Towards Model-based Engineering: Failure Analysis with MDS

Jakob Mauss[1], Volker May[1], Mugur Tatar[1]

Abstract: Model-based engineering supports different engineering tasks using rich digital models of products. Such models should improve product-related communication between engineers, should increase knowledge sharing and the re-use of partial solutions among different products and among different engineering tasks, and should offer new chances for systematic product design and validation. In this paper we present MDS, a tool that implements model-based engineering for various tasks related to the analysis of the behavior of a product when failures are present. We describe the modelling framework and the model analysis services provided by MDS, some recent applications of MDS, and some lessons learned from our work with applications.

1 INTRODUCTION

During the life span of an engineered product, such as a car, a train, an airplane, or a satellite, engineers are faced with various design and analysis tasks: conceptual and detailed design, parameterization, development of control software, cost estimation, safety and diagnosability analysis, production planing, testing, and maintenance.

On the one hand, specific tasks are tackled at different time points in a product's life span. All these tasks require task-specific expert knowledge. Therefore, typically, they are performed by **different engineers**. However, all tasks target one and the **same product**. Hence, to solve a task, an engineer requires knowledge about the specific product, its functions, and its technical realization. Today, this requirement establishes a communication bottleneck: Tasks are delayed or processed after time consuming knowledge acquisition and re-engineering of the product, often based on incomplete or out-of-date information.

On the other hand, the same engineer is usually solving the **same task** for different, but **similar products**. This offers a chance for partial re-use of previously derived solutions that is not fully supported today.

Model-based engineering supports the different engineering tasks by providing and maintaining a digital model of the product. A model is passed along the engineering process chain and accumulates task-relevant knowledge about the product, such as its functional specification, bill of material (BOM), or geometry. The digital model puts product-related communication between engineers on a firm ground.

Product-data management (PDM) systems and infrastructure (e.g. intranet) and standards for product data interchange (e.g. xml-based exchange formats) needed to support model-based engineering are partially available or under development, and their integration into existing engineering processes is on the way. Once models and standardized model exchange formats are going to be available there will be a demand for tools that exploit the available information to solve additional tasks.

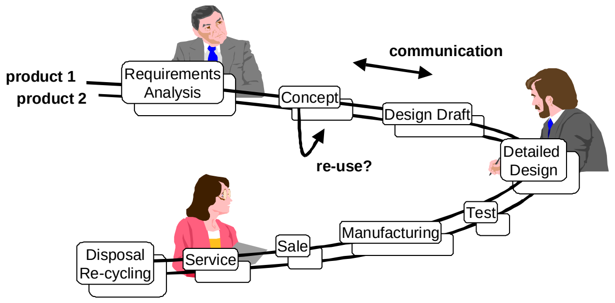

Figure 1. Engineering tasks along a product's life span

In this paper we describe MDS, a model-based tool that supports several engineering tasks related to the analysis of a system in case of failures, such as: on-board / off-board diagnostics development, safety analysis, diagnosability analysis, development of test procedures for end-of-line testing. MDS has been developed at DaimlerChrysler research since 1994. In 1999, MDS has been transferred to a professional software partner, Genrad Ltd. (UK), and will be commercially available soon. After the description of MDS in section 2, we present some applications that have been developed using MDS. Finally, we report some lessons learned during our work with MDS on applications. The paper is written on a non-technical level and has the goal to describe an existing model-based tool that has started to make its way into real applications.

2 MDS

In this section, we give an overview of MDS. We first describe how technical systems are modeled in MDS. Then we describe the various analysis tasks supported by MDS.

The design of MDS has been guided by the quest for broad applicability. For instance, rather then focussing on a restricted domain for behavioral descriptions, such as finite-domain (qualitative) models, linear models, non-linear models, or finite-state machines, we identified as a key requirement of a model-based tool in our industrial context its ability to deal with all of these domains. Hence, MDS was built on a rich and expressive modelling language - even to the price of incomplete model processing, or, sometimes, limited declarativity of models.

