An Environment and Language for Industrial Use of Model-based Diagnosis[1]

Gerhard Fleischanderl, Herwig Schreiner [2] and Thomas Havelka, Markus Stumptner, Franz Wotawa [3]

Abstract. Model-based diagnosis provides a well founded theory and a set of algorithms for finding and fixing a misbehavior caused by components. Actually applying model-based diagnosis effectively requires a flexible implementation which is capable of handling the differing requirements of multiple application domains. The diagnosis framework described in this paper has been developed for the purpose of being used in an industrial setting. It derives a significant part of its effectiveness from being integrated with a component oriented language for describing diagnosis models. The framework itself contains a class library comprising several different diagnosis engines having a standardized interface and allows rapid prototyping of diagnosis applications. The paper describes the framework, shows application domains, where the framework was applied, and gives an overview of the capability of the system description language AD2L.

1 Introduction

Model-based diagnosis [17, 3] has been successfully applied to different domains in the last years, including such diverse fields as automotive industry, space exploration, and software debugging. In this paper we describe a diagnosis framework that was developed to provide a class library for diagnosis in industrial settings, and the language AD2L which is the main interface to the primary users of the framework (i.e., engineers working on diagnosis system descriptions). Parts of the framework have been used in projects such as the Design Diagnosis Project [8], and for implementing a number of prototype systems.

The framework comprises classes implementing different diagnosis engines that provide core diagnosis functionality, i.e., diagnosis and measurement selection, and the parser for the language AD2L, which was designed as a general language for describing component and system models. AD2L was designed to be easy understandable and to allow for writing models that are independent from use of a particular diagnosis engine or implementation platform.

The paper is organized as follows. In the next section we present three different application areas where our framework has been successfully applied. This is followed by sections describing the internals of the implemented diagnosis kernel and the language AD2L.

[1] The work described in the paper has been partially supported 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.

[2] Siemens Austria, PSE PRO LMS, Erdbergerlände 26, A-1030 Wien, Austria, Email: {Gerhard.Fleischanderl,Herwig.Schreiner}@siemens.at, Authors are listed in alphabetical order

[3] Technische Universität Wien, Institut für Informationssysteme, Favoritenstraße 9-11, A-1040 Wien, Austria, Email: {havelka,mst,wotawa}@dbai.tuwien.ac.at, Authors are listed in alphabetical order

2 Application areas for diagnosis

In this section we will describe three real-world application domains that are used to design and test model-based diagnosis systems - at the representation and implementation levels.

2.1 ASIC design specifications

Domain characteristics: During the design process for an ASIC, specifications are written in VHDL. VHDL is an Ada-like programming language developed for describing hardware components and systems. An ASIC specification is developed as a sequence of VHDL programs which describe the ASIC in more and more detail. At each step the ASIC developer might introduce errors into the VHDL program at the more detailed level. Typically, ASIC development projects are carried out in teams of several engineers. The separately developed parts of the circuit specification are then put together to be tested and debugged [15, 20].

Diagnosis task: By analyzing a VHDL specification at level N+1 (produced by adding more structure and detail to level N) the diagnosis tool VHDLDIAG supports the developer by localizing the part in the VHDL program where an error was introduced (see figure 1).

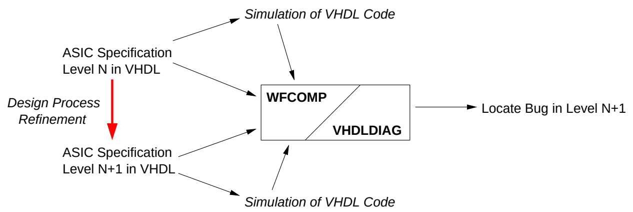

Figure 1. VHDLDIAG: Diagnosis of ASIC Specifications

Diagnosis method: VHDLDIAG uses behavior descriptions to represent individual elements of the VHDL language as diagnosis components. The behavior models have to be in sync with the tools and VHDL libraries used by the developers. By parsing the VHDL specification, the structure of the program is determined. Outputs from simulations are used as measurements.

