The Diagnostic Challenge Competition 2009 (DCC'09)

Tolga Kurtoglu[1], Sriram Narasimhan[1], Scott Poll[1], David Garcia[1],
Lukas Kuhn[2], Johan de Kleer[2], and Alexander Feldman[2,3]

[1]NASA Ames Research Center
[2]Palo Alto Research Center
[3]Delft University of Technology

November 19, 2008

Contents

1 Introduction

The Diagnostic Challenge Competition (DCC) will be the first of a series of international competitions to be hosted yearly at the International Workshop on Principles of Diagnosis (DX). The objectives of the competition are to accelerate research in theories, principles, and computational techniques for monitoring

and diagnosis of complex systems, to encourage the development of software platforms that promise more rapid, accessible, and effective maturation of diagnostic technologies, and to provide a forum that can be utilized by algorithm developers to test and validate their technologies on real-world physical systems.

The overall goal of this competition is to systematically evaluate different diagnostic technologies and to produce comparable performance assessments for different diagnostic methods. To achieve this goal, a number of standardized specifications are introduced, including a physical testbed, a standardized fault catalog, a library of modular and standardized test scenarios, a test protocol, a common method for processing diagnostic data, and common metrics. Different diagnostic technologies will be run under the same conditions and using the same data.

The competition will be announced at the 19'th International Workshop on Principles of Diagnosis (DX-08) and hosted for the first time in 2009.

1.1 Participation Procedures

Those who wish to compete must submit an "intent to participate" form, and must make their tool available to competition organizers in an approved format (See Section 3.1). Both prerequisites must be met by the deadlines specified in Table 1. Registration and competition information will be posted on DASHlink, https://dashlink.arc.nasa.gov/topic/diagnostic-challenge-competition/.

Upon request, participants may be given access to the evaluation code that will be used to calculate their tool's performance. If the tool is accepted for the competition, the participants will be notified accordingly. At least one author per prize-winning tool must come to the conference and discuss the system in person. This will entail giving a short talk during the competition workshop and being ready to demonstrate the system and answer questions during the demonstration session. The competition organizing committee reserves the right to modify the rules of participation and disqualify any participants at their discretion.

1.2 Important Dates & Tentative Schedule

The tentative schedule of the competition is shown in Table. 1.

Date	Venue	Description
Sep 2008	DX'08	Formal announcement
Oct 2008		Submission deadline for intent to participate
Nov 2008		Full information release
Mar 2009		Deadline for submission of diagnostic algorithms
Jun 2009	DX'09	Present results and winners of challenge

Table 1: Important dates.

1.3 Chairs and Organizing Committee

Sriram Narasimhan from NASA Ames and Johan de Kleer from PARC will be co-chairs of DCC'09. Tolga Kurtoglu from NASA Ames will be the organizing committee chair. Scott Poll and David Garcia from NASA Ames and Lukas Kuhn from PARC are the other members of the organizing committee.

1.4 Acknowledgments

We extend our gratitude to Arjan van Gemund (Delft University of Technology), Gregory Provan (University College Cork), Peter Struss (Technical University Munich), the DX'09 organizers and many others for valuable discussions, criticism and help.

2 Competition Architecture

The first competition will start with two tracks (cf. Table 2). Each track will define one or more diagnostic problems. Each diagnostic tool may compete in one or more tracks. There will be some incentives for the winners in each track. (to be specified at a later date)

Identifier	Name	Tier	Systems (See Table 3)	Description
`ind1`	Industrial	1	ADAPT-Lite	Basic faults injected into a simplified EPS testbed.
`ind2`	Industrial	2	ADAPT	More complex faults injected into the full ADAPT system.
`synt`	Synthetic	1	ISCAS-85, ISCAS-89	Arithmetic circuits, random models, complex systems, etc.

Table 2: Tracks in the diagnostic competition. A party providing a system will be referred to as vendor in the remainder of this document.

For the first competition we will have Advanced Diagnostics and Prognostics Testbed (ADAPT) in the industrial track and ISCAS-85 and ISCAS-89 in the synthetic track. Each vendor will provide data and/or a simulator for the artifacts he submits. If the data provided is generated from simulation, the simulation software may or may not be open to the participants.