[1] DaimlerChrysler AG, Research and Technology, FT3/EW, Alt-Moabit 96a, D-10559 Berlin, Germany. Email: {Jakob.Mauss, Volker.V.May, Mugur.Tatar}@DaimlerChrysler.com

2.1 SYSTEM MODELS

In MDS a *system* consists of components and connections among components. A *component* is either a system - systems can be hierarchically composed - or an atomic constituent, such as a plug, a bulb, a valve, or a tank. Each component has a type that defines its generic properties and its behavior. These generic properties include: the component internal variables, the component ports, the behavior modes, the possible observation and control actions, and a graphical icon associated with the component type. The *ports* define the possible connection points of a component. The ports hold one or several variables that may be shared between components and connections. A *connection* represents a channel through which two components may exchange energy, information or matter. Typical examples are wires, pipes and shafts. Conceptually, there is no distinction between a component and a connection in MDS. A *behavior mode* of a component or a connection classifies a certain kind of behavior, i.e. nominal or faulty behavior. A valve model may have, for instance, 3 behavior modes 'ok', 'stuckOpen' 'stuckClosed', representing *nominal behavior*, and two different *component faults*. Every behavior mode has a numerical weight representing the likelihood, or the safety-criticality of the mode. A special internal variable of each component holds the currently assumed or inferred behavioral mode of the component. For every behavior mode, there may be a *behavior model*, representing the behavior of the component in the corresponding mode. A mode may also have no model, in this case representing unknown behavior.

A *behavior model* is described using a set of *constraints* relating the local variables of a component. Basic *variable domains* are numbers (integers, floating points, infinite precision rationals, meta-numeric extensions for infinity and infinitesimal), intervals, booleans, symbols and strings. Intervals are useful for expressing imprecise knowledge about numeric parameters and are processed by MDS using interval-arithmetic procedures. A constraint is described using one or several propagation rules. A *rule* has the form

$$c_1(X) \wedge c_2(X) \ldots \wedge c_p(X) \Rightarrow x_1 := f(X)$$

where $p \geq 0$, $X = (x_1, \ldots x_n)$ are local variables of the component, i.e. internal or port variables, and f and c_k can be arbitrary executable procedures. The conditions c_k have to return true or false. The function f returns either one or several alternative values from the domain of x_1, or a predefined symbol `#contradiction`, indicating that $X \notin \mathrm{dom}(f)$. The ability to use arbitrary executable code for representing component behavior is a key feature of MDS and provides great flexibility for modeling. This allows, for example, to easily integrate new *application-dependent variable domains*. Variable values may then be arbitrary objects (consider, for instance, a message with a structured content). Certain basic properties of a new domain such as equality, inconsistency or subsumption, have to be defined using the native implementation language of MDS, i.e. Smalltalk. This allows also to define domains where multiple consistent assignments of values to a variable are allowed – for instance for blackboard variables. Further constructs allow to define generic constraints or rule sets that encode specific relations (e.g. Ohm's law) and which can later be instantiated in the context of a specific component model.

Some of the internal parameters of a component can be declared to be *state variables*. This allows to define a component model as a deterministic or non-deterministic state machine. For example, a relay can be modeled as a deterministic finite-state machine having a state variable *s* and coil current *i* using the rule

$$s = closed \wedge i > 0 \quad \Rightarrow \; nextState(s) := open$$

When analyzing a system model having state variables, MDS performs forward simulation of the component models (concurrent finite-state machines), until a steady-state is reached, or a cycle is detected

MDS offers also means for representing possible interactions of a system or a component with a human user or an external agent. For this purpose *action models* can be attached to a component. Actions are associated with state-dependent *costs*, i.e. a number that reflects how easy or expensive is to carry out an action, and with *preconditions*, reflecting the situations in which the respective action is allowed to be performed. Because preconditions usually encode safety conditions for carrying out an action, they are considered to be fulfilled only if they are *entailed* by the model *irrespective* of the currently assumed fault modes. There are two categories of actions: *observation* and *control* actions.

