,”-”).

    Val(c,”0”), Val(d,”+”) =: Val(omega,”-”).
    Val(c,”0”), Val(d,”0”) =: Val(omega,”0”).
    Val(c,”0”), Val(d,”-”) =: Val(omega,”+”).
    Val(v,”0”), Val(d,”+”) =: Val(omega,”-”).
    Val(v,”0”), Val(d,”0”) =: Val(omega,”0”).
    Val(v,”0”), Val(d,”-”) =: Val(omega,”+”).
    Val(c,CV), Val(v,CV), Cond(CV != ”0”), Val(d,D),
      Cond(D != ”+”) =: Val(omega,D).
    Val(c,C), Val(v,V), Cond(C != V),
      Cond(C != ”0”), Cond(V != ”0”),
      Val(d,D), Cond(D != ”-”) =: Val(omega,D).

    Val(d,”+”), Val(omega,O), Cond(O != ”+”),
      Val(c,C), Cond(C != ”0”) =: Val(v,C).
    Val(d,”0”), Val(omega,”+”), Val(c,”+”) =: Val(v,”-”).
    Val(d,”0”), Val(omega,”+”), Val(c,”-”) =: Val(v,”+”).
    Val(d,”0”), Val(omega,”0”), Val(c,C), Cond(C != ”0”)
      =: Val(v, ”0”).
    Val(d,”0”), Val(omega,”-”), Val(c,C), Cond(C != ”0”)
      =: Val(v,C).
    Val(d,”-”), Val(omega,O), Cond(O != ”-”), Val(c,”-”)
      =: Val(v,”+”).
    Val(d,), Val(omega,), Val(c,”+”) =: Val(v,”-”).

    Val(d,”+”), Val(omega,O), Cond(O != ”+”),
      Val(v,V), Cond(V != ”0”) =: Val(c,V).
    Val(d,”0”), Val(omega,”+”), Val(v,”+”) =: Val(c,”-”).
    Val(d,”0”), Val(omega,”+”), Val(v,”-”) =: Val(c,”+”).
    Val(d,”0”), Val(omega,”0”), Val(v,V), Cond(V != ”0”)
```

