# The Diagnostic Challenge Competition: Probabilistic Techniques for Fault Diagnosis in Electrical Power Systems

**Brian W. Ricks*, **. Ole J. Mengshoel*****

*University of Texas at Dallas, Dallas, TX 75080 USA
**USRP, NASA Ames Research Center, Moffett Field,
CA 80523 USA (e-mail: bwr031000@utdallas.edu).
***Carnegie Mellon University, NASA Ames Research Center,
Moffett Field, CA 80523 USA
(e-mail: Ole.J.Mengshoel@nasa.gov).

**Abstract:** Reliable systems health management is an important research area of NASA. A health management system that can accurately and quickly diagnose faults in various on-board systems of a vehicle will play a key role in the success of current and future NASA missions. We introduce in this paper the ProDiagnose algorithm, a diagnostic algorithm that uses a probabilistic approach, accomplished with Bayesian Network models compiled to Arithmetic Circuits, to diagnose these systems. We describe the ProDiagnose algorithm, how it works, and the probabilistic models involved. We show by experimentation on two Electrical Power Systems based on the ADAPT testbed, used in the Diagnostic Challenge Competition (DX 09), that ProDiagnose can produce results with over 96% accuracy and < 1 second mean diagnostic time.

## 1. INTRODUCTION

From physical electrical systems to computer networks, the need to quickly and accurately diagnose faults in a system is an important part of the puzzle of keeping systems healthy and operating smoothly.

Here are two examples of shortfalls in systems health management. They showcase just a couple of the vastly many negative outcomes that can arise with systems that have inadequate or no health management system.

In December of 1999, the Mars Polar Lander, a NASA Mars exploration vehicle, descended into the Martian atmosphere, never to be heard from again. The leading theory on the loss states possible misinterpretation of sensor noise received by the lander's on-board software. It is believed that the descent engines were shut down during leg deployment, while the lander was still about 40 meters above the surface, causing it to crash. Had the Mars Polar Lander been equipped with a better health management system that had not generated these false positives, it is very possible that the lander would have been able to carry out its mission.

In December of 2004, the F-22 Raptor suffered its first crash, after a brief interruption in the electrical power system on board caused sensors that monitor the plane's pitch, roll and yaw to stop working. The pilot did not know this until right at take-off, and by that time it was too late. A health management system on board could have detected the fault and taken corrective action by alerting the pilot of the issue before take-off.

In this paper, we discuss the *ProDiagnose* algorithm, which is designed to accurately and quickly diagnose faults such as the ones mentioned in the two examples above. ProDiagnose diagnoses different types of faults for sensors and components. It can determine if a sensor is stuck or offset. It can also determine if a component has failed, or if a component is operating in a mode that it is not supposed to be in.

ProDiagnose processes all incoming environment data *(observations* from a system being diagnosed), and acts as a gateway to a probabilistic inference engine. The inference engine analyzes the observations given to it by ProDiagnose, and computes diagnoses. ProDiagnose uses the Arithmetic Circuit Evaluator, or ACE. ACE uses *arithmetic circuits* (*AC*s), which are compiled from Bayesian network models (Chavira & Darwiche 2007; Darwiche 2003). The primary advantage to using ACs is speed, which is key in resource-bounded systems such as aircraft and spacecraft (Mengshoel 2007).

To demonstrate ProDiagnose in action, we have developed two probabilistic models of *Electrical Power Systems* (EPS) for diagnosis. Both of these models were based on the ADAPT testbed (Poll et al. 2007) at NASA Ames Research Center <http://ti.arc.nasa.gov/project/adapt-diagnostics/>, a physical EPS that behaves similar to EPSs found on board NASA spacecraft. These probabilistic models are discrete and static *Bayesian networks.* The ADAPT testbed also was used in the *DX 09* Diagnostic Challenge Competition <http://www.dx-competition.org/>, in which ProDiagnose competed and achieved the highest scores.

In this paper, we describe the ProDiagnose Algorithm, and DX 09 Competition results of ProDiagnose. We also describe each probabilistic model of ADAPT in depth.

## 2. OVERVIEW

ProDiagnose uses a probabilistic modeling system for diagnosis, called a Bayesian network, or belief network (Lauritzen & Spiegelhalter 1988; Pearl 1988). A Bayesian network is a directed acyclic graph (DAG), combined with an associated set of conditional probability tables (CPTs). Each vertex of the graph represents a *discrete random variable*.

Each random variable has a CPT of size that is dependent on the number of parent vertices, and the number of discrete *states* that these vertices contain. The directed edges typically represent the causal dependencies between variables. By denoting, or *clamping,* as evidence specific observations (to a state) with 100% probability for certain random variables, it is possible to compute the marginal probability of all other vertices in the graph very quickly. The marginal probabilities can then be used to diagnose the system itself.

Arithmetic circuits are a fast way to evaluate Bayesian networks. An arithmetic circuit derives marginal probabilities by addition and multiplication operations (Chavira & Darwiche 2007; Darwiche 2003). During each ProDiagnose call to ACE, the partial derivatives of this AC are computed with respect to each discrete random variable. ProDiagnose queries the arithmetic circuit to return the marginal probabilities in constant time.

### *2.1 Notations and Definitions*

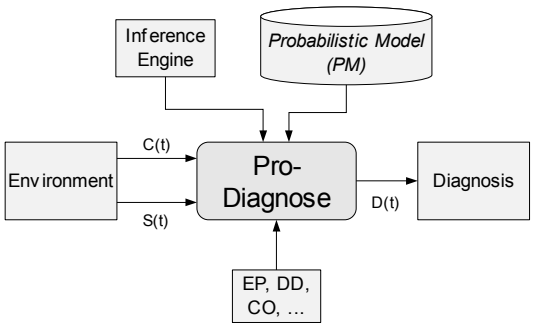


Figure 1: The ProDiagnose Architecture. Two types of diagnosis-related messages can be received, commands, C(t) and sensor readings, S(t) (or sensor data). Commands can be received any time, whereas sensor data comes in at specific times, according to the sample cycle. Diagnosis, D(t), is sent after each sample cycle completes.