A *control action* can be attached to a port variable and defines a persistent assignment of a specified value to the specified port variable. The assignment will hold after a certain delay in all the following states until it is overridden by another control action. A control action may also cause a sequence of internal state transitions of the system model, e.g. when switching on the power of a system that contains a relay (control action), the relay may change its switch state (state transition).

An *observation action* can be attached to any variable. It represents the ability of a human user or of an external agent to observe or measure the variable, provided that the specified action preconditions, defined in terms of the local variables of the component, are satisfied. For example, measuring the oil level of an engine may require that the engine is not running. Preconditions can be satisfied by applying appropriate control actions, i.e. by transitioning the system into another state.

Many of the applications that we had to address required the ability to model (discrete) dynamic asynchronous components / systems. In order to synchronize internal state transitions with external control actions in asynchronous systems, MDS assumes (a) that the internal state transitions are fast w.r.t the time span between two control actions, and (b) that the system reaches a steady-state after applying a control action. A system satisfying these assumptions is called *pseudo static* in our framework. Hence, control actions are only applied in steady states in pseudo-static systems.

Type definitions for variables, ports, connections, actions, components and systems are organized into a type library. Due to the locality of all constraint definitions - only variables from the scope of the type owning the constraint are available - the types are reusable in many different application contexts.

MDS is delivered with standard libraries for electrical, hydraulic, and mechanical domains. Hence, for a technical system from such a domain, modeling reduces mostly to drag-and drop operations: component types are selected from the library, placed on a drawing pane, connected using the connections available from the library, named, and parameterized. Each component on the drawing pane will already be associated with possible component faults, and default control and observation actions, as specified by the corresponding component type.

2.2 SUPPORTED ENGINEERING TASKS

MDS provides an engineer with various services for analyzing a given system model. In this section, we describe these services. Each service is implemented by a service-specific engine. All engines use a common set of core capabilities of MDS. The separation of core technology (e.g. constraint solving, search engines, dependency maintenance, etc) and service-specific engines (e.g. diagnosis) helps us to quickly add new services to MDS by reusing the already existing core. Another advantage of that architecture is that all services can directly benefit from future improvements of the core. The following description is informal.

MDS implements the ideas of consistency-based diagnosis, as presented in [1], [3], [4], [13]. For formal definitions of key notions such as diagnosis, minimal diagnosis, candidate, or minimal conflict, see there. To find out, how these definitions have been refined and extended in order to deal with pseudo-static systems, see [6], [7].

MDS is currently offering four services to the engineer.

1. **Interactive diagnosis and test proposal**. Starting from numerical or qualitative measurements or observations, MDS detects non-nominal behavior (if any), and guides the user through a sequence of further measurements and useful control actions (such as opening or closing plugs), until the faulty component is localized and its fault mode is identified, or no more useful measurements are found. When proposing a measurement or control action, MDS aims to minimize the costs associated with the actions, and to maximize the expected information gain of the measurement. A simplified information entropy measure (cf. [16]) is used to assess the expected information gain. MDS keeps track of possible side-effects of control actions during diagnosis, e.g. if we switch on the power of an electrical system that contains a short, a fuse may break. Previously non-faulty components may transition into a fault mode. MDS keeps track of these dynamically introduced cascading faults.
2. **Decision tree generation**. Starting from a given set of relevant faults, an initial situation (e.g., which plugs or valves are open), and an optional sequence of previous control actions and measurements, MDS derives a decision tree that aims to discriminate at best[2] among the given faults using the available measurements. The resulting decision tree is a compiled diagnostic solution. Such decision trees are useful for analyzing the diagnosability / testability during the system design, for developing on-/off-board diagnostic software, or for developing test procedures for end-of-line testing.
3. **Sensor placement**. Starting from a set of faults, an initial situation, a set of possible sensors (model variables) MDS computes the “minimal”[3] set of sensors that can be used for the detection and identification of the given faults. MDS can explain how to use the sensors for diagnosis by presenting a decision tree that uses only the proposed sensors for diagnosing the system. As above, MDS aims to minimize sensor costs, and to maximize the information gain achieved by the sensors.
4. **Support for safety analysis**. Some currently available services allow to visualize and compare faulty and normal simulation results, to generate some reports and explanations. These services will be further extended in future in order to automate the FMEA (failure-mode and effect analysis) and the FTA (fault tree analysis) of a system. See also the section on future work.