```
      =: Val(v, "0").
    Val(d,"0"), Val(omega,"-"), Val(v,V), Cond(V != "0")
      =: Val(c,V).
    Val(d,"-"), Val(omega,O), Cond(O != "-"), Val(v,"-")
      =: Val(c,"+").
    Val(d,), Val(omega,), Val(v,"+") =: Val(c,"-").
  end behavior
end component
```

As can be seen qualitative models are possible in AD2L, but sometimes look a little bit complicated. This is due to the fact that AD2L represents that knowledge explicitly. No hidden reasoning mechanism or knowledge is needed. If knowledge is represented by formulas or tables, additional mechanisms are needed to reason about the value of variables if knowledge is incomplete.

## Further Work

There are still open problems and of course avenues for improvement concerning AD2L.

Physical Units: AD2L does not support physical units at the time. Purely for expressiveness reasons, there is no necessity for it. There is no reasoning about physical units involved, and the results themselves should be clear from the model itself.

Relations between elements of enumeration types: Enumeration types are specified by enumerating their elements. This enumeration gives a natural order for the elements. Equality between these elements, on the other hand, can be specified explicitly.

Functions: There is only a basic set of functions/operators implemented in AD2L. More complex functions cannot be implemented as functions themselves, but can easily be implemented as components. What is missing from AD2L at the moment is a set of mathematical *standard functions* such as square root, sinus, cosinus, etc.

Dynamic System Changes: Some systems are not fixed over time, but are subject to continuous change as new components are added to the system and others are removed (e.g., a computer network at a department, or a structural fault whose occurrence alters the behavior ofthe system). AD2L does not have any means of dynamically altering such a system. To do this either the language itself must be extended, or an implementation framework around AD2L must be defined that allows specifying such changes and generates the current system specification in AD2L.

Temporal Reasoning: Temporal models are currently not supported in AD2L, since most of the models encountered by us the industrial domains do not require temporal reasoning. However, our intent is to extend behavior descriptions in AD2L so as to handle temporal models as well.

### Discussion

AD2L has been developed as a system description language for model-based diagnosis. Although it is not as expressive as full first order logic, AD2L provides sufficient expressive power for formulating models used in a variety of industrial domains or for educational purposes. In addition, the various syntactical and semantical restrictions of AD2L guarantee that it can be used effectively. Time complexity is reduced to a minimum and the program size for typical example knowledge bases is quite small. Models written in AD2L can be used by different diagnosis engines and the behavior is described declaratively. Currently, AD2L is not yet a fully generic language, but it is not domain specific.

Another property of AD2L is that a subset of AD2L can be transformed into a propositional horn clause sentence, thus allowing use of fast consistency check algorithms (Minoux 1988). Such algorithms provide a consistency check in linear time depending on the number of literals in the propositional sentence, providing high performance to system descriptions of this form. This AD2L subset is specified by all programs using finite domains, e.g., not using **integer** or **real**.

We have previously commented on the fact that AD2L directly supports the modeling task by providing the basic concepts required for diagnosis modeling as language primitives: components, connections, and various advanced structural properties such as generic ports and hierarchical components. However, the language also provides capabilities for structuring models at a larger granularity, analogous to features that promote modularity in programming languages. Components are grouped into types, which makes it possible to check and guarantee basic properties of components. In addition, components and types can be organized in *packages* which encompass a certain set of definitions. Any desired subset of these can be exported from the package and is then visible from the outside. Conversely, packages can be imported into other packages or into models, making their exported definitions usable at that point. This provides a much better basis for organizing a model than having it presented as a flat set of horn clauses for some theorem prover, for whose syntactic and semantic correctness (in terms of appropriate use of types, connections, and complete description of components) the modeler is responsible alone, without having domain-oriented language support. What is more, packages provide a convenient mechanism for defining and combining model fragments into full domain models.

## Conclusion

We have discussed the necessity of defining a standardized language to describe systems for diagnosis purposes. From our experience in working in close cooperation with people from industry, we have gained the impression os that such a modeling language would be highly beneficial to fostering acceptance and growth of model-based diagnosis applications. AD2L is a proposal for a modeling language that has been developed for industrial applications but should be extended to catch as many application areas as possible.

The development of AD2L is not yet complete, whether in terms of syntactic features or expressiveness. Possible extensions (currently in development) of AD2L include the use of disjunctions, providing language constructs for expressing relations between models and meta-knowledge, allowing specification of connection as observable/non observable, and finally, allowing to declare axioms and rules for filtering diagnoses (e.g., physical impossibilities). However, the use of AD2L system descriptions for the domains described in this paper has both shown the applicability of the language in a variety of domains, the possibility of mapping it to efficient implementations, and above all the clarity and brevity of the resulting descriptions. It thus provides a convenient keystone for a model-based diagnosis framework on which an actual diagnosis system implementation should be based.

## Acknowledgments

The authors wish to thank Christian Piccardi for implementing the AD2L compiler and the interface to our diagnosis engine. The work described in this paper has been partly funded by Siemens Austria research grant DDV GR 21/96106/4, the Hochschuljubiläumsstiftung der Stadt Wien grant H-00031/97 and the Austrian Science Fund Project N Z29-INF.

## References

Cascio, F.; Console, L.; Guagliumi, M.; Osella, M.; Panati, A.; Sottano, S.; and Dupré, D. T. 1999. Generating onboard diagnostics of dynamic automotive systems based on qualitative models. *AI Communications* 12(1/2).

Crawford, J.; Farquhar, A.; and Kuipers, B. 1990. QPC: A compiler from physical models into qualitative differential equations. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 365–372. Boston: Morgan Kaufmann.

Dague, P. 1995. Qualitative Reasoning: A Survey of Techniques and Applications. *AI Communications* 8(3/4).

Davis, R. 1984. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence* 24:347–410.

Davis, R. 1993. Retrospective on "diagnostic reasoning based on structure and behavior". *Artificial Intelligence* 59:149–157.

de Kleer, J., and Williams, B. C. 1987. Diagnosing multiple faults. *Artificial Intelligence* 32(1):97–130.

de Kleer, J., and Williams, B. C. 1989. Diagnosis with behavioral modes. In *Proc. IJCAI*, 1324–1330.

Dilger, W., and Kippe, J. 1985. COMODEL: A language for the representation of technical knowledge. In *Proceedings $9^{th}$ International Joint Conf. on Artificial Intelligence*, 353–358. Los Angeles, CA: Morgan Kaufmann.

Finin, T.; Labrou, Y.; and Mayfield, J. 1997. KQML as an Agent Communication Language. In Bradshaw, J. M., ed., *Software Agents*. AAAI Press / The MIT Press. 291–317.

Flores, J. J., and Farley, A. M. 1999. Reasoning about linear circuits: a model-based approache. *AI Communications* 12(1/2).

Forbus, K. D. 1984. Qualitative process theory. *Artificial Intelligence* 24:85–168.

Friedrich, G., and Nejdl, W. 1992. Choosing observations and actions in model-based diagnosis-repair systems. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, 489–498. Cambridge, MA: Morgan Kaufmann.

Friedrich, G.; Gottlob, G.; and Nejdl, W. 1990. Physical impossibility instead of fault models. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 331–336. Also appears in Readings in Model-Based Diagnosis (Morgan Kaufmann, 1992).

Friedrich, G.; Stumptner, M.; and Wotawa, F. 1999. Model-based diagnosis of hardware designs. *Artificial Intelligence* 111(2):3–39.

Genesereth, M., and Fikes, R. 1992. Knowledge Interchange Format, Version 3.0, Reference Manual. Technical report, Technical report Logic-92-1, Stanford University Logic Group.

Heck, F.; Laengle, T.; and Woern, H. 1998. A Multi-Agent Based Monitoring and Diagnosis System for Industrial Components. In *Proceedings of the Ninth International Workshop on Principles of Diagnosis*, 63–69.

Iwasaki, Y.; Farquhar, A.; Fikes, R.; and Rice, J. 1997. A Web-Based Compositional Modeling System for Sharing of Pysical Knowledge. In *Proceedings $15^{th}$ International Joint Conf. on Artificial Intelligence*.

Malik, A., and Struss, P. 1996. Diagnosis of dynamic systems does not necessarily require simulation. In *Proceedings of the Seventh International Workshop on Principles of Diagnosis*.

Minoux, M. 1988. LTUR: A Simplified Linear-time Unit Resolution Algorithm for Horn Formulae and Computer Implementation. *Information Processing Letters* 29:1–12.

Piccardi, C. 1998. AD$^2$L An Abstract Modelling Language for Diagnosis Systems. Master's thesis, TU Vienna.

Reiter, R. 1987. A theory of diagnosis from first principles. *Artificial Intelligence* 32(1):57–95.

Snooke, N. 1999. Simulating electrical devices with complex behavior. *AI Communications* 12(1/2).

Steimann, F.; Fröhlich, P.; and Nejdl, W. 1999. Model-based diagnosis for open systems fault management. *AI Communications* 12(1/2).

Struss, P., and Dressler, O. 1989. Physical negation — Integrating fault models into the general diagnostic engine. In *Proceedings $11^{th}$ International Joint Conf. on Artificial Intelligence*, 1318–1323.

Stumptner, M., and Wotawa, F. 1998. Model-based reconfiguration. In *Proceedings Artificial Intelligence in Design*.

Stumptner, M., and Wotawa, F. 1999. Debugging Functional Programs. In *Proceedings $16^{th}$ International Joint Conf. on Artificial Intelligence*.

Williams, B. C., and Nayak, P. P. 1997. A reactive planner for a model-based executive. In *Proc. $15^{th}$ IJCAI*, 1178–1185.

Wotawa, F. 1999. Guest–Editorial Special Issue on Model-Based Reasoning. *AI Communications* 12(1/2).

## Appendix: Full Adder Implementation in AD2L

The following AD2L program implements a digital full adder with three inputs and two outputs.