Before discussing the algorithms, we introduce notation and definitions, see also Figure 1.

*PM* (Probabilistic Model): The probabilistic model represents the system that ProDiagnose will diagnose. The probabilistic model that ProDiagnose uses is an Arithmetic Circuit compiled from a Bayesian Network.

*e* (evidence): *e* represent the evidence that is used in the diagnosis process. Evidence comes from commands and sensor readings.

A random variable in the network is referred to as a node, and a group of nodes forms a component, which represents a physical object that we are modeling. Each node in the network is described as follows:

**C** (Command Set): A Command node $C \in \mathbf{C}$ represents a command given to a component. A command is clamped to the node as evidence.

**D** (Delta Set): A Delta node $D \in \mathbf{D}$ represents the difference (delta) between the current sensor reading $S$(t) and its previous reading $S$(t - 1). Its value represents either a negative delta, zero (no) delta, or a positive delta. Note that **D** is not the same as D(t) in Figure 1.

**A** (Attribute): An Attribute $A \in \mathbf{A}$ represents a subset of nodes that describe various attributes of a component. These attributes could be voltage $V$ and current $I$ for an electrical device. $A$ usually depends on $A' \in \mathbf{A}$ upstream, where $A' \neq A$.

**CL** (Closed): A Closed node $CL \in \mathbf{CL}$ represents a generalized state of operation for the component.

**S** (Sensor Set): A Sensor node $S \in \mathbf{S}$ represents the current reading of a sensor. This reading is a discretized real value, which represents a range for real-valued sensors, or the actual state of 0 or 1 for a boolean position sensor. The discretized sensor reading is clamped as evidence in the network.

**ST** (Stuck Set): A Stuck node $ST \in \mathbf{ST}$ represents the stuck state of a sensor. A sensor becomes stuck when its reading is the same over a period of time, regardless of what the underlying process state is.

**H** (Health Set): A Health node $H \in \mathbf{H}$ represents the current health state of a component. The set of states of a node $H$ is partitioned into normal and abnormal states. Abnormal states indicate a fault in the component.

**CH** *(*Change Set): A Change node $CH \in \mathbf{CH}$ represents overall trends in sensor readings. They are good for detecting small changes in sensor readings over a period of time. This change is clamped as evidence, but it also depends on $H$, as certain states of health for relevant components can play a role in how the change nodes affect the rest of the network.

*Base_Component*: A *base_component* of a node represents its physical component or device in the real world. base_components are used as a common link for lookups of various nodes that all share a *base_component*. For example, a physical sensor will have an $H$ and $S$ node associated with it, and may have $D$ and $ST$ nodes also.

The following is a list of all ProDiagnose parameters and their purpose:

*Sample Cycle, SC*: The amount of time, measured in milliseconds, between sample readings.

*Command Epsilon, EP*: A global threshold for determining if a given command should be clamped as evidence immediately or queued in regard to the time stamp of the last sensor set. This is discussed in more detail in the Command Data section.

*Diagnosis Delay, DD*: A global value, measured in sample cycles, that gives the delay to start diagnosis output. Diagnosis delay is used at the beginning of environment monitoring. This variable is useful to filter out transients and other normal behavior that may appear abnormal and thus have false positive diagnoses associated with them.

*Command Offset*, *CO*: A global value, measured in sample cycles, that gives the delay to output diagnosis when a command is received. This variable is useful for situations in which n sample cycles after a command have some transients for sensors that are slower to update than others. For these situations, false diagnosis output will be generated regardless of whether the command is queued or not. For most sensors, $CO = 2$ is usually enough to prevent this kind of behavior.

*Sensor Stuck Delay, SSD*: A value, measured in sample cycles, that gives, for a sensor $S$ with a reading that is the same, a maximum number of sample cycles to wait before setting that sensor to a stuck state.

ProDiagnose is designed to handle two main types of faults: sensor faults and physical component faults. Each type has a

set of faults, depending on the probabilistic model being diagnosed.

## 3. PRODIAGNOSE ALGORITHM

The ProDiagnose algorithm can be broken down into two stages: The pre-processing and diagnosing stages. The pre-processing stage initializes ProDiagnose to a state in which it can start accepting data from an environment. The diagnosing stage analyzes each *message* **S**(t) or C(t) when they come in and outputs diagnosis of abnormal health *(H)* states according to the sample cycle.

### *3.1 Pre-Processing Stage*

```
1  Algorithm ProDiagnose(EP, DD, CO)
2  Begin:
3    initialize_DA(EP, DD, CO, Init_Params : PDB)
4
5    Send_Message(Message : M = DA_Ready)
6
7    do
8    Begin:
9      receive Message : M from environment
10
11     Process_Message(M, EP, DD, CO)
12   loop until M = Terminate
13 End
```

The pre-processing stage sets up ProDiagnose, including parameters and all data structures that will be used during diagnosing.

*H* nodes are used on the output side, and *C, D, S, ST, CH* are for input. On the input side, commands *(C)* and sensor *(S)* readings are given to ProDiagnose. *D* and *ST* node values are derived from their respective component's *S* node sensor reading (before discretization), and a *CH* node's value is derived from an *S* node assigned to it.