The industrial track has two tiers and the synthetic track has one. The first tier in the industrial track will involve a simplistic version of the system and will include basic faulty behavior (single fault, no transients, static loads, limited testbed configuration). This tier is intended to encourage competitors interested in participating with minimal effort. The second tier will include complexities such as multiple faults, data transients, multiple loads, and an extended system configuration. The synthetic track will involve select benchmark problems from ISCAS-85, and ISCAS-89. The judges will decide the winner for each track/tier using the evaluation results and the metric scores computed for each algorithm. The final rules will be posted on the competition web page. There will be a presentation of the results and winners' prizes during the conference. Final scores for each entry will also be posted on the competition web page after the competition. An artifact is what the participants are asked to model and diagnose. The Electrical Power System (EPS) implemented by ADAPT is the sole artifact in the industrial track for our first competition. The 74181 ALU is another artifact. An artifact consists of interconnected components, and each component is an instance of a given component type (e.g., a relay, a valve, an and-gate, a wire).

2.1 XML System Descriptions

XML system descriptions will be provided to describe the functioning of the artifacts. Beyond being formatted in XML, they are not formalized. This is to prevent biasing the modeling choices. We recognize the fact that *any* artifact description (e.g., graphical, programmatic, etc.) contains certain biasing towards a modeling approach, but the above applies not only to the description of the artifact but to the artifact itself.

Almost any diagnostic technology today uses some kind of graph-like structure for describing the system structure, hence we will provide graphs of the physical connectivity of the system. This graph is not annotated: for example there is no directional information, etc. The latter kind of data can be extracted from the repository of documents describing the system, if available.

This is an example XML system description:

```
<system>
  <systemName>polycell</systemName>
  <description>
    A familiar circuit. Contact: dekleer@parc.com. Publications: [dW87]
  </description>
```

Name	Description
System Name	Unique identifier
Artifact Description	Brief text summary of the system and pointers to documentation, forums, mailing lists, and other resources
Component Catalog	List of component identifiers, with reference to component type and commands that affect the component
Interconnection Diagram	Each node of the graph contains a component identifier/instance identifier pair, and there is an edge for any two (physically) connected components

Table 3: System description data.

```
  <components>
     <component>
        <name>A</name>
        <type>PROBE</type>
     </component>
     <component>
        <name>M1</name>
        <type>MUL</type>
     </component>
     <component>
        <name>A1</name>
        <type>ADD</type>
     </component>
  </components>

  <connections>
    <connection>
      <c1>A</c1>
      <c1>M1</c1>
    </connection>
    <connection>
      <c1>M1</c1>
      <c1>A1</c1>
    </connection>
  </connections>
</system>
```

2.2 XML Component Type Descriptions

Next we have to provide specifications for all component types mentioned in the system description. Consider the "CircuitBreaker" ADAPT component type (referenced, for example, by the component with unique ID `CB180`):

```
<componentType xsi:type="circuitBreaker">
  <name>CircuitBreaker4Amp</name>
  <description>
    4 Amp CircuitBreaker. http://adapt.nasa.gov/ Contact: scott.poll@nasa.gov.
  </description>
  <modesRef>CircuitBreaker</modesRef>
```

Name	Description
Component Type Name	Unique identifier
Component Description	Brief summary of the component type and pointers to documentation, forums, and other resources
Modes	Reference to a mode group (Table 5)
Component-Specific Info	Examples: sensor min/max, load wattage, circuit breaker rating

Table 4: Component description data.

```
  <rating>4</rating>
</componentType>
```

Or an "ACVoltageSensor" ADAPT component type:

```
<componentType xsi:type="sensor">
  <name>ACVoltageSensor</name>
  <description>AC voltage sensor.</description>
  <modesRef>ScalarSensor</modesRef>
  <datatype>double</datatype>
  <engUnits>VAC</engUnits>
  <rangeMin>0</rangeMin>
  <rangeMax>150</rangeMax>
</componentType>
```

As a part of a more abstract example we can consider a description of an and-gate, part of a digital circuit:

```
<componentType>
  <name>AND2</name>
  <description>
    AND gate.
  </description>
  <modesRef>AndGate</modesRef>
</componentType>
```

2.3 XML Mode Catalog

Component operating modes are organized by Mode Groups. More than one component can refer to the same group. The Mode Group format is described in Table 5.

Name	Description
Modes Group Name	Unique identifier for each mode class
Mode Names	Names of the possible modes
Mode Descriptions	Text descriptions

Table 5: Mode group description.