3 APPLICATIONS

During the last 6 years, parallel to the MDS software development, we have spent a significant modelling effort for a large number of applications. Prototype applications were developed for / with DaimlerChrysler business units from the automotive, train, aerospace and energy supply domains (last one formerly part of Daimler-Benz):

- Automotive divisions
  - Off-board troubleshooting of electric-electronic subsystems;
  - Diagnosis and failure analysis of automatic transmissions, complex hydraulic and mechanical subsystems, see also [12];
- Aerospace divisions
  - Integration tests for satellite electrical subsystems;
  - Decision tree generation for on-board diagnosis and sensor placement support for spacecraft propulsion subsystem, pneumatic-hydraulic subsystems, see also [8];
- Train and tram divisions
  - Troubleshooting of electric-electronic subsystems;
  - Troubleshooting of a communication and control subsystem (distributed controllers)
- Energy supply networks
  - Generation of safe control sequences for reconfiguration

Many of the applications brought with them new requirements and raised new problems from which we had to learn and which forced us to improve, extend or reconsider the representation primitives, the model libraries, the modelling language, the analysis algorithms and the provided services. The rest of this section illustrates two of these applications.

3.1 Troubleshooting a car electrical circuit

One of the earliest applications of MDS dealt with automating the workshop troubleshooting of car electric-electronic subsystems. Here part of an anti-blocking system (ABS) model is shown in Figure 2. Wires connect the ABS electronic control unit (ECU) via 3 plugs to a warning lamp L and two relay boxes B1 and B2 containing relays K1, K2 and a diode D. Considered defects include, for instance, broken plug-connectors, broken, shorted to ground or to battery wires, stuck switches, etc. Only a limited set of measurements and control actions with differing costs and preconditions are possible. For instance, voltage or resistance measurements are not possible to perform at any point. Usually this requires to open a plug connector (structural change) in order

[2] Prediction, consistency checking and test proposal are as during interactive troubleshooting. At each tree node the best (i.e. minimal costs, maximal benefit) sequence of control actions and measurement is selected under a one-step-look-ahead search. All possible outcomes of the measurement are followed then recursively by alternative tree branches.

[3] A best-first (under one-step-look-ahead) and greedy algorithm for sensor selection is used. This does not guarantee the global optimallity in general, but seems to achieve an acceptable balance between optimality and complexity.

to access a pin and to connect a measuring device (expensive action with additional safety preconditions).

The sample dialog given below shows how, starting from an observed symptom, MDS proposes control-actions to transition the system in useful states, and measurement actions to discriminate among possible diagnoses. It also shows that MDS tries to minimize costs for actions: Expensive actions are delayed as long as possible.

In the initial situation, K15 and K30 power the circuit with 12V, all 4 plugs are closed, and the ECU is performing a self-check. Normally, in this situation, the ABS warning lamp should be lit.

Step 1: Observation: L is not lit. MDS computes the first n leading diagnoses (e.g. n = 15) and proposes to perform a cheap build-in ABS self test (control action) to check the lamp L.