### *3.2 Diagnosis Stage*

The diagnosis stage is executed each time data from the environment is received. The first course of action is to determine the data type of the incoming message. ProDiagnose evaluates the *PM* and computes diagnoses only when sensor data is received.

```
1  Algorithm Process_Message(Message : M, EP, DD, CO)
2  Begin:
3    if M = Scenario_Status : Terminate then Exit
4
5    if M = C(t) : (Command : C_Command, Value : V)
6    Begin:
7      C ← get_node(C_Command)
8      tᵢ ← C.timestamp
9      tⱼ ← S(t - 1).timestamp + SC
10     if tⱼ -  tᵢ < EP
11       command_queue ← C
12     else
13       C.command ← Discretize_For_PM(V)
14   End if
15
16   if M = S(t)
17   Begin:
18     for each S(t) : (Sensor : S_Sens, Value : V) ∈ S(t)
19     Begin:
20       S ← get_node(S_Sens)
21       S.value ← Discretize_For_PM(V)
22
23       if D ∈ Base_Component(S)
24         D.value ← Discretize_For_PM(Calc_Delta(D))
25
26       if ST ∈ Base_Component(S)
27         ST.value ← Discretize_For_PM(Calc_Stuck(ST))
28     End for
29
30     for each CH
31       CH.value ← Calc_Change(CH)
32
33   End if
34
35   Calculate_Marginals(PM)
36
37   Output_Diagnosis(H, DD, CO)
38
39   Update_Command_Queue(command_queue)
40 End
```

*Scenario_Status* (Line 3): This datatype is a constant specifying any status updates that arrive to ProDiagnose as message M. If M is the constant specifying termination, then ProDiagnose frees up its resources and exits gracefully.

C(t) (Line 5): This datatype is a tuple, (*C_Command, V*), in which *C_Command* is a command given, and V is the value of the command. ProDiagnose first fetches the appropriate *C* node (line 7). It then checks the timestamp of the command. If the command $C(t_i)$ has come in too close to $\mathbf{S}(t_j)$, where $j > i$ and $t_j - t_i < EP$, then we queue the command (line 11). Otherwise we update the *C* node with the new command. ProDiagnose will queue commands to make sure that a command does not update before the sensor readings reflect the state change of the command. Not doing this can often result in false positives (usually the component that we are commanding coming back as stuck) due to the sensor readings not immediately reflecting how the new command affects the rest of the network. The worst case scenario is that the commanded component is believed to be healthy, which sets off many false positives of other components. Keeping these commands queued for one sample set usually prevents this from happening.

**S**(t) (Line 16): This datatype is a set, $\{(S\_Sens, V) \mid S\_Sens \in \mathbf{S}\}$, in which *S_Sens* is a sensor, and *V* is the value for the sensor. Each sample has a key/value pair for every sensor in the network. The keys map to an *S* node, and the values (*V*) represent the current sensor reading for the respective *S* node. For each *S* node, its new sensor reading is discretized (Line 21) and value updated to the new reading. During each iteration ProDiagnose also looks for any *D* or *ST* nodes that share the same *base_component* as the *S* node in the network. These operations consist of simple lookups using the *base_component* for the sensor.

If a *D* or *ST* node is found for a specific *base_component*, then its value is updated using the current sensor value (lines 24, 27). This value is further discretized for clamping as evidence in the network.

After all *S* nodes are processed, ProDiagnose updates the values of any *CH* nodes that may be present in the Bayesian network. Since *CH* nodes can be bound to any sensor in the Bayesian network, a reference to the bound *S* node is stored in the *CH* node. Because of this, we can iterate through the *CH* nodes after all *S* nodes are updated, as opposed to doing *CH* node lookups for each *S* node (though it is worth mentioning that *CH* nodes can be treated similar to *D* and *ST* nodes). The *CH* node's value is then updated using the bound *S* node's value as a base (and discretized like the rest of the nodes). At this point all our input nodes are ready for clamping to the network and evaluation of the network itself.

```
1  Algorithm Discretize_For_PM(Value : V, Thresholds : TH)
2  Begin:
```

```
3  A ← NEGATIVE_INFINITY
5
6    for each N ∈ TH
7    Begin:
8      B ← N
9      if V ≥ A and V < B
10       return TH.Index(N)
11     else
12     Begin:
13       A ← B
14     End else
15   End for
16
17   return TH.Index(TH.size + 1)
18 End
```

The Discretize_For_PM method takes the current sensor value and returns an index value that is used in network nodes as states (clamped evidence). This index is the index value between two *thresholds*. A threshold has **TH.**size + 1 different Index values (line 6) that are possible, starting at 0, where **TH.**size is defined as the number of thresholds N in the set **TH**. The discretized value is Index(N) for which V is [A, B) (lines 9, 10), or Index(**TH.**size + 1) if V is above all thresholds (line 17). For example, a sample sensor has three discrete states in the *PM*: low, mid and high, which correspond to index values 0, 1 and 2 respectively. Two sample thresholds are given: 50 and 100. Any sensor reading below 50 is given an index of 0, [50, 100) is given an index of 1, and above 100 is given index 2.

Now we describe the algorithms we have not discussed already, which we refer to as *dynamic processing* for the Bayesian network:

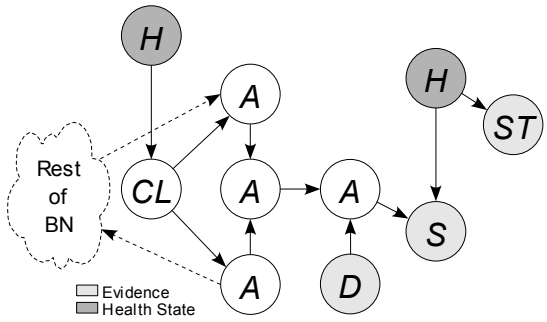


Figure 2: The Bayesian network representation of a fan or pump. Fans and pumps utilize delta *D* and stuck *ST* nodes. If the *base_component* is a sensor, then *H* depends on *S* and *ST*, as a faulty or stuck sensor would indicate an abnormal health state. If the component is a physical device, then *H* depends on *CL*, as a device that has malfunctioned would indicate an abnormal health state.

```
1  Algorithm Calc_Delta(D)
2  Begin:
3    I ← Sensor_Average(Base_Component(D).S)
4    I_prev ← Sensor_Average(Base_Component(ST).S(t-1))
5    D.value ← I − I_prev
6
7    return D
8  End
```

```
1  Algorithm Sensor_Average(S)
2  Begin:
3    Sum ← 0
4    P ← 0
5
6    for each S  ∈ {S(t), ..., S(t − p)}
7    Begin:
8      Sum ← Sum + S.value
9      P ← P + 1
10   End for
11
12   return A / P
```

