White Paper

DSP0138 **Status: Preliminary**

Copyright © 2002 Distributed Management Task Force, Inc. (DMTF). All rights reserved.

DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems management and interoperability. Members and non-members may reproduce DMTF specifications and documents for uses consistent with this purpose, provided that correct attribution is given. As DMTF specifications may be revised from time to time, the particular version and release date should always be noted.

Implementation of certain elements of this standard or proposed standard may be subject to third party patent rights, including provisional patent rights (herein "patent rights"). DMTF makes no representations to users of the standard as to the existence of such rights, and is not responsible to recognize, disclose, or identify any or all such third party patent right, owners or claimants, nor for any incomplete or inaccurate identification or disclosure of such rights, owners or claimants. DMTF shall have no liability to any party, in any manner or circumstance, under any legal theory whatsoever, for failure to recognize, disclose, or identify any such third party patent rights, or for such party's reliance on the standard or incorporation thereof in its product, protocols or testing procedures. DMTF shall have no liability to any party implementing such standard, whether such implementation is foreseeable or not, nor to any patent owner or claimant, and shall have no liability or responsibility for costs or losses incurred if a standard is withdrawn or modified after publication, and shall be indemnified and held harmless by any party implementing the standard from any and all claims of infringement by a patent owner for such implementations.

CIM Diagnostic Model White Paper
CIM Version 2.9
Document Version 1.0 July 6, 2004

Abstract

Diagnostics is a critical component of systems management. Diagnostic services are utilized in problem containment to maintain availability, achieve fault isolation for system recovery, establish system integrity during boot, increase system reliability, and perform routine preventive maintenance. The goal of the Common Diagnostic Model (CDM) is to define industry standard building blocks based upon, and consistent with, the DMTF Common Information Model (CIM), that enables seamless integration of vendor-supplied diagnostic services into system and SAN management frameworks.

In this paper, the motivation behind CDM is presented. In addition, the core architecture of the CDM is presented in the form of a diagnostic schema. The original version of the schema is presented (CIM V2.3), along with extensions introduced beginning with V2.7. Proper usage of the schema extensions is presented in a tutorial manner. Future direction for the CDM is discussed to further illustrate the motivations driving CDM development, including interoperability, self-management, and self-healing of compute resources.

Change History

Version 0.1	Russ Carr	Initial: Intro Draft & outline
Version 0.2	Russ Carr	Team reviewed sections: 1-2.2
Version 0.3	Russ Carr Ray Pedersen Michael Kehoe Barbara Craig	Re-organized & revised sections 2.3-2.7 - sections 2.3, 2.6, & 2.7 - section 2.4 - section 2.5
Version 0.4	Ray Pedersen	Modifies outline, content largely unchanged.
Version 0.5	Ray Pedersen Michael Kehoe Rob Branch	Content clarification and corrections
Version 0.6	Ray Pedersen	Cleanup for tm-diag final review
Version 0.7	Ray Pedersen	Added “Who Should Read the Paper” Added some terms and conventions Updated with tm-diag feedback
Version 0.8	Ray Pedersen	Updated with comments from tm-diag and sysdev reviews.
Version 0.9	Ray Pedersen	First ballot feedback.
Version 0.91	Rob Branch	Minor content corrections and clarifications
Version 0.92	Ray Pedersen	Cleanup for V2.8 review.
Version 0.93	Ray Pedersen	Cleanup for V2.8 review.
Version 0.94	Ray Pedersen	Cleanup for V2.8 review. Incorporate new CRs for LogOptions and ConcreteJob.
Version 0.95	Ray Pedersen	Update for CIM V2.9 New Logging mechanism.
Version 1.0	Ray Pedersen	Submitted to SysDev for Review

TABLE OF CONTENTS

ABSTRACT 2

CHANGE HISTORY 3

1 INTRODUCTION 7

1.1 Overview 7

1.2 Goals 8
1.2.1 Manageability through Standardization 8
1.2.2 Interoperability 8
1.2.3 Diagnostic Effectiveness 8
1.2.4 Global Access 9
1.2.5 Life-cycle Applicability 9

1.3 Who Should Read this Paper 9

1.4 CDM Versions 9

1.5 Background Reference Material 10

1.6 Terminology 10

1.7 Conventions Used in this Document 10

2 MODELING DIAGNOSTICS 12

2.1 Consumer - Provider Protocol 12

2.2 Implementation Neutral Modeling 12

2.3 Backward Compatibility 12

2.4 Diagnostics are Services 13

2.5 Diagnostics are Applied to Managed Elements 13

2.6 Generic Framework 14
2.6.1 Diagnostic Control 14
2.6.2 Diagnostic Logging & Reporting Assumptions 14
2.6.3 Localization 15

3 CDMV1 16

3.1 Overview 16

3.2 Model Components 17
3.2.1 The DiagnosticTest Class 17

3.2.2 The DiagnosticSetting Class 17
3.2.3 The DiagnosticResult Class 18

3.3 CDMV1 Usage 19
3.3.1 Settings Protocol 19
3.3.2 Looping 20
3.3.3 Result Persistence 21
3.3.4 LogOptions for Typed Messages 21
3.3.5 Diagnostic Results 22
3.3.5.1 Monitoring Diagnostic Test Progress 22
3.3.5.2 Using Typed Messages in TestResults[ ] 23

4 CDMV2 24

4.1 Overview 25

4.2 Model Components 25
4.2.1 Diagnostic Service 25
4.2.2 Diagnostic Jobs 26
4.2.3 Diagnostic Logs 27
4.2.3.1 DiagnosticRecord 28
4.2.4 HelpService 29

4.3 CDMV2 Usage 30
4.3.1 Diagnostic CIM Client Protocol 30
4.3.1.1 Query for Services 30
4.3.1.2 Configure the Service 30
4.3.1.2.1 Settings 30
4.3.1.2.2 Capabilities 30
4.3.1.2.3 Characteristics 31
4.3.1.2.4 Affected Resources 31
4.3.1.2.5 Dependencies 31
4.3.1.3 Execute the Service 31
4.3.1.3.1 Starting a Job 31
4.3.1.4 Monitor/Control the Service 32
4.3.1.5 Complete the Service 32
4.3.2 Correlation of Records 32
4.3.2.1 CDM Key Structure 34
4.3.2.1.1 ConcreteJob Keys 34
4.3.2.1.2 DiagnsoticRecord Keys 34
4.3.3 Using the Physical Model for FRU Identification 35

5 FUTURE DEVELOPMENT 36

5.1 CIM Indications 36

5.2 Interactive Testing 36

5.3 Diagnostics DTD/XSL 37

5.4 Services 37
5.4.1 Daemons 37
5.4.2 Exercisers 37
5.4.3 Executives 37

5.5 Logging 37

5.6 Self Healing and Autonomic Healthcare 37

1 Introduction

The **Common Diagnostic Model (CDM)** is an architecture and methodology for exposing system diagnostic instrumentation through standard CIM interfaces. The CDM schema was introduced in CIM V2.3 as a simple set of classes representing tests, test settings and results. Subsequently, a significant amount of implementation has taken place and a number of opportunities for improvement have been identified in the original model. In addition, CIM has matured, the schema has been extended, and some of these changes are being applied to CDM to improve versatility and extendibility. A number of major changes occurred as part of the V2.9 release and it is anticipated that phasing over to the new schema will be completed with V3.0.