Step 2: We perform the ABS test. Now, L is lit. MDS recomputes the set of most plausible diagnoses which rules out several diagnosis related to the lamp circuit and plugs. MDS recommends to cancel the ABS lamp test (control action) and then to read out the voltage as measured by the ECU at one of its pins (cheap measurement).

Step 3: On request, the ECU tells us u = 11 V ± 3V. Only 9 single faults are left now. MDS recommends to reactivate the lamp test (control action) and to measure a certain voltage at the ECU with the voltmeter (expensive measurement, no other cheaper measurement would help).

Step 4: We measure 0V which rules out more candidates. Only 6 left. MDS recommends to switch on relay K1 via an ECU command (control action) and to measure a voltage.

Step 5: We measure 12 V. Only 5 diagnoses left. MDS proposes to open the plug at relay box B1 (expensive control action) and to measure the resistance of relay K1.

Step 6: We measure R = 8 Ω, which is the expected value. There are two diagnoses left - a corroded pin in the B1 plug or a broken wire inside B1. MDS tells us that there are no more measurements available to discriminate these two cases.

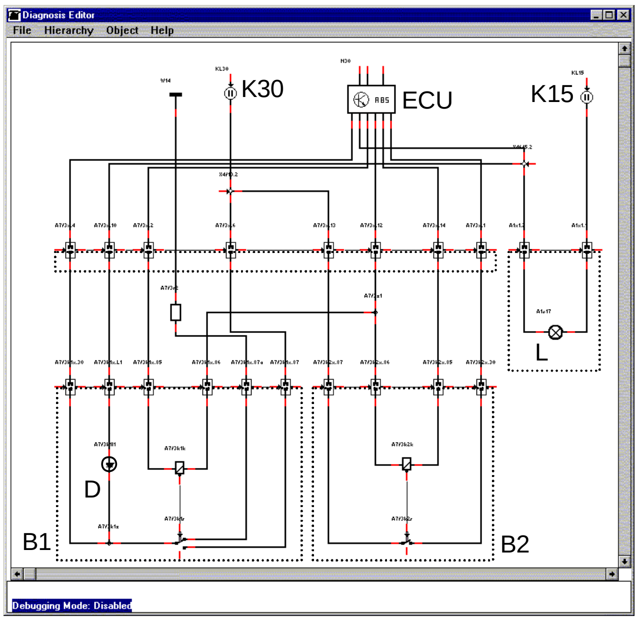

Figure 2. ABS electrical subsystem modeled with MDS

3.2 Supporting the design of a propulsion system

Figure 3 shows a hierarchical model of the propulsion system of the Automated Transfer Vehicle (ATV), an unmanned shuttle currently under development for supplying the International Space Station (ISS)[4]. The propulsion energy results from the spontaneous oxidation reaction (combustion) between fuel and oxidizer taking place in the large main and smaller attitude and orbit control thrusters. 8 membrane tanks, 4 for oxidizer, 4 for fuel, are used for storage. Helium is used as a pressurant to apply pressure to the membrane of the fuel and oxidizer tanks. This provides the energy for transporting the fuel and the oxidizer from the storage tanks to the thrusters. A fuel pump is not needed. Pressurant, fuel and oxidizer supply are controlled using a large number of valves that enable or disable the flow to different parts of the system. To increase fault tolerance, the supply structure of the system is highly redundant. The valves are controlled via an electrical system, the Propulsion Drive Electronics (PDE), which is not currently part of the model.

Among the considered component faults there are: valve stuck-open or stuck-closed, clogged filters, faulty pressure regulators, wrong sensor indications, internal leakage at valves, external leakage at welding points, etc. The model has been used to

- Generate decision trees to support the development of Fault Detection Isolation and Recovery (FDIR) on-borad software.
- Optimize sensor placement for the propulsion subsystem.

A future application, based on an extended version of MDS, will be the model-based failure-mode effect and criticality analysis (FMECA), in order to verify the hard fault-tolerance requirements of the propulsion system.