The Calc_Delta method returns the difference between the current and previous averaged sensor values of the delta *D* node's *base_component* (line 3, 4: Calc_Delta, Figure 2). The average is defined as the summation of any contiguous subsequence of sensor readings (lines 6, 8: Sensor_Average) from the current *S*(t) sample cycle to *S*(t − *p),* divided by *p* (line 12: Sensor_Average), where *p* is defined as the position in the sample timeline.

```
1  Algorithm Calc_Stuck(ST, Counter : I, Sensitivity : K)
2  Begin:
3    current_value ←  Base_Component(ST).S.value
4    previous_value ← Base_Component(ST).S(t-1).value
5    J ← current_value − previous_value
6
7    if J = 0 and I ≥ K
8      return 0
9    else if J ≠ 0
10     I ← 0
11   else
12     I ← I + 1
13
14   return J
15 End
```

The Calc_Stuck method analyses a component's sensor values for readings that are repeatedly identical, defined if J = 0, by subtracting the current *S*(t) and previous *S*(t - 1) values of the *ST* nodes' *base_component* sensor (line 5, Figure 2). Each time J = 0 a counter I is incremented. If this pattern continues past a given *sensitivity threshold* K so I ≥ K (line 7), the *ST* node is considered stuck. The pattern is broken if J ≠ 0 during a sample cycle (line 5), at which point I is reset to 0 (line 9). A stuck node *ST* has three discretized states, 0, 1, and 2, where 1 represents stuck, and 0, 2 represent non-stuck states.

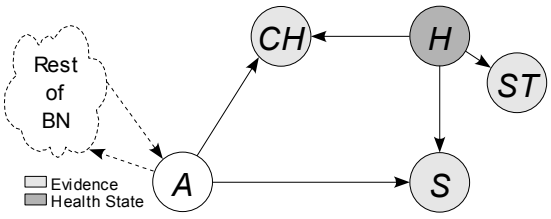


Figure 3 The Bayesian Network Representation of a bound sensor (source sensor) *S* to a change node *CH*. *CH* influences the Attribute node *A* in this figure, and depends on *H*.

```
1  Algorithm Calc_Change(CH, CUSUM : U)
2  Begin:
3    S ←  CH.Bound_Sensor
4    I ← Sensor_Average(S)
5    U_prev ← U
6    U ← (S.value − I) + U_prev
7
8    if U < CH.Lower_Threshold
9      return 0
10   else if U > CH.Upper_Threshold
11     return 2
12
13   return 1
14 End
```

The Calc_Change method calculates a continuous CUSUM, or cumulative sum, which is used to detect slight changes, or *trends,* in a sensor reading over time. The current CUSUM U is calculated by taking the current sensor reading *S* from the *CH* nodes' bound sensor (Figure 3) and subtracting it from an averaged sensor reading I (lines 4, 6), in the same way as for *D* nodes (see Sensor_Average algorithm). This difference is then added to the previous CUSUM, and updated as the new current CUSUM U (line 6). Very slight changes that form a trend will over time will cause the CUSUM to consistently

increase or decrease. If this change accumulates to the point where the CUSUM's value to drop below a lower threshold (line 8) or above an upper threshold (line 10), the index of the *CH* node will change in the *PM* to 0 or 2, respectively.

```
1  Algorithm Calculate_Marginals(PM)
2  Begin:
3    for each Node : N ∈ {S,C,D,ST,CH}
4      e ← fetch_current_evidence(PM, N)
5
6    for each H ∈ H
7      H.state ← argmax(P(H | E = e))
8
9    return H
10 End
```

In the Calculate_Marginals method, ProDiagnose clamps as evidence all of the input nodes (lines 3, 4). Our probabilistic models will always have *S* nodes, but not necessarily *C, D, ST,* or *CH* nodes. ProDiagnose then calculates the marginals, P(*H* | **E** = ***e***), for all **H** (lines 6, 7). The output from the inference engine gives the DA the states of **H**. For each $H \in$ **H**, ProDiagnose takes the most likely value for that node and assigns it as the new health state (line 7).

```
1  Algorithm Output_Diagnosis(H, DD, CO)
2  Begin:
3    Candidate Set : CS
4
5    if first execution of Algorithm
6      dd ← DD
7    if received C(t) within last sample cycle
8      co ← CO
9
10   if dd = 0 and co = 0
11   Begin:
12     for each H ∈ H
13     Begin:
14       if H.state = abnormal
15         CS ← H
16     End for
17   End if
18
19   if dd > 0
20     dd ← dd – 1
21   if co > 0
22     co ← co - 1
23
24   return CS
25 End
```

If the diagnosis delay has reached 0, *dd* = 0 (initially set during the first iteration of this algorithm), and there is no current command offset, *co* = 0 (line 10), ProDiagnose will output a four-tuple (*t*, CS, DS, IS) as D(*t*) (Figure 1) if any abnormal health states are detected. *t* is the current time, CS is a *candidate se*t, DS is a *boolean detection signal*, and IS is a *boolean isolation signal*. A candidate set CS is a set containing zero or more *candidates*. DS and IS are simply: DS = IS = (|CS| > 0). If CS is non-empty, we have CS = $\{C_1, ..., C_n\}$, where n ≥ 1, with each candidate C in CS consisting of two-tuples like this: C = $\{(H_1, a_1), ..., (H_m, a_m)\}$, for m ≥ 1. For ProDiagnose, a health node $H_i$ is included in a candidate C, along with a most likely state $a_i$, if and only if that state is abnormal. ProDiagnose always outputs exactly one ($H_i$, $a_i$) tuple per candidate, and thus candidate weights do not play a role (and have for simplicity been kept out of the discussion above). If *dd* > 0, then it decrements by 1 (line 20). This also happens with *co* > 0 (line 22). *co* will be set to its original value *CO* each time ProDiagnose receives a command within the last sample cycle of **S**(t).

```
1  Algorithm Update_Command_Queue(command_queue)
2  Begin:
3    for each C ∈ command_queue
4    Begin:
5      C ← get_node(C_Command)
6      tᵢ ← C.timestamp
7      tⱼ₊₁ ← S(t).timestamp + SC
8
9      if tⱼ₊₁ - tᵢ < EP
10       keep command in queue
11     else
12     Begin:
13       pop command from queue
14       C ← V
15     End else
16   End for
17 End
```

The last step taken by ProDiagnose after diagnosis output is updating the command queue, pulling any commands C($t_i$) whose timestamp is considered to be out of range of the next sample timestamp **S**($t_{j+1}$) according to the command epsilon, $t_{j+1} - t_i < EP$ (lines 8, 9).