After developing a VHDL specification at level N+1, the developer tests this new specification with a VHDL simulator. The outputs (signal traces) may either be checked against the simulated outputs of the specification at level N (the so-called "golden model") or may be declared OK or NOT_OK by the developer.

Critical application factors: The diagnosis system must be seamlessly integrated with the other tools used by the developers, e.g. compiler and simulator. Error locations have to be localized quickly and efficiently, i.e. with little intervention by the developer. In all, VHDLDIAG offers the developers clear benefits which are not found in other ASIC development tools.

2.2 Software parameters in a PBX

Domain characteristics: A PBX (private branch exchange) is a switching system for phone and data connections. The features of a PBX are mostly provided by its software which can be parameterized very flexibly. These parameters include technical characteristics of lines and subscribers, and functions enabled for certain lines (e.g. callback). However, incorrect configurations (usually due to operator error or slips in the customer's specifications) can occur. See figure 2 for a component oriented view of a phone switching network.

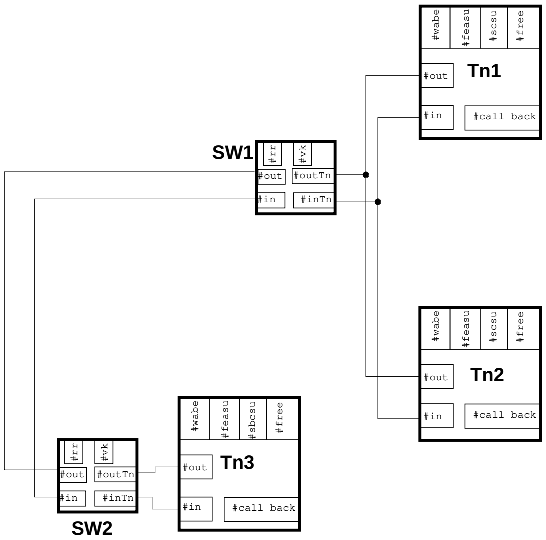

Figure 2. Configuration view of an example telephone network

Diagnosis task: If a feature is not available to a subscriber although it should be, this means some of the system's parameters were incorrectly set and the parameters having false values have to be found. If possible, the correct values for the parameters ought to be proposed as well. The observations shall be carried out as cheaply and efficiently as possible.

Diagnosis method: The behaviors of component types and feature types are described in the diagnosis model. The behavior descriptions map the parameter values to the functionality of the whole PBX.

The whole system structure cannot be represented completely because a single PBX may have several thousand subscriber lines, which would create too many elements for the diagnosis task. Instead the elements are represented in a generic manner. For example, one to three subscriber lines are typically involved in the diagnosis. The properties of these lines and the connections between them are established during diagnosis.

During diagnosis, the values of parameters are queried. Other measurements are the effects observed at subscriber lines and terminals, and the availability of features to the subscribers, e.g. "callback-LED is on or off." The observations may come from online connections to the PBX or from answers to user dialogs [19].

Critical application factors: The user interface for presenting questions has to give clear guidance and instructions on what the user should do next. The sequence of questions has to be reasonable, i.e. the user must be able to understand why a question is posed and why some question is posed before another.

If many parameter values are required to compute a diagnosis, obtaining the values via an online connection to the PBX is preferred, especially when a network of PBXs is diagnosed because this greatly increases the number of parameters and measurements.

2.3 Audio routing system

Domain characteristics: The routing system described here is a switching system for high-quality audio signals and is used in broadcasting stations. The main components of the routing system are input amplifiers, output amplifiers, and a switching matrix. More than one input-channel can be routed to one output-channel (dubbing). On top of the physical structure, a logical structure is used for control purposes.

Measurements can be carried out via a set of standard output plugs that are connected to the main control room. Thus, tests of the system state that use exclusively this set of plugs are easy and cheap. To measure a signal at other plugs, an operator has to stand right in front of the routing system, which is more time-consuming to do. Automated test-equipment can be used in some cases.

Diagnosis task: If a defect occurs, the faulty component has to be found quickly and reliably. There should be as few measurements as possible, and these measurements should be cheap.