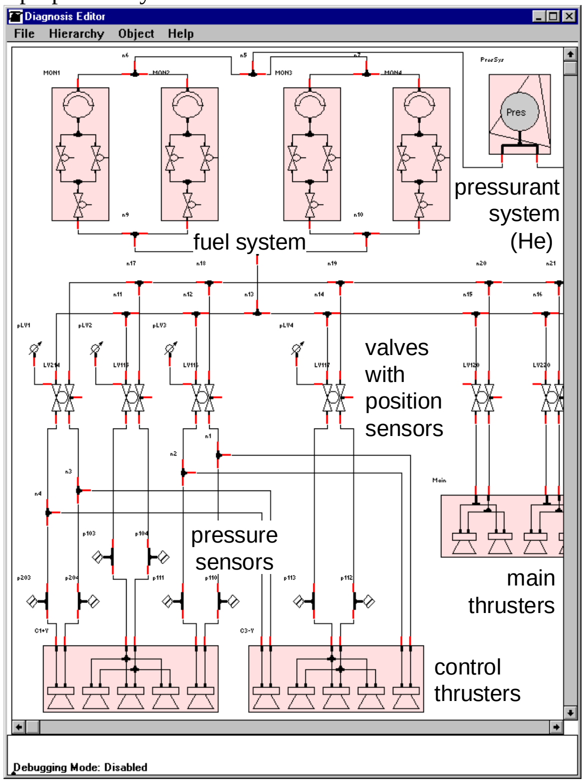

Figure 3. A fragment of the ATV propulsion subsystem modeled with MDS

[4] The model has been developed by our customers at DASA-RIO61 and is presented here with their kind permission.

4 LESSONS LEARNED

Since 1994, we have been trying to transfer model-based technology, as developed by academic research, into real applications. During this process, we gained three main insights

1. **Application-driven tool development is key to success**. The applications helped us to identify requirements that had not been addressed by academic research so far. In this process, new concepts emerged, e.g. the pseudo-static framework, and the impact of existing techniques had to be re-evaluated.
2. **Support for model-engineering is essential**. Modelling is a complex and expensive task. The hardest requirement that model-based engineering will have to face is that *it has to be cost-effective*. This requires, among others, powerful high-level modelling languages, support for explanation, model debugging, maintenance, customization, consistency checking, reusability, import / export, etc. Additional demands for integration with conventional engineering tools have to be considered.
3. **Professional software partner**. Since a research group can usually not provide the support required and expected by industrial users, convincing a professional partner to do this job is a crucial step towards successful transfer of a research prototype into applications.

In the rest of this section, we report some insights or opinions that we gained on the technical level.

Dynamic behavior and temporal reasoning. The pseudo-static framework, as described in section 2.1, emerged as a result of our applications related to the diagnosis of car electrical systems. Since then, it has proven to be useful in many other application domains. Its main advantage is that it allows us to handle time without deviating too much from a static point of view. In some cases, even when the correct system can be modeled as a pseudo-static system, the system oscillates in the presence of failures (e.g. a relay or valve is continuously opening and closing). Although such cycles can be detected, further concepts for synchronization and temporal abstraction are required in order to continue the diagnosis when such behaviors are exhibited. For instance, since instantaneous value measurements are hard to synchronize with the model, a more appropriate solution seems to be to switch the time-scale abstraction and to process temporally abstract observations (cf. [15]) such as 'oscillating', 'steady', etc.

Measurement / test proposal. When applying existing model-based algorithms ([3], [4], [16]) to off-board troubleshooting or end-of-line test generation, we also had to extend the measurement proposer. Not only the current state is searched for discriminating observable variables, but also states reachable by applying sequences of control actions to the current situation are considered. Best-first iterative deepening search algorithms are used for this purpose. However, the maximal effort spent for this search is currently specified in advance by the user. Deep search in large spaces is not tractable in practice. We believe that further research (or adaptation of results from the planning community) is needed here. The consideration of action costs and preconditions was a requirement in many of the applications that we had to consider.