## 4. MODELS

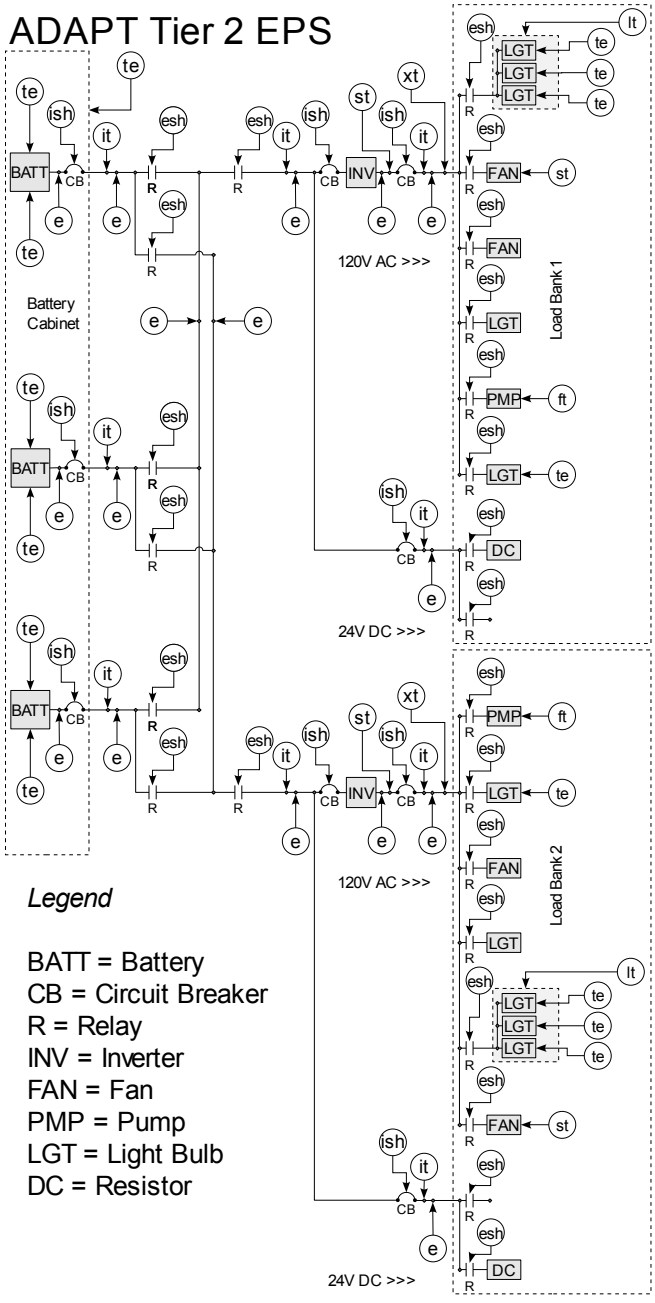


Figure 4: The ADAPT Tier 2 EPS. ADAPT Tier 1 is a subset of Tier 2.

### 4.1 ADAPT Tier 1

The ADAPT Tier 1 Bayesian network models a subset of the ADAPT testbed. This EPS consists of the inclusive path from the second (middle) battery, the bottom DC → AC inverter, and the bottom large fan in Load Bank 2 (Poll et al. 2007, see Figure 4). The Bayesian network model consists of 133 nodes, 149 edges, and a minimum and maximum domain cardinality of 2 and 6, respectively. A breakdown of node quantity for sensors is referenced in Table 1 below.

| ADAPT EPS | | | | | Bayesian Network | |
|---|---|---|---|---|---|---|
| | | | Quantity per EPS | | Quantity per Sensor | |
| Name | Symbol | Description | Tier 1 | Tier 2 | Nodes | Evidence Nodes |
| DC Current Sensor | it | Measures DC current in amps | 2 | 7 | 3 | 2 |
| AC Current Sensor | it | Measures AC current in amps | 1 | 2 | 3 | 2 |
| DC Voltage Sensor | e | Measures DC voltage in volts | 4 | 12 | 3 | 2 |
| AC Voltage Sensor | e | Measures AC voltage in volts | 2 | 4 | 3 | 2 |
| Circuit Breaker Position Sensor | ish | Senses whether a circuit breaker is opened or closed | 3 | 9 | 2 | 1 |
| Relay Position Sensor | esh | Senses whether a relay is opened or closed | 3 | 24 | 2 | 1 |
| Temperature Sensor | te | Measures temperature in Fahrenheit of batteries, battery cabinet, and light bulbs | 2 | 15 | 3 | 2 |
| Speed Transmitter | st | Measures RPM of the large fans | 1 | 2 | 5 | 3 |
| Phase Angle Transducer | xt | Measures the phase shift in degrees between the sine waves of AC current and voltage | 1 | 2 | 6 | 2 |
| AC Frequency Transmitter | st | Measures the AC frequency in Hertz | 1 | 2 | 3 | 2 |
| Flow Transmitter | ft | Measures the flow rate in gallons per hour through a pump | 0 | 2 | 5 | 3 |
| Light Sensor | lt | Measures the intensity in millivolts of incoming light | 0 | 2 | 3 | 2 |
| | | **TOTAL** | 20 | 83 | 41 | 24 |

Table 1: ADAPT EPS sensors, with their quantity in the ADAPT Tier 1 and Tier 2 EPS. Also shown is the number of nodes in the Bayesian network representation of the sensors, and which quantity of those nodes are evidence nodes.