Diagnosis method: The structure of an audio routing system is represented with its components and connections. To carry out measurements at the standard output plugs, the logical structure (control structure) has to be modeled, too.

The normal behavior of components is to route the audio signal without jamming or interference. In addition, failure models are used to describe particular behavior for defects that are frequently found.

Critical application factors: The most frequent test is whether an output signal is identical to its respective input signal. This test requires two measurements and one comparison which ought to be carried out without user intervention via an online connection to the routing system. The structure of the routing system has to be derived automatically from its configuration data.

3 Diagnosis Kernel

The diagnosis kernel implements all classes and methods necessary for building a diagnosis application. It was designed for flexibility and ease of use. The diagnosis kernel framework comprises generic classes for representing general interfaces and specific classes implementing the functionality. Figure 3 gives an overview of the currently implemented parts. The diagnosis engine on the right is divided into a diagnosis system and a theorem prover. The diagnosis system implements a diagnosis algorithm and stores knowledge about observations, connections, and components of a specific system. The theorem prover stores the behavior of the component to allow checking whether a system together with the observations and assumptions about the correctness of components is consistent or not. In cases where a consistency check is not necessary, a theorem prover is not used, e.g., the implementation of the TREE algorithm [18] requires

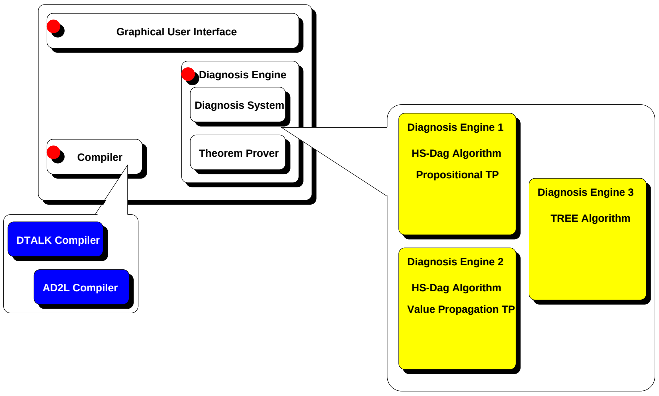

Figure 3. The Diagnosis Kernel (DiKe)

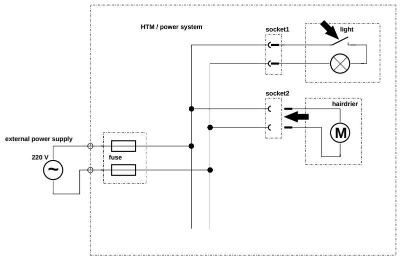

Figure 4. Parts of a home power network

no explicit theorem prover. The implementation of Reiter's hitting set algorithm [17, 11] on the other hand needs a theorem prover.

Currently, our framework provides three different diagnosis engines. Two engines use Reiter's algorithm while the other implements the TREE algorithm. Although the diagnosis algorithm is the same for the first two implementations, they use different theorem provers. One uses a propositional theorem prover and the other a constraint systems and value propagation. All concrete implementations have the same generic superclass. The generic diagnosis system class provides the interface, e.g., names of methods for executing diagnosis, requesting the next optimal measurement point, adding and removing observations, and others. The user writing an application using our diagnosis framework should choose the most appropriate diagnosis engine. If the model contains operations on numbers, the user should choose the value propagation algorithm. If the model is tree structured as defined in [18] the user should take the TREE algorithm. In all other cases the algorithm using the propositional theorem prover ensures best runtime performance that is almost equal and sometimes better than the performance published for other algorithms [9, 21].

The diagnosis kernel provides two languages for describing specific diagnosis systems, e.g., a digital full-adder. The first language, DTalk is closely related to Smalltalk syntax and semantics. For every kind of diagnosis engine there are specific language constructs representing the distinct behavior descriptions. While the knowledge about structural properties of a diagnosis systems are almost the same for every engine, this is not the case for the component models of DTalk. Therefore, we have developed a more general language *AD2L* to overcome this problem. Models written in AD2L are not restricted to one diagnosis engine, although currently only the transformation of AD2L programs into the constraint based diagnosis engine is supported.