Consider the allowable modes for the CircuitBreaker component from the preceding section:

```
<modeClass>
  <name>CircuitBreaker</name>
  <mode>
    <name>Nominal</name>
```

```
      <description>
        Transmits current and voltage. Trips when current exceeds threshold.
      </description>
    </mode>
    <mode>
      <name>Tripped</name>
      <description>
        Breaks the circuit and must be manually reset.
      </description>
    </mode>
    <mode xsi:type="faultMode">
      <name>FailedOpen</name>
      <description>
        Trips even though current is below threshold.
      </description>
    </mode>
  </modeClass>
```

2.4 Communications Regarding System Descriptions

Participants may ask vendors questions about the artifacts and their components. The vendor will decide whether or not to answer these questions. The only condition is that *all* the dialog will become available to *all* the participants in the competition via the competition website.

3 Software Architecture for Running DCC

The DCC effort includes the development of a software framework for running and evaluating diagnostic algorithms. The framework has been designed with the following considerations in mind: (1) the overhead of interfacing existing diagnostic algorithms should be reduced by supplying minimalistic APIs, (2) inter-platform portability should be provided by allowing clients to interface C++ and Java API or by implementing a simple ASCII based TCP messaging protocol, and (3) the framework should be realistic by reflecting industrial practices and needs.

To facilitate algorithm development and testing, we will provide binary packages for Windows and Linux. All Java and C++ source code will be made available. Our framework will contain a very simple diagnostic algorithm and a few examples.

3.1 Software Components

Figure 1 shows an overview of the DCC software components and the primary information flows. As we have already mentioned, all communication is ASCII based and all the modules communicate via TCP ports by using a simple message-based protocol which we describe in more detail below.
Next, we provide a brief description of each software component.

Scenario Loader: Executes the Scenario Data Source, Recorder, and Diagnostic Algorithm. Ensures system stability and clean-up upon scenario completion. This is the main entry point for performing a diagnostic experiment.

Scenario Data Source: Provides scenario data from previously recorded datasets. The provenance of the data (whether hardware or simulation) depends on the system in question. A scenario dataset contains sensor readings, commands[1], and fault injection information (to be sent exclusively to the scenario recorder).

[1]Note that the majority of classical MBD literature does not discern commands from observations.

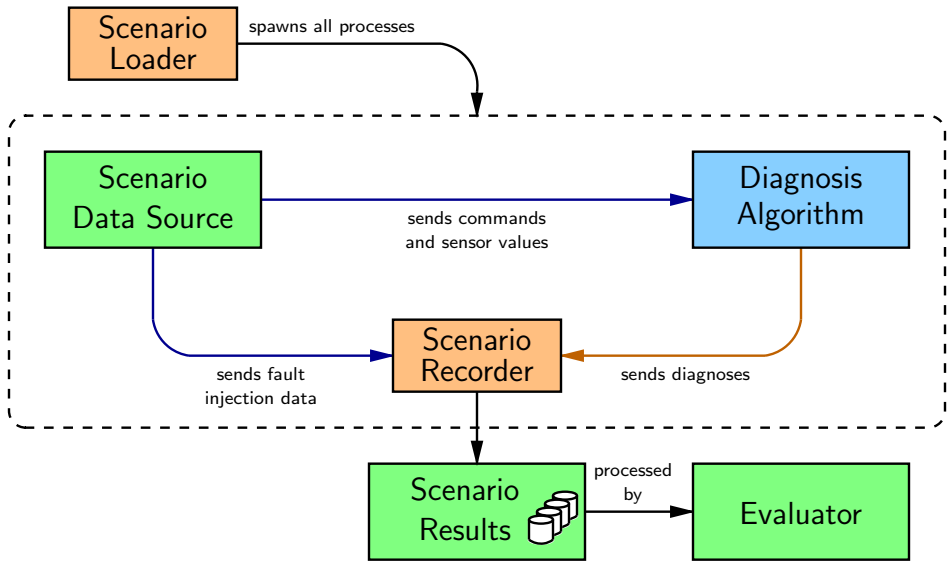

Figure 1: Architecture overview of the DCC software framework.

The Scenario Data Source will publish data following a wall-clock schedule specified by timestamps in the scenario files.