### 4.2 ADAPT Tier 2

The ADAPT Tier 2 Bayesian network models the full ADAPT testbed (Figure 4). This Bayesian network represents an EPS that is similar to EPSs found aboard NASA spacecraft and aircraft (Mengshoel et al. 2008). ADAPT Tier 2 consists of 3 batteries connected in parallel through 2 DC → AC inverters to 2 load banks (Poll et al. 2007, see Figure 4). The Bayesian network model consists of 601 nodes, 681 edges, a minimum domain cardinality of 2, and a maximum domain cardinality of 6. Reference Table 1 for a breakdown of node quantity for sensors.

### 4.3 Bayesian Network Representation

ProDiagnose currently employs two different *static* Bayesian network models, corresponding to ADAPT Tier 1 and Tier 2, respectively. Both Bayesian networks have two types of *parts*: components and sensors. A component models a physical device in the EPS, such as a fan, circuit breaker, relay, or light bulb. A sensor models a physical sensor in the EPS. Sensors can take measurements of components or *wires*. ADAPT *e* and *it* sensors are wire sensors. Figure 2 models a physical component (left side) and its sensor (right side). Figure 3 models a wire sensor (though most wire sensors do not have *CH* nodes associated with them). These structures are combined with attribute *A* nodes to form a complete Bayesian Network model of the EPS.

Associated with each node in a Bayesian network model is a *Conditional Probability Table* (CPT). The CPT gives the conditional probability that a specific node will be in a specific state given the state values of its parent nodes.

| *H* | |
|---|---|
| **healthy** | 0.85 |
| **offsetToZero** | 0.02 |
| **offsetToLow, offsetToMid, or offsetToHigh** | 0.04 |
| **stuck** | 0.01 |

Table 2: The CPT for a health node *H*. This CPT represents the health of a fan sensor.

| *S* | | | | | |
|---|---|---|---|---|---|
| *H* | *A* | **zero** | **low** | **mid** | **high** |
| **healthy** | **zero** | 0.997 | 0.001 | 0.001 | 0.001 |
| | **low** | 0.001 | 0.997 | 0.001 | 0.001 |
| | **mid** | 0.001 | 0.001 | 0.997 | 0.001 |
| | **high** | 0.001 | 0.001 | 0.001 | 0.997 |
| **offsetToZero** | **zero, low, mid, or high** | 0.997 | 0.001 | 0.001 | 0.001 |
| **offsetToLow** | **zero, low, mid, or high** | 0.001 | 0.997 | 0.001 | 0.001 |
| **offsetToMid** | **zero, low, mid, or high** | 0.001 | 0.001 | 0.997 | 0.001 |
| **offsetToHigh** | **zero, low, mid, or high** | 0.001 | 0.001 | 0.001 | 0.997 |
| **stuck** | **zero, low, mid, or high** | 0.25 | 0.25 | 0.25 | 0.25 |

Table 3: The CPT for a sensor node *S*. This CPT represents a fan sensor. Sensor readings are after *discretization* clamped to *S* nodes.

A health node *H* gives the health state of a component or sensor in the Bayesian network. Most *H* nodes follow the same type of CPT pattern as Table 2. Sensor *S* nodes represent sensors, and are evidence nodes in the Bayesian network. Sensor readings are clamped to *S* nodes as evidence. Evidence nodes are the way in which ProDiagnose inputs information to the Bayesian network.

| *ST* | | | |
|---|---|---|---|
| *H* | **negDelta** | **zeroDelta** | **posDelta** |
| **healthy, offsetToZero, offsetToLow, offsetToMid, or offsetToHigh** | 0.499 | 0.002 | 0.499 |
| **stuck** | 0.001 | 0.998 | 0.001 |

Table 4: The CPT for a stuck node *ST*. Stuck nodes tend to have the same CPT pattern. This CPT represents the stuck state of a fan sensor.

Stuck nodes *ST* are used to make a stuck state more probable within the same part's health *H* node. The two *ST* states *negDelta* and *posDelta* refer to a negative or positive change, respectively, in the sensor *S* node's sensor reading. The *zeroDelta* state represents a stuck state. After the *SSD* has been reached, this state will be clamped in the *ST* node. When an *ST* node is clamped to *zeroDelta*, the *H* node's state

has a very high probability (99.8%, Table 4) of being stuck, and since the *ST* node is directly connected to it, it yields great influence over the most likely value of the *H* node (we have equal conditional probabilities for the stuck state in the *S* node, Table 3, to make sure that the *S* node itself cannot yield any considerable influence on the *H* node being stuck).

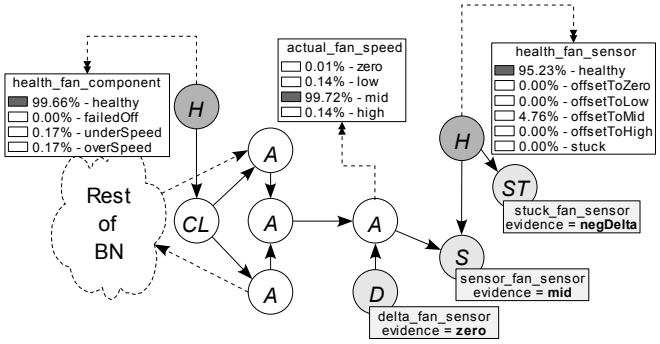


Figure 5: The marginal distributions for health *H* nodes health_fan_component and health_fan_sensor as well as the actual_fan_speed attribute *A* node (same representation as in Figure 2). The actual_fan_speed *A* node represents the actual state of the fan's blades.