The CDM version introduced in V2.3 is now being referred to as CDMV1 to distinguish it from new concepts introduced in V2.9. CDMV2 encompasses these concepts and will be the only version supported in CIM V3.0.

> **Note:** Most of the extensions made available in CIM V2.9 were actually processed as Change Requests for V2.8. The entire group of changes for CDMV2 was not complete when V2.8 went Final, so all experimental classes were moved into V2.9 Preliminary. This paper will reference V2.9 for all these changes.

The purpose of this paper is to describe the CDM schema as it appears in CIM V2.9, distinguish between CDMV1 and CDMV2, and point out where future work is planned. Guidance is provided, where appropriate, to client and provider implementers in order to reinforce the standardization goal. Guidance to diagnostic test developers is not within the scope of this paper and is being documented by the CDM Industry Group[1].

1.1 Overview

Diagnostics is a term that has been used to describe a wide variety of problem determination and prevention tools that include exercisers, excitation/response tests, information gatherers, configuration tools, and predictive failure techniques. This paper presupposes the most general interpretation of this terminology and addresses all forms of diagnostic tools that would be utilized in OS-present and pre-boot environments. The focus is on CDM, the enabling infrastructure.

The OS-present environment presents a formidable set of challenges to diagnostics programmers. They must deal with a plethora of system status and information, neatly hidden behind proprietary APIs and undocumented incantations; this situation is remedied by CIM. They are also faced with OS barriers placed between user space and the target of their efforts, making it difficult, often impossible, to manipulate the hardware directly. The CDM focuses on easing this situation through a standardized approach to diagnostics that utilizes the more sophisticated aspects of CIM – the ability to manipulate manageable system components by invoking methods.

[1] The CDM Industry Group is currently an ad hoc committee of industry CDM promoters that is developing a set of CDM implementation guidelines. See http://www.intel.com/design/servers/CDM/index.htm.

1.2 Goals

The goals of the Common Diagnostic Model are:

1. Manageability through Standardization
2. Interoperability
3. Diagnostic Effectiveness
4. Global Access
5. Life-cycle Applicability

1.2.1 Manageability through Standardization

Faced with the requirement to deliver diagnostic tools to their customers, chip and adapter developers have had to deal with a variety of proprietary APIs, report formats, and deployment scenarios. The CDM specifies a common methodology, with CIM at its core, which will result in a "one size fits all" diagnostic package. Diagnostic management applications will be able to obtain information about diagnostic services available, configure and invoke diagnostics, monitor the diagnostics progress, control diagnostic execution, and query CIM for information gathered by the diagnostic service. If the CDM methodology is followed, these standard diagnostic packages can be seamlessly incorporated into applications implemented as CIM clients. This relieves the diagnostic programmer from the effort associated with satisfying multiple interfaces and permits more time to be spent improving the effectiveness of the tools.

1.2.2 Interoperability

CIM is, by design, platform-neutral. There is no requirement that implementations of CIM (clients, object managers, and providers) are platform-neutral, but this is the goal. To the extent that CIM implementations promote interoperability, so will the CDM, and this is a very important advantage. Diagnostic CIM clients and providers can be made portable, not only between customers, but also across platforms and in heterogeneous environments.

1.2.3 Diagnostic Effectiveness

Surrounding this infrastructure are the diagnostic tools themselves. Not only do they become less difficult to deploy when developed to the CDM, but also there is a significant potential for improving the effectiveness of the entire package. There are several factors at play. Ease of deployment through standardization and interoperability increases availability, thus expanding coverage. Tool developers have the entire WBEM instrumentation database to draw upon in their problem determination and resolution efforts. The CDM also goes beyond WBEM in recommending techniques to vendors that lead to integration of diagnostics into device drivers, thus gaining access to the more intimate details of the device being diagnosed.

1.2.4 Global Access

WBEM provides a framework for managing system elements across distributed environments, endowing the CDM with the potential for servicing systems without regard to locale. The way is paved for very cost effective serviceability scenarios and major warranty expense reduction.

1.2.5 Life-cycle Applicability

The CDM is designed to be applicable at all product life cycle stages from system development and test to manufacturing, end user, service and warranty repair.

1.3 Who Should Read this Paper

This paper has been prepared to aid diagnostic client and provider developers to understand the CIM components of the Common Diagnostic Model along with other areas of the model that are leveraged to fulfill the requirements of a comprehensive problem determination methodology for modern computer systems. It should be read and understood by anyone planning to create CDM-compliant diagnostic tools.

This paper presupposes the availability and similar study of the CIM Version 2.9 schema, represented by the MOF files. There is detailed information in these files that will not be covered in its entirety in this paper.

This paper deals primarily with architecture. The CDM includes implementation standards as well, in order to promote OEM/vendor interoperability and code reuse. In a separate effort, industry promoters of this technology are preparing a CDM Implementation Guide, which is released at Version 1.0 at the writing of this paper. It is available at the link in Section 1.5. This document should also be read and understood, since it addresses issues related to compliance. Tools are being developed to verify CDM compliance and it is expected that procurement processes will include such testing.

1.4 CDM Versions

CDM Version 1.0 (CDMV1) was introduced in CIM V2.3. It is based upon a simple settings/test/results model and has been enhanced in subsequent versions of the CIM schema. The intent is to deprecate the model components peculiar to CDMV1 prior to the introduction of CIM V3.0, at which time support for CDMV1 clients and providers will be discontinued. This is being done with the realization that CDMV1 has not been widely implemented, providing an opportunity to redefine CDM to be more versatile and extensible, and to more completely leverage applicable CIM schema.

CDM Version 2.0 (CDMV2) was introduced with CIM V2.9. The settings/test/results concept is still present, but modeled using services, jobs, and logs.

1.5 Background Reference Material

1. Original white paper: *A Diagnostic Model in CIM*, http://www.dmtf.org/educ/whit.html
2. CIM Tutorial, http://www.dmtf.org/spec/cim_tutorial/
3. CIM Schema at http://www.dmtf.org/standards/index.php
4. CDM Implementation Guide at http://www.intel.com/design/servers/CDM/index.htm

1.6 Terminology

Term	Definition
CDM	Common Diagnostic Model
CDMV1	Version 1 of the CDM (based on CIM V2.3)
CDMV2	Version 2 of the CDM (based on CIM V2.9)
CIM	Common Information Model
CIMOM	CIM Object Manager
CR	(CIM) Change Request
DBCS	Double Byte Character Set
FRU	Field Replaceable Unit
ME	ManagedElement
MOF	Managed Object Format
MSE	ManagedSystemElement (the class or its children)
NLS	National Language Support
RAS	Reliability, Availability, and Serviceability
SAN	Storage Area Network
UML	Unified Modeling Language
WBEM	Web Based Enterprise Management
XML	Extensible Markup Language

1.7 Conventions Used in this Document

Classes and properties are written using capitalized words without spaces, as in ManagedElement, versus "managed element" when referring to the generic form.

