All Roads Lead to Fault Diagnosis: Model-Based Reasoning with LYDIA

Alexander Feldman Jurryt Pietersma Arjan van Gemund

Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Mekelweg 4, 2628 CD, Delft, The Netherlands
Tel.: +31 15 2781935, Fax: +31 15 2786632
e-mail: {a.b.feldman,j.pietersma,a.j.c.vangemund}@tudelft.nl

Abstract

Model-Based Reasoning (MBR) over qualitative models of complex, real-world systems has proven succesful for automated fault diagnosis, control, and repair. Expressing a system under diagnosis in a formal model and infering a diagnosis given observations are both challenging problems. In this paper we address these challenges. By building a fault model of a real-world artifact (the fuel-system of a light aircraft), we introduce the software package for MBR LYDIA and show its applicability in practice. We demonstrate how structure exploitation and compilation can be used to attack the main challenge to MBR - its high computational cost. Last, we compare our approach to other state-of-the art techniques for MBR and analyze its performance.

1 Introduction

Automated reasoning over qualitative models has proven successful for a wide range of diagnostic, control and repair tasks, involving wafer scanners, interplanetary space-probes, and large radio-telescopes [11]. Motivated by the success of systems like Livingstone [14], we have embarked on the design and implementation of a system of our own, named LYDIA[1]. Comprising a *modeling language* and a set of translation and reasoning tools, LYDIA aims at improving the state-of-the-art in MBR by providing more *powerful* modeling primitives and *faster* diagnostic engines.

Recently there has been much interest in MBR and especially in model-based diagnosis (MBD) [7]. MBR involves two major challenges: (i) expressing the system under diagnosis in terms of a formal language, and (ii) performing the diagnostic inference given observations. In this paper we outline how the LYDIA approach addresses both challenges.

The biggest problem MBR is facing is its high computational cost. In particular, [17] shows that the time complexity of MBD related to the number of components comprising a system is in $\Sigma_2 P$ even in some very restricted cases. Researchers are looking for solution to this problem in two directions. One is exploiting structure which is present in any man-made system. Another, related approach is to compile the model to a form allowing faster, in the strict case polynomial-time, reasoning.

One of the most successful approaches to compilation is the work of Darwiche [3] in Decomposable Negation Normal Forms (DNNF). The latter work focuses on the algorithmic and computational aspects of propositional reasoning while our goal is to bridge the modeling, reasoning, and validation aspects of the whole concept. Similar to SAT, in diagnostic search, conflicts are an important source of speedup. The use of conflicts in MBR is one of the contributions in [18]. Exploiting hierarchy is another potential source of speedup. This can be used both in compilation-based approaches [2, 9] or directly [16].

[1] Language for sYstem DIAgnosis. The LYDIA package for model-based fault reasoning can be downloaded from `http://fdir.org/lydia/`.

One of the most comprehensive experiments involving model-based reasoning is the Deep Space 1 NASA mission [15], the latter aiming amongst others to test software components designated as strategic by NASA. The use of reasoning software for craft control, task planning, fault identification and recovery has resulted in reduced costs for mission execution as well as cheaper *design* of spacecraft systems which itself has contributed to the improved reliability throughout the whole mission.

The rest of this paper is organized as follows. In Section 2 we illustrate the main LYDIA language primitives by modeling the fuel system of a light aircraft. The workings of the diagnosis engine are discussed in the following Section 3. Experimental results and performance characteristics of our approach are shown in Section 4. Finally, conclusions and future work are discussed.

2 Modeling in Lydia

LYDIA is declarative modeling language specifically developed for Model-Based Diagnosis. The language core is propositional logic, enhanced with a number of syntactic extensions for ease of modeling. The accompanying toolset currently comprises a number of diagnostic engines and a simulator tool.

As a modeling example we use the fuel system of the Piper PA-28-161 light aircraft. Figure 1 shows its schematics. The composition of the model is determined by the level of detail we need to obtain in a system diagnosis. We assume that identifying one or more of the field-replacable components in Figure 1 is adequate for this purpose, therefore this schematic dictates the topology of the model.