In Figure 5 we see the most likely values for the health *H* nodes of a fan component and sensor, based on the evidence shown (Figure 5). The rest of the Bayesian network also influences these outcomes. Notice how the *actual_fan_speed A* node agrees with the evidence of the *S* node.

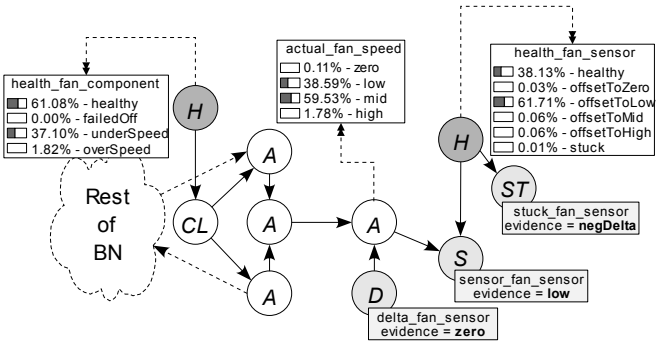


Figure 6: The marginal distributions for health *H* nodes health_fan_component and health_fan_sensor as well as the actual_fan_speed attribute *A* node, when the fan sensor's evidence (sensor reading - state) is changed to low.

Suppose now that the sensor readings for the same fan sensor dip downward so that the discretized state for the sensor *S* node is now *low* (Figure 6). Assuming the evidence clamped to the rest of the Bayesian network is the same as in Figure 5, we see that the most likely value for the sensor's health is now *offsetToLow*, based on the marginal distribution for that node (Figure 6). However, there is still enough evidence to suggest that the sensor's health could be *healthy*, but with a lower probability. Therefore, we say that the sensor's health is *offsetToLow*. A similar logic applies to the fan component's health state as being *healthy*.

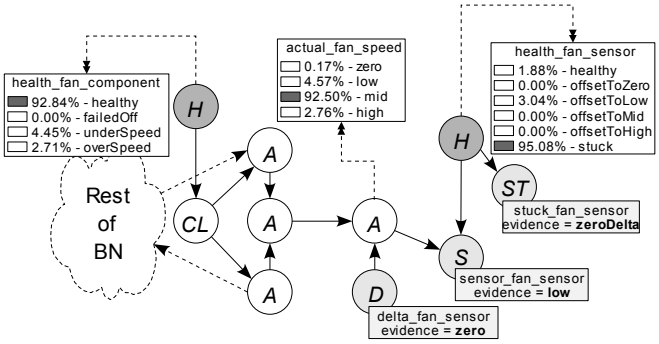


Figure 7: The marginal distributions for health *H* nodes health_fan_component and health_fan_sensor as well as the actual_fan_speed attribute *A* node, when the stuck *ST* node is clamped to the stuck state.

Now we show what happens when ProDiagnose determines that a sensor is *stuck*. In Figure 7, the stuck *ST* node is clamped to *zeroDelta*, the Bayesian network name for a stuck state. Again assuming the evidence in the rest of the Bayesian network is the same as in Figures 5 and 6, we see that the most likely value for the sensor's health is *stuck*, with high probability, based on the marginal distribution (Figure 7).

## 5. EXPERIMENTAL RESULTS

ProDiagnose competed in both the ADAPT Tier 1 and Tier 2 Industrial Track of the DXC 09 Competition under the name *ProADAPT* [1]. The competition results are based on multiple metrics, which we will now briefly summarize. A *false positive* refers to detecting a fault when a fault is not present. A *false negative* refers to not detecting a fault when a fault is present. *Classification errors* refer to the number of misdiagnoses made during an entire scenario run. *Detection accuracy* is the percentage of correct fault detections when taking into account the total percentage of false positives and false negatives. *Mean time to detect* refers to the time elapsed between specific fault injection and first detection of a fault. *Mean time to isolate* is similar to the mean detection time, except that an *isolation* refers to identification of the correct fault. *Mean CPU Time* is a measure of CPU resources used by ProDiagnose, and *Mean Peak RAM Usage* measures the maximum amount of memory needed by ProDiagnose.

### *5.1 Tier 1*

The Tier 1 competition consisted of 62 scenarios, either nominal (no fault) or single fault, with no commands (Kurtoglu et al. 2009). Each scenario features the ADAPT Tier 1 system in a fully powered-up state from the beginning.

In Table 5, the ProADAPT DX 09 Competition results are given alongside the results from two naïve variants of ProDiagnose, ProDiagnose' and ProDiagnose", in which various diagnostic features are disabled. ProDiagnose' is defined with *DD* enabled, and *EP*, *CO*, *SSD* disabled. ProDiagnose" is defined with *DD*, *EP*, *CO*, and *SSD* disabled.

[1]The BN files used for Tier 1 and Tier 2 in this paper are named DXCT1.net and DXCT2.net respectively. Discretization and other relevant information is kept in files DXCT1.plog and DXCT2.plog.

| ADAPT Tier 1 (Vista x64 / Core2 T6400, Java) ProADAPT | | | |
|---|---|---|---|
| | **ProDiagnose** | **ProDiagnose '** | **ProDiagnose"** |
| False Positives | 3.33% | 3.33% | 12.12% |
| False Negatives | 3.12% | 15.62% | 13.79% |
| Classification Errors | 2 | 14 | 15 |
| Detection Accuracy | 96.77% | 90.32% | 87.10% |
| Mean Time to Detect | 1387 ms | 139 ms | 135 ms |
| Mean Time to Isolate | 4080 ms | 306 ms | 308 ms |
| Mean CPU Time | 2016 ms | 1908 ms | 2111 ms |
| Mean Peak RAM Usage | 53 MB | 51 MB | 52 MB |

Table 5: Comparison of the ProDiagnose DA against ProDiagnose' and ProDiagnose", for ADAPT Tier 1.