Scenario Recorder: Receives fault injection data and diagnosis data, and compiles it into a Scenario Results File. The results file contains a number of time-series which will be described below. These time-series are used by the Evaluation module for scoring and can be supplied to the participants for detailed analysis of the algorithmic performance.

The Scenario Recorder is the main timing authority, i.e., it timestamps each message upon arrival before recording it to the Scenario Results File.

Diagnostic Algorithm: A Windows or Linux executable (or Java bytecode in an executable .jar file) contained in a single directory, requiring no compilation, installation, or external libraries except the sockets API and its dependencies. Receives sensor data and command data, performs diagnosis, and sends diagnosis results back.

Evaluator: Takes Scenario Results File and applies metrics to evaluate Diagnosis Algorithm performance. The metrics and evaluation procedures are detailed in Sec. 4.

3.2 Messaging Datatype Specifications

This section defines the data types for the messages that will be sent to and from the Diagnosis Algorithm. The DCC API and TCP/IP interfaces conform to these data types. Documentation for these interfaces can be found in Appendix A and Appendix B.

DiagnosisData
+timeStamp +candidateSet: Set <Candidate> +detectionSignal: boolean +isolationSignal: boolean +notes: string

Candidate
+faults: Map<componentIds->componentStates>
+weight: double

SensorData
+timeStamp
+sensorValues: Map<sensorIds->sensorValues>

CommandData
+timeStamp
+actuatorID: string
+command: string

ErrorData
+timeStamp
+message: string

3.3 Diagnostic Session Overview

The diagnostic algorithms will be tested against a number of diagnostic scenarios. From the viewpoint of the scenario player, a diagnostic scenario is a series of observations (sensor readings, commands) $A = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, taken within an interval of time. The vendor produces A for each scenario with the help of a physical test-bed, a simulator or other method for creating observations (it is not necessary for all available sensor values to be reported to the diagnostic engine). The aim of the vendors is to provide scenarios with varying levels of difficulty. The diagnostic scenarios will be kept secret from the participants, and only a small number of diagnostic scenarios will be provided for testing.

The first competition will have non-intermittent faults only (this may be reconsidered in the follow-up events). Note that multiple faults can be injected at different times.

We will analyze the progression of one diagnostic scenario. Each diagnostic session defines some standard key points and intervals which are best illustrated by Fig. 2.
Figure 2 splits the diagnostic session in three important time intervals: Δ_{startup}, $\Delta_{\text{injection}}$, and Δ_{shutdown}. During the first interval Δ_{startup}, the diagnostic algorithm is given time to initialize, read data files, etc. Though sensor observations may be available during Δ_{startup}, no faults will be injected at this time. Fault injection and diagnosis is to take place during $\Delta_{\text{injection}}$. Finally, to stimulate good programming practices, the algorithms will be given some time to gracefully terminate during Δ_{shutdown}. After this time, live diagnostic processes will be killed and the system will be recycled for the next diagnostic experiment.

Below are some notable points for the example diagnostic scenario from Fig. 2:

t_{inj} – A fault is injected at this time;

t_{fd} – The diagnostic algorithm has detected a fault;

t_{ffi} – The diagnostic algorithm has isolated a fault for the first time;

t_{fir} – The diagnostic algorithm has retracted its isolation assumption, expecting more faults;

t_{lfi} – This is the last fault isolation during $\Delta_{\text{injection}}$.

A sequence diagram of an example diagnostic session is shown in Fig. 3.
At the end of the diagnostic session the scenario player has collected the following time-series and (actual and hypothesized) fault data to be used in the metrics computation:

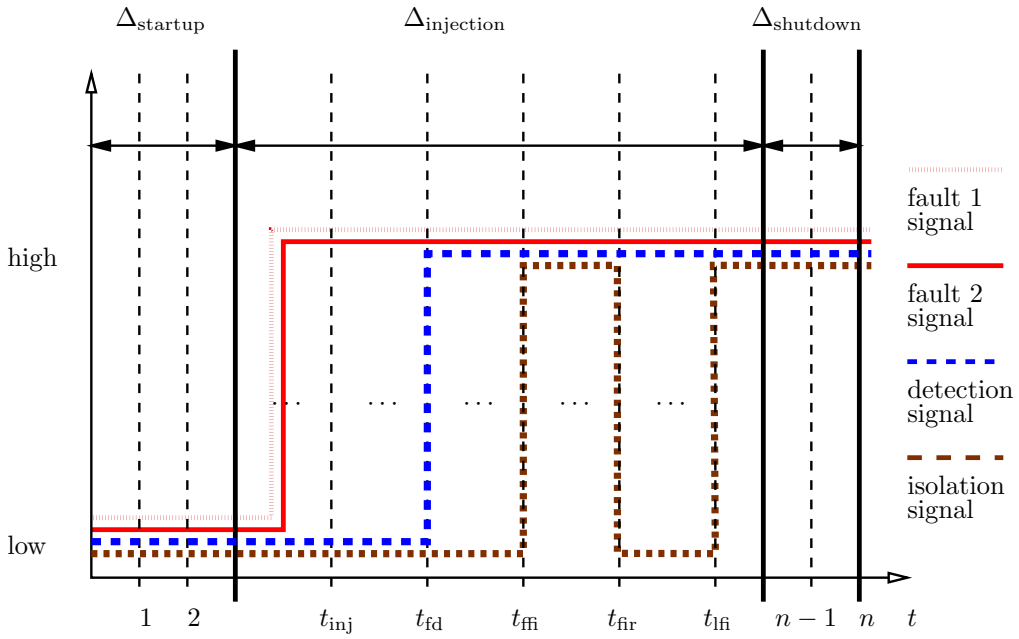

Figure 2: Key time points, intervals, and signals.

T_{inj} – Fault injection signal;

T_{fd} – Fault detection signal;

$\omega^\star$ – An actual fault set (once all faults are injected).

$\Omega = \{\omega_1, \omega_2, \ldots, \omega_n\}$ – A (possibly empty) set of candidate diagnoses.

In addition to that the scenario player should collect the following diagnostic engine session performance data (to be used for the computation of performance metrics):

T_d – Computation time in ms at each step;

M_d – Amount of allocated memory in bytes at each step.

4 Metrics end Evaluation Engine

We have defined a number of metrics for evaluating the diagnostic engines. Each metric is a real-valued function. Table 6 summarizes the eight metrics we plan to use for evaluating the results from a diagnostic competition.
Table 7 provides a summary of the notation used throughout this section.
We will next describe each metric in more detail.

4.1 "Per System Description" Metrics

False Positives Rate (M_{fp})**:** The metric below penalizes diagnostic engines which announce spurious faults. The false positive rate is defined as:

$$M_{\text{fp}} = \frac{\sum_{s \in S} m_{\text{fp}}(s)}{|S|} \tag{1}$$

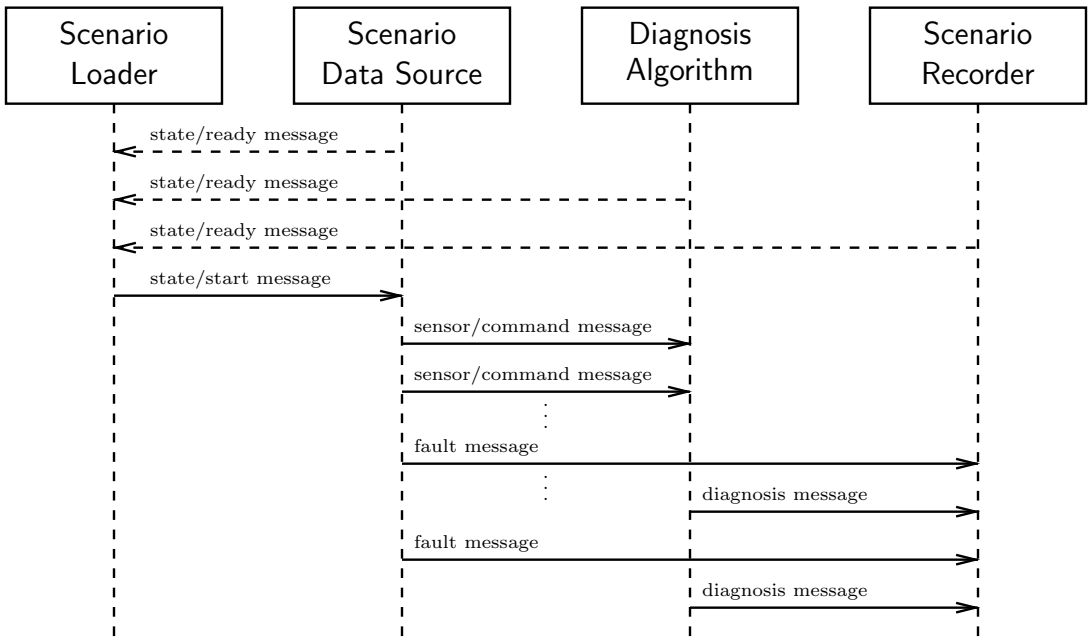

Figure 3: Session sequence diagram.