Apart from classes for representing diagnosis knowledge, we have added classes for building user interfaces to the diagnosis kernel, to enable rapid prototyping of complete diagnosis applications. Using the demo applications and the diagnosis kernel classes as starting point, a first prototype of a diagnosis system implementing most of the required diagnosis functionality can be developed quickly. One of the demo interfaces uses a text-based user interface allowing to load systems and handle observations and other diagnosis knowledge, e.g., fault probabilities. The second variant uses a graphical approach for representing components and connections, similar to a schematics editor. Both applications provide messaging interfaces for starting the diagnosis and measurement selection process.

The diagnosis kernel, i.e., the class library for the user interface, the diagnosis engine, and the compiler, was implemented in Visualworks Smalltalk. Diagnosis and measurement selection runtimes are competitive with other implementations [9, 21, 5]. Parts of our VHDL debugger [8, 20, 22] were implemented using the diagnosis kernel.

4 The AD2L Language

The purpose of designing a dedicated system description language for model-based diagnosis is to support the user in writing the actual models. He should not be required to engage in applications programming, and the language should provide constructs to directly express the basic primitives that are generally used in system descriptions for model-based diagnosis. In other words, the language is supposed to provide a vocabulary that corresponds to the structure generally present in system descriptions for various domains, such as those described in Section 2.

We assume a diagnosis model to be composed out of smaller *model fragments*. Such a model fragment describes the behavior of a single component, e.g., a n-input AND gate, whereas a complete model describes the structure and behavior of a whole system in a logical way. The art of writing model fragments is that of describing the behavior in a context independent way, i.e., the behavior description of a component should not determine its use. In practice context independence cannot always be achieved, nor is it possible to define a language that guarantees context independence.

In this section we introduce the basic concepts of the AD2L language [16] designed for the purpose of communicating diagnosis knowledge. Instead of formally describing the language we show its capabilities using an example from the electrical domain. Consider a home power network, which typically involves a connection to the local power supplier, fuses, sockets, and devices attached to sockets: lights, washers, and other power consumers. Figure 4 shows a small part of such a net.

In order to write a model for a power network we (1) define types for connections, (2) declare a model fragment for every component, and (3) connect the fragments to receive the final model.

4.1 Defining types

Types are used for representing the domain of connections and component ports. In AD2L there are 5 predefined types: boolean, character, string, integer, and real, with some predefined functions, e.g. +,*, and others for integer and real values. In addition, the programmer can declare enumeration types. For example, in the power network domain we want to describe a qualitative model for currents and voltages, only using the information about whether a current or voltage flows or not. In this case we define the following type:

```
type electrDomain : { "on", "off" }.
```

Apart from such simple enumeration types, AD2L allows the use of predicates and the specification of tolerances and equivalences.

```
type quantDomain : real tolerance [ -5% , 10% ].
type myLogic : { '0' , 'L' , '1' , 'H' , 'X' , 'Z' }
   equivalence { '0' = 'L' , '1' = 'H' }.
```

The tolerances and equivalences are used for determining a contradiction during computation. For example, if we can derive the value '1' for a connection S and we have an observation 'H' for the same connection, then no contradiction arises for *myLogic* connection. If no equivalence relations are defined a contradiction occurs because it is assumed that a connection can only have one value.

The use of predicates in type declarations is another feature of AD2L. Consider the case where a connection can have several values, e.g., a radio link that broadcasts the signal of several channels at the same time. The type for this connection is defined as:

```
type channel : { "nbc", "cnn", "abc" }.
type radioLink : { predicate online ( channel ) }.
```

The *channel* type enumerates all possible channels that can be broadcasted. A contradiction only occurs in this case if a connection of type *radioLink* has a predicate and its negation as its value at the same time, e.g., *online("abc")* and *-online("abc")*.

Using types for connections has two advantages. The first advantage is that type checking can be performed at compile time. The second is that the list of domain values can be employed at the user interface level to present a list of possible values, or for checking the validity of user input after data entry.