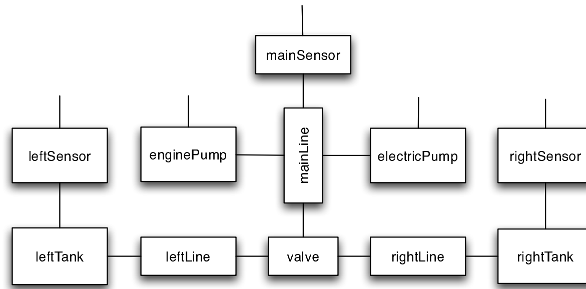

Figure 1: Schematics of the fuel system of Piper PA-28-161 light aircraft.

Next, we need to determine the proper system variables to be modeled. For this, the main functionality of the system is used as guidance. The main function of the fuel system is to provide an uninterrupted supply of fuel to the engine. This supply should have a specific quantity and should be controllable by the pilot. Hence, we choose fuel mass and it's derivative fuel mass flow, as leading system variables. These parameters are influenced by all system components. This influence is partially controllable and measurable.

The next listing shows parts of the LYDIA source for our model. The keyword `system` indicates the definition of a component. Consider, for example, the system `fuelTank`. This component has three variables of type mass, defined with the keyword `type`. The health of this component is of type `hContain`. Health variables (which are to solved by the Lydia diagnostic engine) in LYDIA are declared by the attribute `health`. An a priori probability is assigned as search heuristic for the diagnostic inference.

```
type mass = enum {zero, low, nom, high};
type hContain = enum { nom, leak, unknown };
type selectorPosition = enum { off, left, right };

system fuelTank(mass quantity, flow, lineFlow)
{
    hContain h;
    attribute health (h) = true;
    attribute probability (h) = cond (h) (hContain.nom -> 0.99; hContain.leak -> 0.009; ...);
```

```
    (h = hContain.nom) => (((quantity = mass.zero) => (flow = mass.zero)) and (lineFlow = flow));    10
    if (h = hContain.leak) {
        (quantity=mass.zero) => (flow=mass.zero);
        lineFlow = cond (flow) (mass.high -> mass.nom; mass.nom -> mass.low; mass.low -> mass.zero);
    }
}

/* Definition of other components... */

system fuelSystem(selectorPosition selector, bool enginePumpOn, electricPumpOn, mass leftSensorMass,    20
                  leftSensorFlow, rightSensorMass, rightSensorFlow, mainSensorFlowOut)
{
    mass leftMass, leftFlow, rightMass, rightFlow, leftLineFlow, rightLineFlow, mainFlow;
    mass leftLineFlowOut, rightLineFlowOut, mainFlowOut, mainMassOut, mainSensorMassOut;

    attribute observable(selector, enginePumpOn, electricPumpOn, leftSensorMass, leftSensorFlow) = true;
    attribute observable(rightSensorMass, rightSensorFlow, mainSensorFlowOut) = true;

    system fuelTank leftTank(leftMass, leftFlow, leftLineFlow);
    system fuelTank rightTank(rightMass, rightFlow, rightLineFlow);    30
    system sensor leftSensor(leftMass, leftFlow, leftSensorMass, leftSensorFlow);
    system sensor rightSensor(rightMass, rightFlow, rightSensorMass, rightSensorFlow);
    system line leftLine(leftLineFlow, leftLineFlowOut), rightLine(rightLineFlow, rightLineFlowOut);
    system selectorValve valve(selector, leftLineFlowOut, rightLineFlowOut, mainFlow);
    system pump enginePump(enginePumpOn, mainFlow), electricPump(electricPumpOn, mainFlow);
    system line mainLine(mainFlow, mainFlowOut);
    system sensor mainSensor(mainMassOut, mainFlowOut, mainSensorMassOut, mainSensorFlowOut);
}
```

For this particular component nominal (healthy) behavior is defined as having zero flow in case of an empty tank, i.e., quantity is zero, and the tank flow being equal to the flow in the tank line. This (weak) model would already be sufficient for diagnosis but for better diagnostic results we choose to extend it with an explicit failure mode. For a tank an obvious failure mode is a leak, in which case the line flow is less than the tank flow as some of the fuel also escapes through the leak. This is implemented with the health mode `hContain.leak`.

The behavior of the other components is defined in a similar fashion. All components are used in the top-level system model `fuelSystem`. This is where the componenents are instantiated and the model topology is created by sharing the variables between components. The main characteristic that makes diagnosis a non-trivial problem is limited observability. Not all physical system variables are observable. For this particular system the fuel mass and flow are only observable at sensor locations. Observable variables are identified with the attribute `observable`.