The **Bold** attribute is added for visual impact with no other implied meaning.

Methods include ( ) for quick identification, as in RunTest( ).

Arrays include [ ] for identification, as in TestResults[ ].

A colon between class names is interpreted as "derived from" as in ConcreteJob : Job.

A "dot" between a class name and a property name is interpreted as "containing the property" as in Capabilities.InstanceID (InstanceID is a property of the Capabilities class.)

The prefix "CIM_" will often be omitted from class names for brevity and readability reasons.

2 Modeling Diagnostics

The diagnostic model extends the CIM schema to cover the management domain of diagnostics. This includes diagnostic tests, executives, monitoring agents, and analysis tools. The objective of diagnostic integration into CIM is to provide a framework where industry standard building blocks that contribute to the ability to diagnose and prognosticate the system's health can be seamlessly integrated into enterprise management applications and policies. This chapter discusses the modeling concepts that are relevant to implementing diagnostics with CIM.

2.1 Consumer - Provider Protocol

A CIM diagnostic solution has two components: diagnostic consumers (or diagnostic CIM clients) and diagnostic providers. Diagnostic providers register the classes, properties, methods, and indications they support with the CIM object manager. When a management client queries CIM for diagnostics supported on a given managed element, CIM returns the instances of the diagnostic services associated with that managed element. This establishes communication between the discovered diagnostic providers and the managing client. The management client can now query CIM for properties, enable indications, or execute methods according to the WBEM standard and the diagnostic protocol conventions described in this document. The conventions that diagnostic consumers and providers must follow include naming of keys, consistent manipulation of properties, adherence to life cycle attributes of objects, and synchronization of object references.

2.2 Implementation Neutral Modeling

The diagnostic model is implementation neutral. It does not make any assumptions on any of the following provider implementation approaches:

- Whether the provider is re-entrant or for exclusive use.
- Whether the provider is implemented in-process and blocks until the method requested completes, or is implemented out-of-process so that more than one method can be executed at a time by the same provider.
- Whether the provider is implemented as an "always resident" service or it loads a separate instance for each request and unloads when complete.
- Whether a provider reuses objects, or creates and destroys them for each use.
- Whether more than one provider is used to implement the diagnostic service.
- Whether or not the diagnostic provider supports indications.

2.3 Backward Compatibility

The CIM V2.9 diagnostic model (aka CDMV2) does create some parallel semantics to the CDMV1, and also extends the model to provide additional semantics. In order to make these extensions cleanly, some parts of the diagnostic model were deprecated in

favor of more scalable approaches. These deprecations will continue to be supported in the CIM schema until Version 3.0. Provider developers should implement to the new V2.9 semantics, avoiding any use of the deprecations. Clients, however, may need to support both the CDMV1 and CDMV2 semantics for backward compatibility, until the transition to V3.0 is complete.

2.4 Diagnostics are Services

Diagnostics are more than just test applications. Diagnostics create controlled stimuli and monitor, gather, record and analyze information about detected faults, state, status, performance, and configuration. This diverse nature of diagnostics best lends itself to being modeled as a service that launches or enables the components necessary to implement the diagnostic actions requested by the client.

These diagnostic components may be implemented as test applications, monitoring daemons, enablers for built-in diagnostic capabilities, or proxies to some other instrumentation that is implemented outside of WBEM.

2.5 Diagnostics are Applied to Managed Elements

Diagnostics are applied to Managed Elements. By "applied", we mean that a test checks a managed element, a diagnostic daemon monitors a managed element, diagnostic instrumentation is built into the managed element, etc. One of the goals of CIM-based diagnostics is the packaging of diagnostics along with the vendor's deliverable or Field Replaceable Unit (FRU). Thus diagnostics are often applied at that FRU level of granularity.

Diagnostics Services are commonly applied to:

- **Logical Devices**: Most vendor-supplied diagnostics are for add-on peripherals such as adapters and storage media, which fall into the logical device category. In this case there is clear correspondence between the diagnostic's scope and a CIM defined logical device class.
- **Collections**: Some vendors may choose to apply diagnostics to a collection that represents the aggregated functionality of a managed element. This is supported in CIM by CIM_Collection, which describes an aggregation of managed elements. Since CIM_Collection is a managed element it can be associated to a diagnostic service.
- **Systems**: Not all diagnostic use cases have coverage that corresponds to logical devices or simple collections of distinguishable CIM-modeled devices. Some diagnostic services are often best applied to a system as a single functional unit or as a collection of miscellaneous devices that are scoped to it as a FRU. Some examples are:
    1. System stress tests and monitors which measure aggregate system health.
    2. Miscellaneous, non-modeled, or baseboard devices that are often best viewed as part of a system level FRU.

Comment [JS1]: Needs to more closely align with the CIM definition of a system.

3. Controllers that are part of an internal system bus structure may not be independently diagnosable and must be tested by proxy through another logical device. In this case the controller is an embedded, indistinguishable component that contributes to the overall system health.

- **Other Services**: Diagnostic services may also be applied to other non-diagnostic services. These diagnostics may be used to insure the reliability of the associated service.

2.6 Generic Framework

Diagnostic services should share the semantics of the model regardless of whether the service launches tests, starts a monitoring agent, or enables instrumentation. They should share the same mechanisms for publishing, method execution, parameter passing, message logging, and reporting FRU information.

The diagnostics model also leverages other areas of the CIM model to provide extended diagnostic capabilities rather then introducing diagnostic centric mechanisms. Examples are the "jobs" model for monitoring, the "log" model for capturing information, and effective utilization of the logical and physical models.

2.6.1 Diagnostic Control

Diagnostic clients may need to control and monitor the status and progress of the diagnostics elements launched by the service provider to implement a service request. This control and monitoring capability is achieved in a generic manner utilizing the CIM job and process model. The elements launched by the diagnostic service can be collectively controlled and monitored through an instance of ConcreteJob returned by the diagnostics start method in the diagnostic service. In CIM V2.9, diagnostics leverages this portion of the CIM System model as is, without any diagnostic-specific sub-classing.

2.6.2 Diagnostic Logging & Reporting Assumptions

Diagnostics require the ability to record information about detected faults, state, status, performance, and configuration of both the diagnostic components launched and the relevant managed elements. This information may be gathered dynamically at checkpoints while the diagnostic service is active for concurrent analysis or after the service is complete for post mortem analysis. In CIM V2.9, diagnostics utilize a log to record the information that diagnostic service applications, agents, and instrumentation deem relevant.