4.2 Writing behavior models

The *component declaration statement* is the basic tool in AD2L for describing the interface and behavior of components. AD2L distinguishes between two different component declarations, base components and hierarchical components. Atomic components have a fixed, declared behavior and cannot be further subdivided. Hierarchical components derive their behavior from their set of internal subcomponents which are separately described. The subcomponents themselves may either be hierarchical components or atomic components.

Using the power net example, we now show the use of AD2L for writing atomic components. Verbally speaking, a light is on if its switch is on and it is connected to a current source. Formally, this behavior can be described in AD2L as follows:

```
component light
   comment "This is a qualitative model of a light"
   input current, voltage : electrDomain.
   input switch_on : bool.
   output light_on : bool.
   default behavior nab
      Val(switch_on,true), Val(voltage,on) =:= Val(current,on).
      Val(current,on) =:= Val(light_on,true).
      Val(light_on,false) =:= Val(current,off).
      Val(light_on,false) =:= Val(switch_on,false).
   end behavior
   behavior ab
      =: Val(current,off).
      =: Val(light_on,false).
   end behavior
end component
```

In the first line of the AD2L declaration of the component *light*, a comment is given. It is followed by the declaration of the interface, i.e., the *ports* which are used for connecting different components via *connections*. The AD2L compiler checks the types of connected ports and reports an error if they are not equivalent. In our case we define 4 ports: *current*, *voltage*, *switch_on*, *light_on*. The declaration of interfaces allows to specify whether a port is an input or output port or both (**inout**). Note that this information is not used to restrict the behavior description. It is intended to be used by diagnosis engines to determine a focus set or to optimize questions to the user about values. In addition, in AD2L the programmer can specify parameterizable *generic ports*. A generic port can be used to configure the component for different systems. For example, writing

```
generic Width : integer = 2.
```

defines a component with a generic number of inputs. The input definition in this case is

```
input i[1-Width] : bool.
```

After the interface, the behavior of the component can be defined. It is possible to define several behaviors. Each of them has a name (also called a *mode*), e.g., *nab* standing for *not abnormal*. In the example we distinguish between two modes. One defines the expected and the other the faulty behavior of *light*. AD2L requires one mode to be designated as default mode. The default behavior is used by the diagnosis engine as a starting point for diagnosis.

A behavior itself is described using rules. A rule consists out of two parts (the left and the right side) separated by an operator =: or =:=. For rules of the form L =: R the semantics are easy: If L evaluates to true, then all predicates in R must be true. Rules of the form L =:= R are a shortcut for L =: R and L =: R. For rules of the form L =: R the left side is called condition and the right side action part.

The left and the right side of rules are conjunctions of predicates. Disjunctive sentences have no direct representation in AD2L for complexity reasons. Predicates are predefined. The use of quantifiers is possible. The most important predicate is the **Val** predicate. The first argument is the port and the second the value of the port. It evaluates to true if the port has the given value. Another important predicate is **Cond** with a condition as the only argument. If the condition is true, the predicate evaluates to true. For example, the rule

```
Val(anInput,X), Cond(X>20) =: Val(anOutput,true).
```

specifies that if the value of *anInput* is greater than 20 the port *anOutput* must contain the value *true*. Note that **Cond** can only be used in the condition part of a rule. (Thus, in rules containing **Cond** the use of =:= is not allowed.) Another predicate is **Fail** raising a contradiction. This predicate has no arguments and can only be used in the action part of a rule. Again, its use in =:= rules is not allowed.

The use of quantifiers in rules is defined in AD2L. The intention is to use quantifiers for making the model as concise as possible. For example a quantifier can be used in the case we have to set all input ports to a specific value.

=: **forall INPUTS** : **Val**(**INPUTS**,on).

Note, that the existential quantifier (**exists**) can only be used in the condition part. In this case only the =*:* rule operator is allowed. The **forall** can be used in both parts of the rule. The quantification operator only influences the part of the rule where it is used. All of this restrictions are necessary to avoid complexity problems. The variable **INPUTS** is a built-in variable storing all inputs of the current component. There are several other built-in variables predefined in AD2L, e.g., **OUTPUT** and others. The user can also define variables using the **variable** declaration that must be located in the interface part of the component declaration. All variables are restricted to a finite domain.