where for each scenario s the "false positive" function $m_{\text{fp}}(s)$ is defined as:

$$m_{\text{fp}}(s) = \begin{cases} 1, & \text{if } t_{fd} < t_{inj} \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

Some applications (e.g., aborting a mission and endangering human life based on false positive) do not tolerate false positives. Other applications (e.g., run additional slightly more expensive tests upon suspicion) would tolerate occasional false positives.

False Negatives Rate (M_{fn})**:** The metric defined next measures the ratio of missed faults by a diagnostic engine.

$$M_{\text{fn}} = \frac{\sum_{s \in S_f} m_{\text{fn}}(s)}{|S_f|} \tag{3}$$

where for each scenario s the "false negative" function $m_{\text{fn}}(s)$ is defined as:

$$m_{\text{fn}}(s) = \begin{cases} 1, & \text{if } t_{\text{fd}} = \infty \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

Detection Accuracy (M_{da})**:** The detection accuracy is the ratio of number of correctly classified cases to the total number of cases. It is defined as:

$$M_{\text{da}} = 1 - \frac{\sum_{s \in S} m_{\text{fp}}(s) + m_{\text{fn}}(s)}{|S|} \tag{5}$$

Isolation Accuracy (M_{ia})**:** Let us denote the injected fault as ω^*. The isolation accuracy is then defined as:

$$M_{\text{ia}} = \sum_{c \in \text{COMPS}} \frac{\sum_{\omega \in \Omega} m_{\text{ia}}(c, \omega, \omega^*)}{|\Omega| \cdot |\text{COMPS}|} \tag{6}$$

Symbol	Name	Description
"Per System Description" Metrics		
M_{fp}	False Positives Rate	Spurious faults rate
M_{fn}	False Negatives Rate	Missed faults rate
M_{da}	Detection Accuracy	Correctness of the detection
M_{ia}	Isolation Accuracy	Correctness of the isolation
"Per Scenario" Metrics		
M_{fd}	Fault Detection Time	Time for detecting a fault
M_{fi}	Fault Isolation Time	Time for minimizing the diagnostic entropy
M_{cpu}	CPU Load	CPU time spent
M_{mem}	Memory Load	Memory allocated

Table 6: Metrics summary.

Symbol	Description
S	The set of scenarios for a given system description.
S_n	The set of nominal scenarios for a given system description.
S_f	The set of faulty scenarios for a given system description.
t_{fd}	The time when the fault detection signal has been asserted for the first time.
t_{inj}	The time when the first fault has been injected.
t_{fi}	The time when the fault isolation signal has been asserted for the last time.
ω^*	The injected fault.
Ω	The last set of faults returned by the diagnostic engine.
(ω, c)	The actual health state of a component c in a diagnosis ω.
t_s	The CPU time during startup.
T_d	A vector of CPU times spent during each time step in a scenario.
M_d	A vector of maximal memory allocation sizes during each time step in a scenario.

Table 7: Metrics notation summary.

where $m_{\text{ia}}(c, \omega, \omega^*)$ is defined as:

$$m_{\text{ia}}(c, \omega, \omega^*) = \begin{cases} 0, & (\omega, c) \neq (\omega^*, c) \\ 1, & \text{otherwise} \end{cases} \tag{7}$$

and (ω, c) is the state of component c in diagnosis ω. Note that M_{ia} is different from M_{da} as M_{da} considers only if the component c is healthy/faulty (i.e., a Boolean state of c), while M_{ia} allows multiple health/fault states. An example case study including the calculation of all the aforemetioned per system description metrics will later be posted on the competition website.

4.2 "Per Scenario" Metrics

Fault Detection Time (M_{fd})**:** The next metric quantifies the reaction time for a diagnostic engine in detecting an anomaly.

$$M_{\text{fd}} = t_{\text{fd}} \tag{8}$$

where t_{fd} is the first time when the fault detection signal has been asserted.