In the future, the diagnostic model will link up with planned service models that standardize error codes, indications and trouble tickets in order to integrate CDM diagnostics into WBEM-based industry standard diagnostic policies and RAS use cases. See the DMTF Support WG and CompTIA initiatives for further information.

2.6.3 Localization

Localization refers to the support of various geographical, political, or cultural region preferences, or locales. A client may be in a different country from the system it is querying and would prefer to be able to communicate with the system using its own locale. There are inherent differences to be reckoned with, such as language, phraseology, currency, and many cultural oddities.

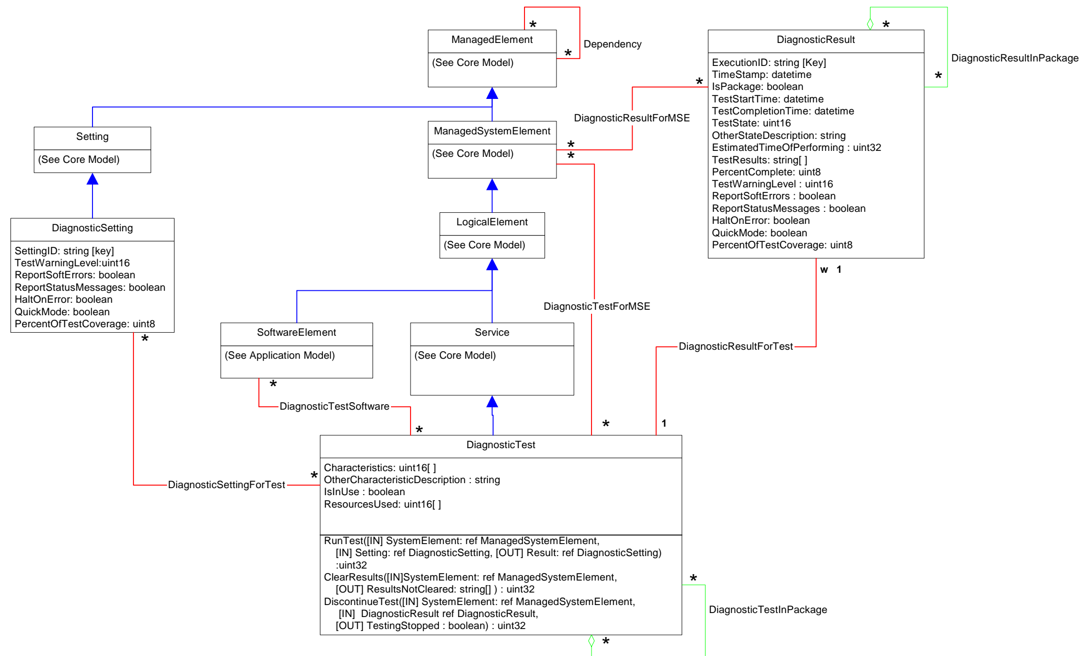
There is no localization support in CIM prior to Version 2.9. Since diagnostics relies on precise reporting of system status and problem data in a user-centric environment, localization is critical. In V2.9, we introduced schema extensions to allow a client to query a diagnostic service for supported locales and to specify the desired locale via a DiagnosticSetting object. The change was written as generically as possible, specifically supporting diagnostics with the intent that it be generalized for broader use in the future.

A new class, CIM_LocalizationCapabilities : CIM_Capabilities was introduced with properties publishing the supported input and output locals. A Locales[] property is added to the DiagnosticSetting class (for passing to the service) and the DiagnosticServiceRecord class (for local identification of the resultant logs).

3 CDMV1

The DiagnosticTest, DiagnosticSetting, and DiagnosticResult classes have been the core of the diagnostic model since CIM V2.3. This section describes this original CDM model, which will be supported until CIM V3.0, along with the minor enhancements made to it in subsequent versions of the CIM schema. The following UML diagram shows the properties and methods relevant to CDM diagnostic clients and providers in CDMV1.

The CDMV1 Diagnostics Model:



3.1 Overview

Although some new features have been added in updates to the CIM schema, the behavior of CDMV1 remains largely unchanged from its introduction in CIM V2.3. Diagnostic tests have DiagnosticSettings passed in, which act like parameters; tests produce a DiagnosticResult, which is a summary report of the test session. The

semantics around these classes have not changed. They have, however, been enhanced with new features as well as some minor changes in order to support integration with the Job model (CDMV2). These changes are included in the descriptions below.

3.2 Model Components

3.2.1 The DiagnosticTest Class

DiagnosticTest is the only diagnostic service class supported in CDMV1. All diagnostic services must be developed in the context of a test. There are several deprecations noted in the following paragraphs. It is important to realize that these deprecated features will continue to be supported until CIM V3.0. They are part of the CDMV1 model.

The properties included in the DiagnosticTest class are designed to be used by a diagnostic client to determine the general effects that are associated with running the test. For example, if a test is going to destroy data or monopolize a resource, the client needs to be aware of this and inform the user or make adjustments to the environment.

The IsInUse and ResourcesUsed properties are deprecated in V2.9 since there are more general ways to expose this information. CIM_Service.Started indicates that the service "is in use" and the ServiceAffectsElement association is the preferred way to surface the resources consumed by the service.

The methods defined for the test class are included to start the test running, stop it prior to normal completion, and clear any stored results that are no longer needed. The RunTest and DiscontinueTest methods are deprecated in V2.9; RunTest is promoted to the new DiagnosticService class (renamed RunDiagnostic) and the DiscontinueTest functionality is achieved using the EnabledLogicalElement.RequestedStatus property.

Even though DiagnosticTest can be directly instantiated, users of the model should subclass and prefix the class name with a unique identifier, including a vendor ID, for example: IBMSG_, for IBM Server Group.

If input parameters are necessary, a DiagnosticSetting instance is created and passed to the test. Results produced by a test are recorded in an instance of the DiagnosticResult class and linked to the test by an instance of DiagnosticResultForTest.

DiagnosticTestInPackage was originally introduced to allow modeling of packages (suites) of tests. This concept introduced so many modeling problems that it was quickly abandoned and left up to the test writer to implement if required. This association class is deprecated in V2.7.

3.2.2 The DiagnosticSetting Class

DiagnosticSetting is derived from CIM_Setting and is used to contain the default and run-specific settings for a given test. Diagnostic service providers publish default settings in an instance of this class (associated to the service by an instance of DefaultSetting) and diagnostic clients are expected to create a new instance and populate it with these defaults with, possibly, user modifications. This new setting object is then passed as an input parameter to RunTest( ). For all properties except SettingID,

LoopParameter, and the deprecated ReportSoftErrors and ReportStatusMessages, the values set by a test client in a DiagnosticSetting object are "qualified" by corresponding properties in DiagnosticServiceCapabilities. If the capabilities do not include support for a setting, then the client must maintain the default for that setting. Any attempt to modify it will not be recognized by the test.