Besides the sensor values[2], the control variables are observable by definition. For this model there are three control variables. One controls the valve. It can be used to select which, if any, tank is used. This is implemented with the `selectorPosition` type. Two control variables are used to control both pumps, one of which is driven by the aircraft's engine the other by a seperate electrical engine.

We have considered two failure scenarios. First the left tank is selected and the left sensor indicates a high flow, while other observations are nominal. As shown below, Lydia infers that, based on the model and these observations, the most likely root causes are either a leak in the tank or a leak in the left line. Other (less likely) single faults are unknown faults in the tank, fuel line, or valve. These are followed by multiple failures that are even less likely (not shown). In a second scenario (also not shown), we keep the left tank selected and now the right sensor indicates a low flow, while other observations are nominal. Lydia now places the root cause is on the right side of the aircraft, either in the tank or the line.

Below, we show the results from the first experiment in the transcript of the Lydia sdNNF solver (for details on its workings cf. Section 3). We can see the assigned values to all observable variables and the six leading diagnoses. Note, that with weak-fault models the number of diagnoses is exponential to the number of non-failed components in the minimal-cardinality fault as each healthy component can be faulty, but still producing nominal output. Hence, in the scenario below

[2]In the real system, the main sensor is a pressure sensor and only measures flow. Hence the use of mass dummy variables.

the Lydia solver asserts the faulty components (in our case a leaking left fuel tank) and does not commit on the state of the remaining variables.

```
@ start output <state>
observable enum leftSensorMass = nom
observable enum leftSensorFlow = high
observable enum rightSensorMass = nom
observable enum rightSensorFlow = zero
observable enum selector = left
observable enum enginePumpOn = true
observable enum electricPumpOn = false
observable enum mainSensorFlowOut = nom
@ stop output <state>                                                                          10
@ start output <fm>
(7.35542e−05) leftTank.h = leak, leftSensor.h = nom, leftLine.h = nom, valve.h = nom, ...
(7.35542e−05) leftTank.h = nom, leftSensor.h = nom, leftLine.h = leak, valve.h = nom, ...
(8.10724e−06) leftTank.h = nom, leftSensor.h = nom, leftLine.h = nom, valve.h = unknown, ...
(8.10724e−06) leftTank.h = nom, leftSensor.h = nom, leftLine.h = unknown, valve.h = nom, ...
(8.10724e−06) leftTank.h = nom, leftSensor.h = unknown, leftLine.h = nom, valve.h = nom, ...
(8.10724e−06) leftTank.h = unknown, leftSensor.h = nom, leftLine.h = nom, valve.h = nom, ...
...
@ stop output <fm>
```

From the above output, a Lydia user can derive the health of the system. In reality, Lydia is supplied with well-defined API for inclusion in higher-level frameworks, where diagnosis is a part of a *reactive loop* of fault monitoring, isolation, reconfiguration and recovery. In the next section we will look more closely on the computational aspects of Lydia.

3 Computing Diagnosis with Lydia

In Section 2 we have built a Lydia model. Our next goal is to provide an algorithm for efficient, sound and complete MBD, a core part of the Lydia toolkit. We will discuss how a model description *SD* and an *observation OBS* (the latter is simply a run-time valuation over a set of variables designated as observable) allow us to compute *diagnosis* of our system. To facilitate the presentation we will formalize the definitions of system and diagnosis.

Definition 1 (System). A diagnostic problem DP is the ordered triple $DP = \langle SD, COMPS, OBS \rangle$, where SD is a set of propositional sentences describing the behavior of the system, $COMPS$ is a set of components, contained in the system, and OBS is a term stating an observation over some set of "measurable" variables in SD.

In this approach for each component $c \in COMPS$ there is a corresponding propositional variable h_c representing its health state. We will call these variables h_c *health variables* and every instantiation of $\bigwedge_{c \in COMPS} h_c$ a *health state*.

Definition 2 (Diagnosis). A *diagnosis*[3] for the system $DP = \langle SD, COMPS, OBS \rangle$ is a set $D \subseteq COMPS$ such that $SD \wedge OBS \wedge \left[\bigwedge_{c \in D} \neg h_c\right] \wedge \left[\bigwedge_{c \in (COMPS \setminus D)} h_c\right] \not\models \perp$.