Prediction based on local propagation. In MDS, model analysis is currently implemented by local propagation of value restrictions through a constraint network. It is well known that this kind of inference is incomplete. Due also to the (usual) limited observability, in typical domains such as electrical or hydraulic networks this incompleteness of the predictor has to be compensated by more sophisticated modeling (e.g. additional model variables, information exchange about surrounding constraint topology, etc). As a consequence, modeling can become an expensive time-consuming process requiring skilled modelers. Hence, we are convinced that for a wide spread of the model-based technology prediction should be based on more complete inference methods, possibly including computer algebra techniques.

Dependency recording using a TMS. When we started implementing MDS in 1994 we used an ATMS [2] to cache predictions, record dependencies, provide explanations and derive minimal conflicts. However, this turned out to be far too expensive for our applications. In most of our applications we use real-valued variables and intervals (see also the following discussion about qualitative/quantitative modeling) and large value domains. Depending also on the constraint topology this can lead to reduced reusability across contexts and high memory and bookkeeping effort demands. We had to go through several improvements of the TMS (see e.g. [5], [7]; currently we use a JTMS-set [7]) and of the techniques for focusing and controlling the propagation of intervals - for instance, we extended the concept of interval shadowing cf. [14] to handle also approximately equal values, we perform special cycle analysis to avoid to create hundreds of intervals, etc.

The pseudo-static framework (cf. [6]) was specially designed to "work well" with a TMS and to reuse predictions not only across diagnostic contexts, but also across time. The TMS records only atemporal inferences (dependencies among assumed inputs and states and entailed predictions, including the next state). Temporal reasoning requires meta-level reasoning w.r.t. the atemporal TMS propositional reasoning. This feature was essential for making the temporal analysis feasible. Prediction reusability across input vectors was a key factor in achieving acceptable times for the test-pattern search.

However, more sophisticated temporal analysis and temporal abstraction seems to be more complicated to implement in this framework.. Because we are continuously facing more and more complex and larger applications we will have to reconsider again the dependency recording and the caching tasks provided by the TMS.

Qualitative / quantitative models. Most of the past research work in the model-based diagnosis community focussed on qualitative modeling and propositional reasoning. Initially we believed that we will be able to work in most of the cases with models using small (qualitative) domains. During the time, especially due to the demands to build reusable model libraries for different system topologies and a growing spectrum of analysis tasks, we had to make the models more detailed and we shifted more and more towards quantitative interval-based models. The current state-of-the-art in qualitative modeling involves determining a set of relevant qualitative values for each variable, where relevance depends on the task we want to solve, the structure of the system and on the contexts that have to be considered. Hence, to find the appropriate set of qualitative values, the user has to analyze the model and the task, which puts additional burden on the user and introduces context-dependencies into the models. Both points contradict the spirit of the model-based enterprise. One could hide the abstract, qualitative level from the user by using qualitative abstractions as an internal, compiled representation only. However, compilation methods for such an approach are still missing, although academic research started some investigations in this direction [11].

We believe that purely qualitative or purely quantitative models are not going to provide the key for the wide-spread application of the model-based techniques in engineering and that we need the flexibility to use both of them and even to automate the switch of the abstraction level – but the last point is still a challenging research topic.