V2.7 added loop controls in the setting class. With this addition, it is possible to loop a test (if supported) under control of a counter, timer, and other loop terminating facilities.

V2.9 also added support for specification of the nature of data being logged by the test through the addition of the LogOptions enumeration. This eliminates the need for some settings that were part of the initial diagnostics model, so these properties are deprecated.

3.2.3 The DiagnosticResult Class

The DiagnosticResult class is used to monitor test progress and receive result data from a specific test instance. When a client executes the RunTest method, a reference to an instance of the result class is returned. If the test finishes very quickly or if it must run synchronously, the result object is not useful for test progress monitoring (through the PercentComplete property). However, if the test is capable of running asynchronously (on its own thread) and publishes its progress, the client can poll this property and relay a progress indication to the user. In addition to PercentComplete, the TestState property can give some progress indication. If TestState is set to any of the completed states ("Passed", "Failed" or "Stopped"), a PercentComplete value less than 100% might indicate an abnormal termination, or some setting that shortened or truncated the test (e.g., HaltOnError). In any case, after the test is complete, the client can read the TestResults property and format the outcome of the test for the user.

> **Note:** Since it is useful to have a record of the settings that produced a particular result, the DiagnosticSetting property values that were passed to RunTest( ) are copied to the result object when it is created.

The ExecutionID key property is used to distinguish between multiple executions of a test on the same managed element. The CDMV2 model recommends that this property be constructed as shown in section **Error! Reference source not found.**, but CDMV1 was introduced without this recommendation and may be implemented as simply an index (i.e., 0,1,2…).

EstimatedTimeOfPerforming is the estimated number of seconds that should be needed to perform the diagnostic test associated with this result. After the test has completed, the actual elapsed time can be determined by subtracting the TestStartTime from the TestCompletionTime.

> **Note:** A similar property is defined in the association, DiagnosticTestForMSE. The difference between the two properties is that the value stored in the association is a *generic* test execution time for the Element and the Test. The value in DiagnosticResult is the estimated time that *this instance with the given settings* would take to run the test. A CIM consumer can compare this value with the value in the association DiagnosticTestForMSE to get an idea what impact their settings have had

on test execution. To get an estimate of time remaining to complete the test, a client could add this value to the start time, and then subtract the current time.

In V2.7, properties were added to record error codes and number of times that code was generated. These error codes may be used for variety of purposes, such as: fault database indexing, field service trouble ticketing, product quality tracking, part failure history, etc. The format of these codes is vendor-specific. It is recommended that hard errors and correctable or recoverable errors be given different codes so that clients with knowledge of the error codes can evaluate correctable, recoverable, and hard errors independently.

Also in V2.7, the addition of looping controls led to the need to count the number of loop iterations that passed and failed. This is relevant in analyzing transitory failures. For example, if all the errors occurred in just one of 100 iterations, the device may be viewed as OK or marginal, to be monitored further rather than failed.

3.3 CDMV1 Usage

The following descriptions have references to model components that were added for CDMV2 but can also be viewed as CDMV1 extensions. These extensions may lead to some confusion in that they were added over a period covering multiple CIM Versions, and early implementations may not have applied theses scenarios since the model wasn't complete. All the reference components appear finally in CIM Version 2.9.

3.3.1 Settings Protocol

To control the operation of a diagnostic service, a CDM provider must satisfy a number of requirements for supporting the diagnostics schema. For each test, the provider publishes a single instance of DiagnosticCapabilties (CIM V2.9) to indicate what features are selectable in a DiagnosticsSetting object. It should provide default settings for the service in an instance of DiagnosticSetting and link the default settings object to the diagnostic service object using the DefaultSetting association. Additionally, a DiagnosticSettingForTest association may be created between this object and the DiagnosticTest object to which the default applies.

> **Note:** DiagnosticSettingForTest is not needed if the recommended implementation is followed. The diagnostic CIM client should create a new instance of DiagnosticSetting that combines the default property values with user input; this is the Setting object passed to the RunTest method. DiagnosticSettingForTest is deprecated in CIM V2.9.

Any CDM client can query the CIMOM for DiagnosticTest instances. After selecting a test to run, the client should check for its default settings and capabilities by querying for the DefaultSetting and ElementCapabilities association instances, and filtering for the instance that references the selected test. The client creates an instance of DiagnosticSetting and populates it with the default settings along with any modifications made by the user, taking into account the published capabilities for that test.

Either the preferred RunDiagnostic( ) method in DiagnosticService (added in V2.9), or the deprecated RunTest( ) method in DiagnosticTest can be used to start a diagnostic test. In either case, a reference to the newly created instance of DiagnosticSetting is passed in as a parameter to the method call. If a setting reference is not passed in, then the CDM provider should use the default setting values.

The diagnostic model utilizes settings to specify parameters standard to all CIM diagnostic services. The diagnostic setting model does **not** utilize any of the methods defined in the Setting class. Instead, as noted here, the diagnostic model passes test settings to the diagnostic service as a parameter to the run method.

When a test's RunTest( ) method is called, the test provider creates an instance of DiagnosticResult. The provider then copies each of the properties in the DiagnosticSetting instance into the DiagnosticResult object. This preserves a record of the settings used for that test execution. When the test has started, a reference to the DiagnosticResult is returned to the client. The client may then use it to check on test progress (PercentComplete, TestState), as well as on the actual results in TestResults[ ].

3.3.2 Looping

Initially, there was no test looping capability included in the model. Looping was left to the client to repeatedly execute the RunTest method. CIM V2.7 adds properties to DiagnosticSetting to allow specification of looping parameters to a diagnostic provider. These properties are actually arrays of controls that may be used alone or in combination to achieve the desired iteration effect.

The LoopControlParameter property is an array of strings that provide parameter values to the control mechanisms specified in LoopControl. This property has a positional correspondence to the LoopControl array property. Each string value is interpreted based on its corresponding control mechanism. There are four different types of controls specified in CIM V2.7:

- Loop continuously
- Loop for N iterations
- Loop for N seconds
- Loop until greater than N hard errors occur

For example, if a client wants to run a test 10,000 times or for 30 minutes, whichever comes first, it could set both count and timer controls into the LoopControl array and would achieve the logical OR'ing of these controls. In another example, if one wants to run a test 1000 times or until 5 hard errors occur, then there are two elements set in this array, one of 'Count' and one of 'ErrorCount'. In the LoopControlParameter array there would be "1000" in the first element and "4" in the second element.

If the LoopControl array is empty, then no looping will take place. Also, if one element is 'Continuous' then no other array elements will have any effect and it is the client that must determine when to stop the test.

In the case of a looped diagnostic, the result that is persisted should contain a summary, and not necessarily a report of each iteration result (depending on LogOptions selected).