A diagnosis D is a *minimal* if no other diagnosis D', such that $D' \subset D$, exists. A *partial diagnosis* P is such a conjunction of health literals h_c or $\neg h_c$, $c \in COMPS$, that for every other conjunction ϕ which contains P it follows that $SD \wedge OBS \wedge \phi \not\models \perp$. Similarly, a *kernel diagnosis* is a partial diagnosis which is not contained in any other partial diagnosis [6].

From the above two definitions, it is visible that in order to perform diagnosis on a Lydia model it is enough to translate it to a well-formed propositional formula (**Wff**) and to use an entailment mechanism for finding those diagnoses D which are consistent with $SD \wedge OBS$, that is D explain $SD \wedge OBS$. The Lydia language is designed in such a way as to facilitate a conversion to a propositional **Wff** in polynomial time. This includes the normal language parsing, type-checking (Lydia is a type-strict language), expanding arrays and array quantifiers and processing of variable attributes.

[3]Throughout this paper we consider consistency-based diagnosis as opposed to abductive diagnosis.

The LYDIA language supports variables both in the Boolean and finite integer (FI) domains. The LYDIA toolkit proposes two approaches for unifying this – encoding FI variables as Booleans and vice-versa. Encoding FI into Boolean is trivial and we will not discuss it for brevity. Working directly in the FI domain is a preferred option and below we introduce a multivalued representation very-close to the traditional Boolean one and suitable to conventional propositional algorithms (e.g., DPLL). The definitions explaining the use of multi-value logic in LYDIA follow.

Definition 3 (Multi-Valued Literal). A multi-valued variable $v_i \in V$ takes a value from a finite domain, which is an integer set $D_i = \{1, 2, \ldots, m\}$. A positive multi-valued literal l_j^+ is a Boolean function $l_j^+ \equiv (v_i = d_k)$, where $v_i \in V, d_k \in D_i$.

Similarly, we introduce negative multi-valued literals. If not specified, a literal l_j can be either positive or negative.

Definition 4 (Multi-Valued Propositional Wff). A multi-valued propositional **Wff** is a formula over the multi-valued literals $l_1, l_2, \ldots, l_n$, and the standard Boolean connectives $\neg, \Leftrightarrow, \Rightarrow, \wedge, \vee$.

Up until now, we have discussed the compilation of the original LYDIA model to a Boolean or multi-valued **Wff**. Actually, this is the conjunction of each system's **Wff** as the actual hierarchy is preserved in this representation. This introduces the notion of *hierarchical system*, which will allow us to perform faster reasoning compared to algorithms working on "flat" representations[4] [10].

Definition 5 (Hierarchical System). A hierarchical system is a rooted, edge-labeled, acyclic multidigraph $H = \langle V, \rho, E \rangle$, where every node V_i, $V_i \in V$, contains a knowledge base SD_i and a set of components $COMPS_i$. The multidigraph is such that $COMPS_1 \cap COMPS_2 \cap \ldots \cap COMPS_n = \emptyset$. The root node is marked by ρ and the labels of the edges in E are maps $f : SD_i \rightarrow SD_j$ between the literals in the knowledge bases represented by the nodes V_i and V_j.

The introduction of hierarchical systems allows us to similarly define hierarchical CNF and hierarchical DNF (the latter is not DNF anymore but is a restricted form of Negation Normal Form). We call this hierarchical DNF semi-decomposable Negation Normal Form (sdNNF). A hierarchical system is simply a conjunction of Boolean or multi-valued **Wff**. It is possible to discard this information (i.e., to *flatten out* the hierarchy) and to continue transforming the **Wff** in its flat representation. A map showing the possible translations between a number of formats is shown in Figure 2.

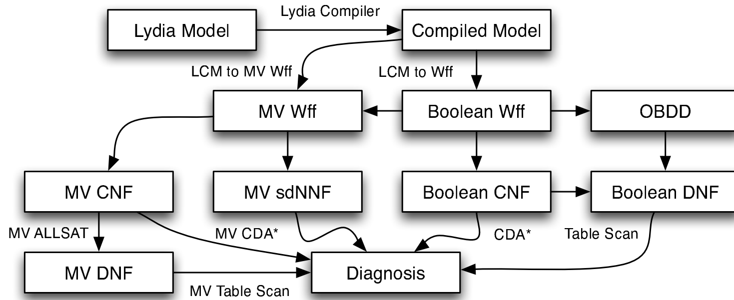

Figure 2: To compute diagnosis LYDIA translates the original model into a number of possible representation.