5 FUTURE WORK

Our current and future work with MDS is focussed on the following issues:

- Safety analysis: We are extending the representation primitives in order to allow to define component and system functions, i.e. knowledge about the intended behavior / use of a product. On-going work addresses also the generation of high-level (qualitative) concise explanations for the fault-effects propagation. This will improve the ability of MDS to support the engineer in the failure-mode and effect analysis (FMEA) and other safety analysis related services.
- Supporting design: We are currently extending the MDS core, such that it can also be used by SDR (System Design for Reusability), a model-based design support tool described in [10]. The long-term goal is to integrate the design support (synthetic) and the design analysis (analytic) activities.
- Constraint processing: We are starting to extend the constraint propagation with more complete analysis methods, based on the aggregation of relations [9]. This will also lead to more declarative and readable models, i.e. simplify modeling and model reuse.
- Temporal representation and reasoning: Demand-driven time scale abstraction for systems violating the pseudo-static behavior and appropriate techniques for dealing with systems exhibiting hybrid (discrete and continuos) dynamic behavior are also future research topics.
- Encapsulation: We intend to encapsulate each MDS engine in a software component, based on protocols such as COM or CORBA. This will ease integration of MDS into existing and future software environments.
- Model compilation, demand-driven abstraction and simplification: These techniques are probably going to be essential in order to analyze large and complex systems. Solutions with respect to these topics are still missing and they are challenging future research topics.

REFERENCES

[1] Raymond Reiter: A theory of diagnosis from first principles, *Artificial Intelligence*, 32(1), pp. 57-95, 1987

[2] Johan de Kleer: An Assumption-based TMS. *Artificial Intelligence*, 28, pp. 127-162, 1986.

[3] Johan de Kleer, Brian C. Williams: Diagnosing Multiple Faults. *Artificial Intelligence*, 32, pp. 97-130, 1987.

[4] Peter Struss, Oskar Dressler: The consistency-based approach to Automated Diagnosis of Devices. Brewka (Ed.): *Principles of Knowledge Representation*. CSLI Publications, pp. 267-311, 1996.

[5] Mugur Tatar: Combining the Lazy Label Evaluation with Focusing Techniques in an ATMS. *ECAI 94*, Amsterdam, 1994.

[6] Mugur Tatar: Diagnosis with Cascading Defects, *ECAI 96*, Budapest, Hungary, 1996.

[7] Mugur Tatar: Dependent Defects and Aspects of Efficiency in Model-Based Diagnosis, *Dissertation, Universität Hamburg*, 1997.

[8] Mugur Tatar, Peter Dannenmann: Integrating Simulation and model-based Diagnosis into the Life Cycle of Aerospace Systems: An Ongoing Project. In: *10th International Workshop on Principles of Diagnosis (DX-99),* Loch Awe, Scotland, pp. 273-280, 1999.

[9] Jakob Mauss, Martin Sachenbacher: Conflict-Driven Diagnosis using Relational Aggregations. In: *10th International Workshop on Principles of Diagnosis (DX-99),* Loch Awe, pp. 174-183, 1999.

[10] Frank Feldkamp, Michael Heinrich, Klaus-Dieter Meyer-Gramann: SyDeR - System Design for Reusability, *AI-EDAM, Special Issue on Configuration Design*, September 1998.

[11] Peter Struss, Martin Sachenbacher: Significant Distinctions Only: Context-dependent Automated Qualitative Modeling. 13th Int. Workshop on Qualitative Reasoning (QR-99), pp. 203-211, 1999.

[12] Bidian, P., Tatar, M., Cascio, F., Thesedier Dupre, D., Sachenbacher, M., Weber, R., Carlen, C.: Powertrain Diagnosis: A model-based approach, Proceedings of 1999 Vehicle Electronic Systems (ERA99), Coventry, UK, 1999.

[13] Hamscher, W., de Kleer, J., Console, L.: *Readings in Model-Based Diagnosis*. Morgan Kaufmann, 1992.

[14] Hamscher, W.: ACP: reason maintenance and inference control for constraint propagation over intervals. Proc. 9th Nat. Conf. On AI, Anaheim USA, 1991. Also in [13].

[15] Hamscher, W.: Modeling digital circuits for troubleshooting. *Artificial Intelligence*, 51(1-3). Also in [13].

[16] Johan de Kleer, Brian Williams: Diagnosis with behavioral modes. *Proc. IJCAI'89*. Also in [13].