We define the semantics of quantifiers based on the semantics of rules and predicates.

Forall Conjunctive sentences of the form ***forall*** *X: P(X) op A* are transformed into a single sentence *P(*v_1*), ..., P(*v_n*) op A*, where v_i is an element of X and *op* is either =*:* or =*:*=.

Exists Conjunctive sentences of the form ***exists*** *X: P(X) =: A* are transformed into several sentences *P(*v_1*) =: A, ..., P(*v_n*) =: A*, where v_i is an element of X.

The user can extend the core behavior definition by additional properties, i.e., repair costs, actions, and probabilities.

```
component light
...
  default behavior nab
    prob 0.999
    cost 2
    action "Replace the bulb"
    Val(switch_on,true), Val(voltage,on) =:= Val(current,on).
...
```

As stated above, hierarchical components can also be defined in AD2L. Their declaration is discussed in the next section. We decided not to distinguish between hierarchical components and systems because there is no conceptual difference between them - both contain components and connections.

4.3 Writing system models

Systems and hierarchical components consist of components and connections. Components can be either atomic components or again hierarchical components. The behavior of a system and a hierarchical component is given by the behaviors of the subcomponents. A hierarchical component can only have two behaviors. If it works correctly, all subcomponents are assumed to work correctly as well. The subcomponent behavior is given by their default behavior. In the other case, where the hierarchical component is assumed to fail, nothing can be derived. The probability of a hierarchical component C is computed using the probabilities of the default modes of the subcomponents $\{C_1, \ldots, C_n\}$

$$p(nab(C)) = \prod_{i=1}^{n} p(default\_mode(C_i)).$$

From the rules of probability theory follows

$$p(ab(C)) = 1 - p(nab(C)).$$

The user can define systems and hierarchical components by (1) declaring the used subcomponents, and (2) defining the connections between them. In our example the power net can be described at the system level as follows:

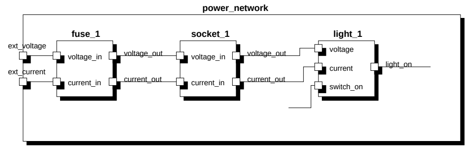

Figure 5. The graphical representation of *power_network*

```
component power_network
  input ext_voltage, ext_current : electrDomain.
  subcomponents
    fuse_1 : fuse.
    socket_1 : socket.
    light_1 : light
  end subcomponents
  connections
    ext_voltage -> fuse_1(voltage_in).
    ext_current -> fuse_1(current_in).
    fuse_1(voltage_out) -> socket_1(voltage_in).
    fuse_1(current_out) -> socket_1(current_in).
    socket_1(voltage_out) -> light_1(voltage).
    socket_1(current_out) -> light_1(current).
  end connections
end component
```

The graphical representation of the $power\_network$ system is given in figure 5.

4.4 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 enough expressive power for formulating models used in a variety of industrial domains or for education 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 [14]. 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.

5 Related Research and Conclusion

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 [2] 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 [13] almost no work in the direction of providing tools for handling models and model libraries has been done. In [13] 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) [10]. CML combines languages used for describing systems using Qualitative Process Theory [7] and the Qualitative Physics Compiler [1]. Other approaches for sharing diagnosis knowledge include [12] where KQML [6] 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 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 [16] 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 [4], but have not found general use in industrial applications. AD2L on the other hand was developed in collaboration with industry.

The development of AD2L is not yet complete. 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 the model-based diagnosis framework on which the actual diagnosis system implementations are based. The combination of both provides an effective and stable platform for using model-based diagnosis in an industrial setting.

REFERENCES

[1] James Crawford, Adam Farquhar, and Benjamin Kuipers, 'QPC: A compiler from physical models into qualitative differential equations', in *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 365–372, Boston, (August 1990). Morgan Kaufmann.