Instead of flattening a hierarchical system we will *selectively apply compilations on subsystems* of *SD*. The need to exploit hierarchy stems from the inherent high-computational price of MBR. By exploiting the hierarchical information and selectively compiling parts of the model it is possible to increase the diagnostic performance and to trade cheaper preprocessing time for faster run-time reasoning. Our hierarchical algorithm, being sound and complete, allows large models to be diagnosed, where compile-time investment directly translates to run-time speedup.

[4]For brevity, we refer to the classical diagnosis approach as "flat", i.e., non-hierarchical.

Furthermore LYDIA *repartitions and coarsens* the original model in an attempt to minimize the subsystem connectivity, a process which leads to faster run-time fault diagnosis at the price of some pre-processing time. This is explained in the two algorithms below which show an advanced way for mixing compilation and hierarchical reasoning for fast diagnosis. The implementation of the two algorithms below is an important part of the LYDIA reasoning tool-kit.

The reason for the compilation map in Figure 2 is threefold. First, we need to reach logically equivalent representation which allows us to cross-validate the correctness of the LYDIA tools. The implementation of other state-of-the-art techniques (like CDA* [18]) allows us to verify the final diagnostic result and to compare the diagnostic performance under fair conditions. Finally, and most importantly, almost any translation causes combinatorial explosion with some models.

Models or submodels, depending on different characteristics (e.g., weak/strong modeling, etc.) can produce very different compilation results. More precisely, there are Boolean functions which have linear OBDD representation, but exponential irreducible CNF. Therefore it is beneficial to choose different representations for different models and parts of models. Full complexity analysis of all representations is impossible due to lack of space in this paper but is available in [4].

Algorithm 1 Compilation of a LYDIA Model to sdNNF.

```
 1: function sdNNFFromLydiaModel(L, K, M)
        inputs: L, a Lydia model
                K, number of partitions, integer
                M, maximum DNF size, integer
        local variables: G, interaction graph
                         P, graph partitioning a set of set of node indices
                         N = ψ1 ∧ ψ2 ∧ ... ∧ ψn, an NNF, a conjunction of Wff
                         W, sdNNF, initially ∅
 2:     N ← LydiaCompile(L)              ▷ Compilation, basic rewritings, etc.
 3:     G ← InteractionGraph(N)   ▷ A node for ψi, and an edge for ψj, ψk sharing a variable.
 4:     P ← PartitionGraph(G, K)         ▷ Use, e.g., an approximate graph partitioning.
 5:     for all p ∈ P do
 6:         W ← W∧OBDDtoDNF(NNFtoOBDD(⋀_{i∈p} ψi))
 7:     end for
 8:     while ∃γ, δ ∈ W : CountSolutions(γ, δ) < M do
 9:         sdNNFNodesMerge(W, γ, δ)     ▷ Perform symbolic multiplication.
10:     end while
11:     return W
12: end function
```

Algorithm 1 shows the compilation of a LYDIA model to sdNNF. First it converts the model L to a hierarchical multi-valued NNF. Let the NNF N be a conjunction of **Wff** $N = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_n$. Then we build an interaction graph G in a fashion similar to [8] by having a node for each expression ψ_i in N and an edge in G if two expressions share a variable. We also weigh the edges of G with the number of variables two **Wff** ψ_i and ψ_j share.

Note that PARTITIONGRAPH is, itself, a computationally expensive process (it can be exponentially hard of the number of nodes in the graph G). LYDIA uses an approximation algorithm for graph partitioning [13] to solve the last problem. After the partitioning phase, we use an approximate model counter to merge nodes in the hierarchical description until any future merging would increase the number of terms in a node to a value exceeding a parameter M.

Next, we proceed with the run-time part, which is based on A*. We assume that components failures are independent and use the *a priori* probability of a fault term to guide a heuristic search for the most likely diagnosis. We assign the same small probability to all the components [5] as the reasoning technique is not probability-driven and it is possible to use other heuristics with similar results (e.g., the cardinality of a fault-mode).

Algorithm 2 computes diagnoses from the model produced by Algorithm 1 and *OBS*. The difference between the standard A* algorithm used in diagnosis, and this hierarchical version is that we try to find a conjunction of terms as opposed to conjunction of assumable variables consistent with *OBS*. The granularity of our approach is coarser and adjustable due to the parameterization of Algorithm 1 which allows trading space for time and in some cases reducing the overall computational complexity.