```
component andgate
  input i1, i2: bool.
  output o: bool.

  default behavior nab
    comment "The normal behavior"
    action "Do nothing, for this is the correct behavior"

    Val(i1,true), Val(i2,true) =:= Val(o,true).
    Val(i1,false) =: Val(o,false).
    Val(i2,false) =: Val(o,false).
  end behavior

  behavior stuckAt1
    comment "The output is stuck at 1"
    prob 0.1
    cost 1000
    action "Throw the whole thing away"

    =: Val(o,true).
  end behavior

  behavior stuckAt0
    comment "The output is stuck at 0"
    prob 0.1
    cost 1000
    action "Throw the whole thing away"

    =: Val(o,false).
  end behavior

end component

component orgate
  input i1, i2: bool.
  output o: bool.

  default behavior nab

    exists INPORTS : Val(INPORTS, true) =: Val(o,true).
    forall INPORTS : Val(INPORTS, false) =:= Val(o,false).
  end behavior

end component

component xorgate
  input i1, i2: bool.
  output o: bool.

  default behavior nab

    Val(i1,V1), Val(i2,V2) =: Val(o,V1 != V2).
    Val(o,true), Val(i1, V1) =: Val(in2, not V1).
    Val(o,true), Val(i2, V2) =: Val(in1, not V2).
    Val(o,false), Val(i1, V1) =: Val(in2, V1).
    Val(o,false), Val(i2, V2) =: Val(in1, V2).
  end behavior

end component

component halfadder
  input i1, i2 : bool.
  output s, c : bool.

  subcomponents
    xor1 : xorgate.
    and1 : andGate.
  end subcomponents

  connections
    c1: i1 -> and1(i1).
    c2: i2 -> and1(i2).
    c3: and1(o) -> c.
    c4: i1 -> xor1(i1).
    c5: i2 -> xor1(i2).
    c6: xor1(o) -> s.
  end connections

end component

component fulladder
  input i1, i2, i3 : bool.
  output s, c : bool.

  subcomponents
    halfadder1, halfadder2 : halfadder.
    or1 : orgate.
  end subcomponents

  connections
    c1: i1 -> halfadder1(i1).
    c2: i2 -> halfadder1(i2).
    c3: halfadder1(s) -> halfadder2(i1).
    c4: halfadder1(c) -> or1(i1).
    c5: i3 -> halfadder2(i2).
    c6: halfadder2(s) -> s.
    c7: halfadder2(c) -> or1(i2).
    c8: or1(o) -> c.
  end connections

end component
```