For ADAPT Tier 1, ProDiagnose had very low false positives/negatives rates, with only 2 classification errors, and very high detection accuracy (Table 5). Also, for ProDiagnose' and ProDiagnose" results, we see very fast mean detection and isolation times in around 1/10 and 3/10 of a second, respectively. This is due to *SSD* being disabled, as *DD, EP* and *CO* don't have much impact on Tier 1. Disabling *DD* in ProDiagnose" only gives us 1 more classification error. If the fan component fault scenarios were taken out, the mean detection time would drop to around 1-2 ms, due to the extra time it takes to accurately detect changes in the fan's RPM. Mean Peak RAM usage for Windows is around 52 MB; Linux RAM usage decreases to <2 MB (Kurtoglu et al. 2009).

### *5.2 Tier 2*

The ADAPT Tier 2 competition consisted of 120 scenarios, either nominal, single, double or triple fault, with various relay and circuit breaker open/close commands (Kurtoglu et al. 2009). The Tier 2 EPS starts in a powered down state, in that all commandable relays are open. Then various relays are closed (and some possibly opened again), depending on the scenario.

| ADAPT Tier 2 (Vista x64 / Core2 T6400, Java) ProADAPT | | | |
|---|---|---|---|
| | **ProDiagnose** | **ProDiagnose '** | **ProDiagnose"** |
| False Positives | 7.32% | 48.94% | 100.00% |
| False Negatives | 13.92% | 13.70% | 0.00% |
| Classification Errors | 76 | 109 | 146 |
| Detection Accuracy | 88.33% | 72.50% | 0.00% |
| Mean Time to Detect | 5973 ms | 8556 ms | N/A |
| Mean Time to Isolate | 11988 ms | 19569 ms | 19569 ms |
| Mean CPU Time | 2922 ms | 2819 ms | 2888 ms |
| Mean Peak RAM Usage | 65 MB | 66 MB | 65 MB |

Table 6: Comparison of the ProDiagnose DA against ProDiagnose' and ProDiagnose", for ADAPT Tier 2.

ProDiagnose again had very low false positives/negatives rates, and very high detection accuracy (Table 6). Here, it becomes clear that *DD* is very important for a low false positives rate. ProDiagnose" had a 100% false positives rate (and almost double the number of classification errors as ProDiagnose), but enabling *DD* decreased this rate by about 51% for ProDiagnose'. ADAPT Tier 2 has many *transients* when relays are initially closed to power up the inverters. During this time many sensors give readings that can easily be mis-interpreted as faulty due to this. *DD* helps eliminate this problem by telling ProDiagnose to not make diagnoses during this time. Our false negatives rate drops for ProDiagnose' due to many of these scenarios now showing false positives instead (*CO* being disabled). ProDiagnose" thus has a 0% false negative rate.

It may seem that an 11 second mean isolation time is high, but this is in large part due to stuck faults, as ProDiagnose waits to ensure with high accuracy that a sensor is indeed stuck before submitting a diagnosis for it. Faults involving components such as fans and pumps usually will have high isolation times also, due to a similar principle of waiting. In this case, ProDiagnose waits until the component's sensor readings trip a certain threshold, and the diagnosis is then made based on other node influences within the Bayesian network (The delta *D* node in Figure 2 aids the accuracy of this process). For most types of faults, ProDiagnose has <1 ms isolation time. RAM usage is for Windows is 65 MB; Linux usage is < 7 MB (Kurtoglu et al. 2009).

## 6. CONCLUSION

ProDiagnose is a highly accurate, fast diagnostic algorithm for probabilistic models. It is characterized by quick detection and isolation times, with a high degree of accuracy for detecting faults. ProADAPT's results in the DX 09 Competition backs up these claims. Part of this success was due to the addition of certain nodes *(*Delta *D,* Stuck *ST,* Change *CH)* to an earlier version of the static Bayesian network for ADAPT (Mengshoel et al. 2008), and using dynamic processing in ProDiagnose to calculate their states.

## REFERENCES


M. Chavira and A. Darwiche (2007). Compiling Bayesian networks using variable elimination, *in Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, (Hyderabad, India), pp. 2443-2449.

A. Darwiche (2003). A differential approach to inference in Bayesian networks, *Journal of the ACM*, **vol. 50, no. 3**, pp. 280-305.

T. Kurtoglu, S. Narasimhan, S. Poll, D. Garcia, L. Kuhn, J. de Kleer, A. van Gemund and A. Feldman (2009). Towards a Framework for Evaluating and Comparing Diagnosis Algorithms, *in Proceedings of the 20th International Workshop on Principles of Diagnosis (DX-09)*, (Stockholm, SE).

S. Lauritzen and D. J. Spiegelhalter (1988). Local computations with probabilities on graphical structures and their application to expert systems (with discussion), *Journal of the Royal Statistical Society series B*, **vol. 50**, **no. 2**, pp. 157-224.

O. J. Mengshoel (2007). Designing resource-bounded reasoners using Bayesian networks: System health monitoring and diagnosis, *in Proceedings of the 18th International Workshop on Principles of Diagnosis (DX-07)*, (Nashville, TN), pp. 330-337.

O. J. Mengshoel, A. Darwiche, K. Cascio, M. Chavira, S. Poll, and S. Uckun (2008). Diagnosing faults in electrical power systems of spacecraft and aircraft, *in Proceedings of the Twentieth Innovative Applications of Artificial Intelligence Conference (IAAI-08)*, (Chicago, IL), pp. 1699-1705.

J. Pearl (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, CA: Morgan Kaufmann.

S. Poll, A. Patterson-Hine, J. Camisa, D. Garcia, D. Hall, C. Lee, O. J. Mengshoel, C. Neukom, D. Nishikawa, J. Ossenfort, A. Sweet, S. Yentus, I. Roychoudhury, M. Daigle, G. Biswas, and X. Koutsoukos (2007). Advanced diagnostics and prognostics testbed, *in Proceedings of the 18th International Workshop on Principles of Diagnosis (DX-07)*, (Nashville, TN), pp. 178-185.