Here is an example of what a client might expect to see for diagnostic result information, per result type:

Single Iteration Result	"Test <test name> [passed | failed with error %s]."
Looping Summary Result	"Test <test name> ran <N> times: passes = <j>; failures = <k>."

3.3.3 Result Persistence

Each time a diagnostic test is launched, an instance of DiagnosticResult is created. Originally, CDM placed no policy or control over result object persistence; it was pretty much left as an implementation detail. There are situations (e.g., abnormal termination) that could lead to an accumulation of old, unneeded results. The potential for this type of problem is exacerbated by the introduction of looping.

In general, diagnostic clients should implement a persistence policy and handle storage of results as needed. Providers should be required to persist results only long enough for clients to secure them. This time can vary, however, depending on the environment in which the testing is being performed and unexpected events that may occur. A new setting property in CIM V2.7 allows a diagnostic client to specify how long DiagnosticResults must be persisted by the provider after the running of a DiagnosticTest. This ResultPersistence property is now part of the DiagnosticSetting and DiagnosticResult classes. For each running of a diagnostic test, the client may now specify whether and how long a provider must persist the results of running the test, after the test's completion. In typical use, a client makes one of the following choices:

1. Do not persist results (ResultPersistence = 0x0): The client is not interested in the results or is able to capture the results prior to completion of the test. The provider has no responsibility to maintain any related result objects after test completion.
2. Persist results for some number of seconds (ResultPersistence = <non-zero>): The client needs the results persisted for the specified number of seconds, after which the provider may delete them. The client may delete the results prior to the timeout value being reached.
3. Persist results forever (ResultPersistence = 0xFFFFFFFF): A maximum timeout value will prohibit the provider from ever deleting the referenced result. It is the client's responsibility then to delete them when/if desired.

Note: There is no default timeout value for this property. It has been suggested however, that a five-minute (300 second) timeout, for example, would allow a client enough time to reconnect and query for results if it were accidentally disconnected from a session.

3.3.4 LogOptions for Typed Messages

In CIM V2.3, it was possible for a client to instruct a test provider to enable or disable only two types of result messages destined for the DiagnosticResult.TestResults[ ]

property: soft errors and status messages. This mechanism allowed a client rudimentary control over the amount and type of information returned from a test session.

In CIM V2.9, two improvements increase the client's ability to analyze and control the information returned from test providers. One is the extension of typed messages to specify more type classifications and to include the type in the message header (for analysis). The other is the LogOptions property in DiagnosticSetting that gives a client gating control per message type.

The list of supported message types in includes: test ("hard") errors, soft errors, status, warnings, FRU (e.g. the "Device Under Test") information, debug messages, statistics, corrective actions to be taken, configuration, subtest reports, and reference information. For specific information on each log option and additions to the list, see the MOF file.

The CDM provider indicates that it supports various types of messages by setting values in the DiagnosticServiceCapabilities.SupportedLogOptions array. A client then selects what messages it wants captured by listing those types in the LogOptions parameters of the DiagnosticSetting class. The log options are independent and may be used in combinations to achieve the desired report. The default behavior is for an option to be off/disabled.

3.3.5 Diagnostic Results

In CDMV1, DiagnosticResults are used for two purposes: monitoring test execution status and recording test results.

3.3.5.1 Monitoring Diagnostic Test Progress

In CDMV1, tests log information to DiagnosticResult.TestResults[ ]. The client can monitor diagnostic test information, both dynamically and upon test completion by periodically polling the DiagnosticResult class and looking at this property to see the messages coming from the test. This approach has led to an implementation requirement that the diagnostic provider must create a unique instance of the diagnostic result class and return a reference to that instance to the client. This permits the client to query it while the diagnostic test is running.

The following example illustrates how clients can effectively monitor test status and progress:

- The CDM provider first creates a diagnostic result object before starting its diagnostic test. All key properties are filled out, and the settings that will be applied to the test are copied into this result object.
- The CDM provider also creates the associations DiagnosticResultForMSE and DiagnosticResultForTest so that the client can identify the results that are related to a particular test running on a particular device.
- The provider sets the property TestState to InProgress and sets the current date and time into the TestStartTime property just prior to calling the test.
- For tests that run more than a few seconds, an internal communication mechanism between the test and the provider should be established so that the provider can

update the PercentComplete and the TestResults properties while the test is running. The client can then monitor the test progress.

- After the test completes, the provider sets the TestCompletionTime property to the current date and time and finishes filling out the TestResults[ ] array with messages and a results summary. Finally, it sets the TestState property to the appropriate value: "Passed", "Failed", or "Stopped".

3.3.5.2 Using Typed Messages in TestResults[ ]

The CIM V2.3 System MOF specifies that each string entry in TestResults array in the DiagnosticResult class should be prefixed with a "message header". In CIM V2.9, the description of the TestResults array is modified to specify that the message type must be prefixed to each message header. This allows results to be sorted and searched by message type. The message type naming convention corresponds to the value of the LogOption that enables logging the particular message. The CDMV1 message header has the following format:

LogOption|DateTime|TestName|TestMessage

Where:

- The delimiter, "|", is used to separate each part of this header.
- LogOption = string identical to the LogOption value in DiagnosticSetting that was used to enable logging this message.
- DateTime = the time stamp for the message (CIM data type).
- TestName = Internal test name or current internal subtest name that sent the message.
- TestMessage = free form string that is the "test result".