Fault Isolation Time (M_{fi})**:** The next metric measures the time for isolating a fault. In many applications this metric is less important than the diagnostic accuracy, but it is important in sequential diagnosis, probing, etc.

$$M_{\text{fi}} = t_{\text{fi}} \tag{9}$$

where t_{fi} is the first instance of time when the latest persistent fault isolation signal has been raised.

CPU Load (M_{cpu})**:** This is the average CPU load during the experiment:

$$M_{\text{cpu}} = t_s + \sum_{q \in T_d} q \tag{10}$$

In the above formula T_s is the startup time of the diagnostic engine and T_d is a vector with the actual CPU time spent by the diagnostic algorithm at every time step in the diagnostic session.

Memory Load (M_{mem})**:** This is the maximal memory load during the experiment:

$$M_{\text{mem}} = \max_{m \in M_d} m \tag{11}$$

In the above formula M_d is a vector with the maximum memory size at every step in the diagnostic session.

4.3 Running and Scoring the Diagnostic Engines

The weighted sum of all the metrics for each scenario in each system description will be used for the ranking of the diagnostic algorithms. The weights are vendor and application dependent, will be determined by the panel of DCC jurors, and will be announced in advance of the competition.

The jury has the right to disqualify any engine which exploits the scoring mechanism to its advantage (e.g., it is possible to cheaply get all points for CPU and memory performance without producing any diagnoses).

5 Future Work

After DCC'09 we aim at broadening the competition by including problems from probing (determination of "best" measurement), Model-Based Testing (ATPG-related problems), intermittent faults, planning problems related to diagnosis and recovery, and others.

A BNF Specification of Messaging

Below is the BNF specification for the TCP/IP messages to and from the Diagnostic Algorithm. It is provided for informational purposes only and subject to change without notification up until the DXC full information release date.

```
/*
 * Messages from Scenario Data Source -> Diagnostic Algorithm
 */

<message_list> ::= <message_list> <message> |
                   <message>
                 ;
```

```
<message> ::= <command_message> |
              <sensor_message> |
              <error_message>
            ;

<command_message> ::= COMMAND_MESSAGE INTEGER /* timestamp */
                       <actuator_id> '=' <string_value> ';'
                     ;

<sensor_message> ::= SENSOR_MESSAGE INTEGER /* timestamp */
                      '{' <sensor_reading_list> '}' ';'
                    ;

<error_message> ::= ERROR_MESSAGE STRING ';'
                   ;

<sensor_reading_list> ::= <sensor_reading_list> ',' <sensor_reading> |
                          <sensor_reading> |
                          /* empty */
                        ;

<sensor_reading> ::= <sensor_id> '=' <value>
                     ;

<actuator_id> ::= STRING
                  ;

<sensor_id> ::= STRING
                ;

<value> ::= <bool_value> |
            <real_value> |
            <integer_value> |
            <string_value>
          ;

<bool_value> ::= BOOL
                 ;

<real_value> ::= DOUBLE
                 ;

<integer_value> ::= INTEGER
                    ;

<string_value> ::= STRING
                   ;

/*
 * Diagnostic Algorithm -> Scenario Recorder
 */
```

```
<diagnosis_message> ::= DIAGNOSIS_MESSAGE
                        DETECTION_SIGNAL '=' BOOL ','
                        ISOLATION_SIGNAL '=' BOOL ','
                        '{' <candidate_list> '}' ','
                        '{' <weight_list> '}' ','
                        NOTES '=' STRING /* misc notes */
                      ;

<candidate_list> ::= <candidate> |
                     <candidate_list> ',' <candidate> |
                     /* empty */
                   ;

<candidate> ::= '{' <fault_list> '}'
              ;

<fault_list> ::= <fault> |
                 <fault_list> ',' <fault> |
                 /* empty */
               ;

<fault> ::= <component_id> '=' <mode_id>
          ;

<weight_list> ::= <weight> |
                  <weight_list> ',' <weight> |
                  /* empty */
                ;

<component_id> ::= STRING
                 ;

<mode_id> ::= STRING
              ;

<weight> ::= REAL
           ;
```

B DCC C++ and Java Diagnostic Algorithm API

The framework implementation has been well under way at the time of releasing the first revision of this document. Upon completion of the API, the current Appendix is going to be updated with the actual documentation generated from the source code and a new release of the document is to be released.

References

[dW87] Johan de Kleer and Brian Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.