[2] Philippe Dague, 'Qualitative Reasoning: A Survey of Techniques and Applications', *AI Communications*, **8**(3/4), (1995).

[3] Johan de Kleer and Brian C. Williams, 'Diagnosing multiple faults', *Artificial Intelligence*, **32**(1), 97–130, (1987).

[4] Werner Dilger and Jörg Kippe, 'COMODEL: A language for the representation of technical knowledge', in *Proceedings 9^{th} International Joint Conf. on Artificial Intelligence*, pp. 353–358, Los Angeles, CA, (August 1985). Morgan Kaufmann.

[5] Yousri El Fattah and Rina Dechter, 'Diagnosing tree-decomposable circuits', in *Proceedings 14^{th} International Joint Conf. on Artificial Intelligence*, pp. 1742 – 1748, (1995).

[6] Tim Finin, Yannis Labrou, and James Mayfield, 'KQML as an Agent Communication Language', in *Software Agents*, ed., Jeffrey M. Bradshaw, 291–317, AAAI Press / The MIT Press, (1997).

[7] Kenneth D. Forbus, 'Qualitative process theory', *Artificial Intelligence*, **24**, 85–168, (1984).

[8] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa, 'Model-based diagnosis of hardware designs', *Artificial Intelligence*, **111**(2), 3–39, (July 1999).

[9] Peter Fröhlich and Wolfgang Nejdl, 'A Static Model-Based Engine for Model-Based Reasoning', in *Proc. 15^{th} IJCAI*, Nagoya, Japan, (August 1997).

[10] M.R. Genesereth and R.E. Fikes, 'Knowledge Interchange Format, Version 3.0, Reference Manual', Technical report, Technical report Logic-92-1, Stanford University Logic Group, (1992).

[11] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson, 'A correction to the algorithm in Reiter's theory of diagnosis', *Artificial Intelligence*, **41**(1), 79–88, (1989).

[12] Florentin Heck, Thomas Laengle, and Heinz Woern, 'A Multi-Agent Based Monitoring and Diagnosis System for Industrial Components', in *Proceedings of the Ninth International Workshop on Principles of Diagnosis*, pp. 63–69, Cape Cod, Massachusetts, USA, (May 1998).

[13] Yumi Iwasaki, Adam Farquhar, Richard Fikes, and James Rice, 'A Web-Based Compositional Modeling System for Sharing of Pysical Knowledge', in *Proceedings 15^{th} International Joint Conf. on Artificial Intelligence*, (1997).

[14] Michel Minoux, 'LTUR: A Simplified Linear-time Unit Resolution Algorithm for Horn Formulae and Computer Implementation', *Information Processing Letters*, **29**, 1–12, (1988).

[15] Zainalabedin Navabi, *VHDL Analysis and Modeling of Digital Systems*, McGraw-Hill, 1993.

[16] Christian Piccardi, *AD^2L An Abstract Modelling Language for Diagnosis Systems*, Master's thesis, TU Vienna, 1998.

[17] Raymond Reiter, 'A theory of diagnosis from first principles', *Artificial Intelligence*, **32**(1), 57–95, (1987).

[18] Markus Stumptner and Franz Wotawa, 'Diagnosing Tree-Structured Systems', in *Proc.* 15^{th} *IJCAI*, Nagoya, Japan, (1997).

[19] Markus Stumptner and Franz Wotawa, 'Model-based reconfiguration', in *Proceedings Artificial Intelligence in Design*, Lisbon, Portugal, (1998).

[20] Markus Stumptner and Franz Wotawa, 'VHDLDIAG+:Value-level Diagnosis of VHDL Programs', in *Proc. DX'98 Workshop*, Cape Cod, (May 1998).

[21] Brian C. Williams and P. Pandurang Nayak, 'A Model-based Approach to Reactive Self-Configuring Systems', in *Proceedings of the Seventh International Workshop on Principles of Diagnosis*, pp. 267–274, (1996).

[22] Franz Wotawa, 'Debugging synthesizeable VHDL Programs', in *Proc. DX'99 Workshop*, (1999).