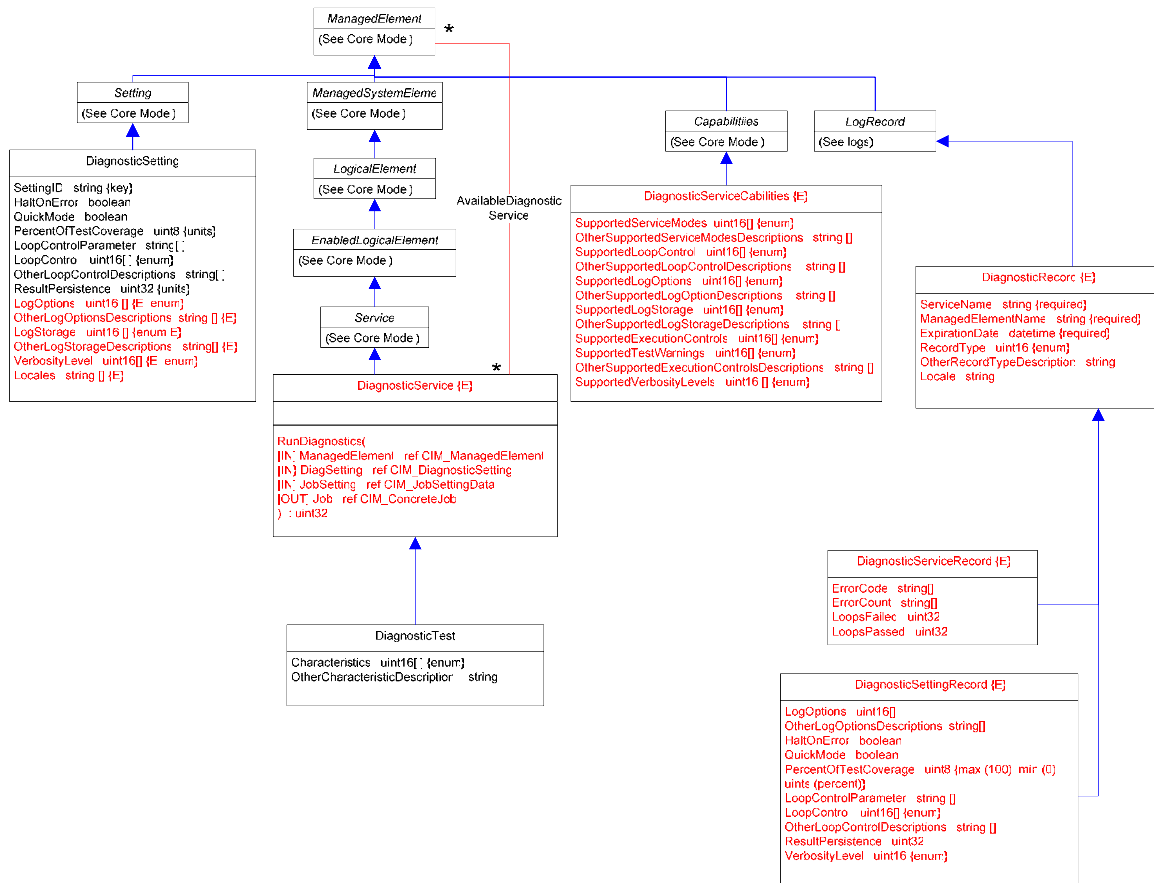
4 CDMV2

Version 1 of the CDM provides a very simple structure for discovery of diagnostic tests, running and monitoring them, and reporting results. Unfortunately, this simplicity introduced limitations by not utilizing the functionality built into other areas of the model. Version 2 of the CDM (CDMV2) introduces a more robust model for diagnostic tools. While CDMV1 was based on a simple settings/tests/results model, CDMV2 supports a more flexible and extendable model based on settings/services/jobs/logs. This will be discussed in the following sections.

The Visio diagram below represents the model components unique to CDM. Other components (e.g., ConcreteJob, Log) can be found by searching the online (www.dmtf.org) documentation.

This document corresponds to CIM V2.9. Always refer to the latest online Visio diagrams and MOF files for the most current version of the model.

The CDMV2 Diagnostics Model:



4.1 Overview

The CDMV2 schema can be partitioned into several major conceptual areas:

- Settings are enhanced and extended with capabilities.
- Diagnostic Services allow extending beyond DiagnosticTest.
- Jobs provide control and monitoring of the diagnostic services launched. (The ConcreteJob class is not shown in the above Visio diagram since there are no diagnostics-specific subclasses. See the System Model schema for ConcreteJob.)
- Logs replace DiagnosticResults as the mechanism for recording information gathered by the DiagnosticService. (The Log class is not shown on the diagram since there are no diagnostics-specific subclasses. Diagnostics will use an aggregation of records to record results.)

4.2 Model Components

4.2.1 Diagnostic Service

The CIM_DiagnosticService class was introduced in V2.9 to accommodate the anticipated extension of the diagnostic model to include additional diagnostic service types. It is felt that there are diagnostic services distinct in their intent and requirements and they should be represented by unique subclasses. CDMV2 currently defines only one subclass of DiagnosticService, DiagnosticTest; other subclasses that have been discussed are exercisers, informational, monitors, and out-of-band test executives. A ServiceType property will likely be introduced when any of these new services are added.

The diagnostic service is associated with the managed element that it tests, monitors, or exercises using the AvailableDiagnosticService : ServiceAvailableToElement class. The managed elements most often targeted by diagnostics services are logical elements such as adapters, storage media and systems, which are realized by the physical model. The physical model contains asset information about these devices and aggregates them into FRUs.

A primary function of the diagnostic service is to publish information about device(s) it services and the effects that running the service has on the rest of the system.

The diagnostic service publishes the following:

- Name & Description of the diagnostic service instance
- Characteristics unique to the diagnostic service type
- Diagnostic capabilities implemented by the diagnostic service
- Default settings that the diagnostic service applies
- Effects on other managed elements

The diagnostic service also provides a method for launching the diagnostic processes that implement the service. The RunDiagnostic( ) method starts a diagnostic service for the

specified ManagedElement (defined using the “ManagedElement” input parameter). How the test should execute, i.e. its settings, is defined in a DiagnosticSetting object. A reference to a setting object is specified using the “DiagSetting” input parameter. The capabilities for the diagnostic service will indicate what settings and other options are supported.

The ServiceAffectsElement class (not shown) represents an association between a service and the managed element(s) that may be affected by its execution. Making this association gives an indication that running the service will pose some burden on the managed element that may affect performance, throughput, availability, etc. This association contains an enumeration, ElementEffects, describing the 'effect' of the service on its associated managed element. The defined values are: "Exclusive Use", "Performance Impact", and "Element Integrity”, replacing the functionality of the DiagnosticTest.ResourcesUsed[ ] property, deprecated in CIM V2.7.

ServiceServiceDependency (not shown) is an association between two services, indicating that the antecedent service is required to be present, required to have completed, or must be absent for the dependent Service to provide its functionality. As an example, one could “order” testing using this association. The actual dependency is published through the TypeOfDependency property specifying "Service Must Have Completed", "Service Must Be Started", or "Service Must Not Be Started".

DiagnosticServiceCapabilities describes the abilities, limitations and/or potential for use of various service parameters and features implemented by the diagnostic service provider.

4.2.2 Diagnostic Jobs

ConcreteJob : Job, introduced in CIM V2.7, provides the properties and methods needed for controlling a diagnostic component (e.g., test application) that was launched by the diagnostic service. It also includes most of the monitoring properties relevant to diagnostics such as percent complete, error code, and job status.

CDM’s use of the ConcreteJob class produces implementations that separate the service monitoring and control functions from the results logging and service publication classes. ConcreteJobs are transient (only exist while the service is running) and contain the controls formerly distributed across unrelated classes.

The DiagnosticService.RunDiagnostic( ) method starts a diagnostic job. This method is invoked with the managed element and settings references as parameters and returns a reference to the instance of ConcreteJob, created to monitor the service.

The ConcreteJob class represents the currently executing service. It is associated with the DiagnosticService that created it via the OwningJobElement association. ConcreteJob contains the following functionality:

1. Job state can be queried.
2. Jobs can be suspended and resumed by invoking the RequestStateChange method of the ConcreteJob class (added in V2.8).