Algorithm 2 A* search in sdNNF.

```
1: procedure HierarchicalDiagnosis(H, OBS)
     inputs: H, root sdNNF node, each node is a disjunction of terms
             OBS, an observation
     local variables: Q, priority queue
                      s, c, terms
2:   Push(Q, InitialState(H))                    ▷ Push the empty term on the queue.
3:   while (c ←Pop(Q)) ≠ ∅ do        ▷ Until all possible conjunction of terms are expanded.
4:     EnqueueSiblings(Q, c)   ▷ Push all paths from the root to terms in the same node as c.
5:     if Diagnosis(c ∧ OBS) then          ▷ True iff c contains a term from each node in H.
6:       OutputDiagnosis(c)
7:     else
8:       if (s ←NextBestState(H, c)) ⊭⊥ then
9:         Push(Q, s)                     ▷ Choose the best state from the descendant of c.
10:      end if
11:    end if
12:  end while
13: end procedure
```

In this particular example Algorithm uses sdNNF but any hierarchical form with nodes consisting of tractable knowledge-bases will achieve similar results. In the main loop Algorithm 2 algorithm is chosen such a term c from the hierarchical node such that some heuristic estimate $f(c)$ is maximized. When a consistent conjunction of terms is found from all the nodes in the hierarchy, OutputDiagnosis is invoked to send the result to the user.

The auxiliary functions Push and Pop perform the respective priority queue manipulation on Q (Pop returns $\emptyset$ if the queue is empty). The initial state in the search tree, returned by InitialState, is the empty term. The selection of the next candidate states to be added to the search queue is done by the functions NextBestState and EnqueueSiblings. The former chooses the child state of the current state c and uses this term s from it, which again maximizes a utility function $f(s)$.

Algorithm 2 produces a diagnosis which completes the main goal of our exposition – to show a way for fast model-based fault diagnosis from modeling to the workings of a contemporary diagnostic search. Next we show some empirical result on the use of Lydia.

4 Experimental Results

As the use of Lydia in diagnosing models has already been demonstrated in Section 2, in this section we present experimental data on Lydia's inference performance.

We have derived a diagnosis benchmark from the ISCAS-85 set of combinatorial circuits [1]. Its basic characteristics are shown in Table 1. We have counted the number of variables $|SD|$ and the number of clauses in the CNF $|\Delta|$. The number of observable variables is denoted as $|OBS|$. The ISCAS-85 specification does not provide for a fault-modeling, hence we use our own standard logic component libraries allowing every gate-level component to fail. Throughout this paper we have used weak-fault models of the components.

In Table 1 we can see the compilation times for converting hierarchical CNF to hierarchical DNF and the time necessary for partial flattening. This time is denoted as T_c. The sum of the terms in each of the nodes of the hierarchical DNF is denoted as $|\phi_t|$. Note, that for the sdNNF we don't have a partial flattening step, hence the compilation time is only the time for converting the hierarchical CNF to hierarchical DNF. The time for finding a leading single-fault diagnosis using Algorithm 2 is denoted as T_d. For a reference we have included the time for computing the same diagnosis using a flat A* solver and the results are in the T_f column.

By using Algorithm 1 for partial coarsening the sdNNF, we gain speedup by a factor varying from 2.9 to 12.3 (cf. Table 1) in comparison to the original uncoarsened sdNNF. For this improvement in speed we pay the price of increasing the representation size in comparison to the original sdNNF [12]. This increase in size is a factor of 5.9 – 49.7.

	Gates	$\|SD\|$	$\|\Delta\|$	$\|OBS\|$	T_c [s]	$\|\phi_t\|$	T_d [ms]	T_f [ms]
c432	146	328	486	43	1.48	10 579	10	200.32
c499	202	445	714	73	69.67	53 653	3.84	132.75
c1908	252	541	911	58	28.88	28 544	4.75	57.44
c880	383	826	1 112	86	2.28	29 305	4.51	373.23
c1355	514	1 069	1 546	73	68.57	43 677	16.36	793.2
c2670	983	2 153	2 856	226	3.43	36 817	111.47	12 676.65
c3540	1 297	2 685	3 861	72	7.27	82 808	187.32	11 038.84
c5315	2 202	4 796	6 983	295	6.13	73 576	2 796.49	53 387.5
c6288	2 416	4 864	7 216	64	4.07	74 014	123.38	67 506.23
c7552	3 024	6 390	9 085	325	93.11	213 382	746.95	133 234.74

Table 1: Compilation time, resulting sdNNF size and first single fault diagnosis search time for the ISCAS-85 benchmark suite models.

While model partitioning is a topic on its own, even these preliminary results suggest the existence of an optimal space/time trade-off which we intend to exploit in subsequent research. Experiments with hand-prepared hierarchies [11] show speedup growing faster than the model size and in the range of $10^2 - 10^5$.

5 Conclusion and Future Work

In this paper we presented the Lydia language for model-based fault reasoning. We have illustrated the process of automatically computing a diagnosis by building a model of a fuel-system of a light aircraft. An example of diagnostic queries allowed the system to automatically compute the health state of the system. To alleviate the computational complexity of the diagnosis computation we have used algorithms implemented in the Lydia toolkit. These algorithms allow compilation of the model to a form which facilitates faster automated reasoning.

We have shown a map of tools for model manipulation. Many ways for computing diagnosis are possible and this is due to the fact that depending on the model and the observation the solution landscape can expose very different search properties. The basic principles on which the Lydia toolkit works are discussed in this paper and the detailed algorithms can be traced in the cited literature.

Automatic model partitioning, discovering of patterns in the model for faster reasoning, and more efficient compilation techniques are part of our plans future work. The current implementation of Lydia allows reasoning over combinatorial models. Introducing time and state would increase the applicability of the technique while it will impose more challenges in comparing the algorithms with alternative techniques as the model-based community lacks established standards for representing dynamic systems. Studying the workings of model-based diagnosis techniques on a bigger set of benchmarks, including real-world problems would improve the reader in the benefits of automated reasoning.

Acknowledgments

We extend our gratitude to the anonymous reviewers for their insightful feedback.

References

[1] Franc Brglez and Hideo Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran. In *Proc. ISCAS'85*, pages 695–698, 1985.

[2] Adnan Darwiche. Model-based diagnosis using structured system descriptions. *JAIR*, 8:165–222, 1998.

[3] Adnan Darwiche. New advances in compiling CNF into DNNF. In *Proc. ECAI'04*, pages 328–332, 2004.

[4] Adnan Darwiche and Pierre Marquis. A knowlege compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[5] Johan de Kleer. Using crude probability estimates to guide diagnosis. *AI*, 45(3):381–291, 1990.

[6] Johan de Kleer, Alan Mackworth, and Raymond Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56(2–3):197–222, 1992.

[7] Johan de Kleer and Brian Williams. Diagnosing multiple faults. *JAI*, 32(1):97–130, 1987.

[8] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., 2003.

[9] Yousri El Fattah and Rina Dechter. Diagnosing tree-decomposable circuits. In *IJCAI'95*, pages 1742–1749, 1995.

[10] Alexander Feldman and Arjan van Gemund. A two-step hierarchical algorithm for model-based diagnosis. In *Proc. AAAI'06*.

[11] Alexander Feldman, Arjan van Gemund, and André Bos. A hybrid approach to hierarchical fault diagnosis. In *Proc. DX'05*, pages 101–106, 2005.

[12] Mark Hansen, Hakan Yalcin, and John Hayes. Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. *IEEE Design & Test*, 16(3):72–80, 1999.

[13] Burkhard Monien and Stefan Schamberger. Graph partitioning with the party library: Helpful-sets in practice. *SBAC-PAD*, pages 1550–1533, 2004.

[14] Nicola Muscettola, P. Pandurang Nayak, Barney Pell, and Brian C. Williams. Remote agent: To boldly go where no AI system has gone before. *AI*, 103(1-2):5–47, 1998.

[15] Barney Pell, Douglas E. Bernard, Steve A. Chien, Erann Gat, Nicola Muscettola, P. Pandurang Nayak, Michael D. Wagner, and Brian C. Williams. An autonomous spacecraft agent prototype. In *Proc. AGENTS'97*, pages 253–261, 1997.

[16] Gregory Provan. Hierarchical model-based diagnosis. In *Proc. DX'01*, 2001.

[17] Farrokh Vatan. The complexity of the diagnosis problem. Technical Report NPO-30315, Jet Propulsion Laboratory, California Institute of Technology, 2002.

[18] Brian Williams and Robert Ragno. Conflict-directed A^* and its role in model-based embedded systems. *Journal of Discrete Applied Mathematics*, 2004.