3. Jobs can be associated with specific MEs using the AffectedJobElement association. Within this association is the ElementEffects property. A diagnostic service, when represented by a job, can indicate it is affecting multiple MEs, and indicate the nature of that effect.

The ManagedSystemElement.OperationalStatus[ ] property indicates the current status of the job. Values are generally used in combinations to reveal diagnostic services status.

- OK – Job is running.
- Stopped/OK – Job is suspended.
- Stopped/Completed/OK – Job is complete and operation passed.
- Stopped/Completed/Error – Job is complete and operation failed.
- Stopped/Completed/Degraded – Job "died".
- Aborted – Job stopped by a KillJob method call.
- Supporting Entity in Error – the Job may be "OK" but another element, on which it is dependent, is in error. An example is a network service or endpoint that cannot function due to lower layer networking problems.

4.2.3 Diagnostic Logs

The ultimate goal of a running a diagnostic service is to collect information about the health of a managed element. Clients need to specify how this information is to be recorded in order to be useful in the problem determination process. Logged information may be analyzed by a client dynamically for fault containment and system recovery purposes, but in many situations the information is gathered for post mortem analysis in message logs for use by field service technicians or quality assurance personnel. Examples of the relevant information include:

- **Fault Analysis:** Diagnostic error codes, error frequency, warnings, test time, resource allocation, and percent completion may all be relevant when analyzing failures.
- **Tracking FRU Health:** Diagnostics can query the system to acquire FRU information relevant to diagnostics such as health history, replacement information, and fault signatures.
- **Reproducing Failures.** Diagnostics can query the system to get configuration and state information from the managed elements to which they are applied, from those elements that are impacted by the diagnostic, and from elements that impact the diagnostic itself.

Introduced as a superclass to MessageLog in V2.9, CIM_Log is derived from EnabledLogicalElement and associated to ManagedSystemElement through the UseOfLog association. It has other associations to various storage/file classes and to the LogRecord class.

CIM_MessageLog (now a child of CIM_Log) was designed to double as a container for freeform records with methods for managing them, and an aggregation point for LogRecord objects. It seemed more object-oriented to have separate classes for these two log mechanisms, so RecordLog was introduced as a peer of MessageLog. RecordLog is strictly an aggregation point, having no extrinsic methods. This class fits the diagnostics model in a more efficient manner, as will be seen.

An empty subclass of RecordLog, DiagnosticsLog, was added in order to allow the development of a consolidated record management methodology for diagnostics. A common set of providers for this log and its associated records SHOULD be used to control functions such as record persistence, query support and overall data integrity in a consistent manner.

4.2.3.1 DiagnosticRecord

The CIM_LogRecord : CIM_ManagedElement class can describe the format of entries in a MessageLog, or can be used to instantiate the actual records in the log. The latter approach provides a great deal more semantic definition and management control over the individual entries in a log, than do the record manipulation methods of the MessageLog class. RecordFormat and RecordData were added to LogRecord in V2.9 to simplify data representation.

CIM V2.9 subclassed LogRecord with DiagnosticRecord and two children (DiagnosticServiceRecord and DiagnosticSettingRecord) in order to add some properties unique to diagnostic services and to segregate the Settings (stored in the DiagnosticResult object in CDMV1).

DiagnosticRecord contains the following properties:

1. **ServiceName** is a required string property that is used to identify which service created the record. In order to insure that ServiceName is unique, it should be set to the value of the Name property of the DiagnosticService that caused the record to be created.
2. **ManagedElementName** is a required string property is be used to identify which Managed Element is related to the record. In order to insure that ManagedElementName is unique, it should be set to the value of the ElementName property of the ManagedElement that caused the record to be created.
3. **RecordID** is overridden to specify how the value should be constructed for diagnostic logs. It is a unique identifier representing a single execution of a diagnostic service on an instance of a managed element. Its value should have source correspondence (constructed identically) with the ConcreteJob.InstanceID value (and an index) so that any client knowledgeable of the InstanceID value can data mine a log after all the diagnostic applications and the diagnostic Job objects have expired. Note that, since the ConcreteJob.InstanceID must be globally unique, the diagnostic session's RecordID will also be globally unique if this recommendation is followed. See Section 4.3.2.1.

4. **ExpirationDate** is the datetime when this record should be deleted by the log provider. It is calculated using the ResultPersistence setting property. If a ResultPersistence value is not provided, the ExpirationDate should be set to the current datetime. Once the date has expired, the instance should be deleted as soon as possible.
5. **RecordType** specifies the nature of the data being entered into the DiagnosticServiceRecord. The value in this property should match one of the values indicated in the DiagnosticSetting.LogOptions property that enabled the diagnostics to log messages of the corresponding type (note the ModelCorrespondence).
6. **Locale** specifies the language used in creating the log data.

DiagnosticServiceRecord contains some additional properties relating to error codes and looping.

DiagnosticSettingRecord is used to capture the settings that were used in running the service (as in CDMV1, DiagnosticResult).

4.2.4 HelpService

HelpService was added in V2.8 fill a need for diagnostic on-line help. It was added to the schema in a manner that is readily useful to other parts of the model - a child of CIM_Service. HelpService has properties that describe the nature of the help documents available and a method to request the desired documents. Diagnostic services may publish any form of help they desire, but some implementation recommendations are being developed by the CDM Industry Group[2].

CIM_ServiceAvailableToElement should be used to associate the diagnostic service to its help information.

[2] The CDM Industry Group is currently an ad hoc committee of industry CDM promoters that is developing a set of CDM implementation guidelines. See http://www.intel.com/design/servers/CDM/index.htm.

4.3 CDMV2 Usage

4.3.1 Diagnostic CIM Client Protocol

This section describes the process by which a diagnostic CIM client can configure, initiate, monitor, control and complete a diagnostic service.

4.3.1.1 Query for Services

Clients query the CIMOM for the diagnostic services associated with the managed elements of interest that are scoped to the hosting system. This system scope could be a computer system, unitary device, or represent a network of remotely controlled systems.

Since services and managed elements are related through an association, the client may start its instance query with:

1. the service, traversing the association to find the managed element,
2. the association, then retrieving the antecedent and dependent classes, or
3. the managed element, traversing the association back to the service.

4.3.1.2 Configure the Service

Once the applicable services are enumerated, the client should then discover the configuration parameters for each service. (This can be done for all services up front or individually when a service is actually invoked.)

4.3.1.2.1 Settings

Settings are the runtime parameters that apply to diagnostic services, defined in the DiagnosticSetting : Setting class. Diagnostic services may or may not support all the settings properties, and this support is published using Capabilities (see below).

A diagnostic service should publish its default settings with an instance of DiagnosticSetting, associated by an instance of DefaultSetting. Clients combine these defaults with user modifications (if supported in Capabilities) into a new instance of DiagnosticSetting to be used as an input parameter when invoking the RunDiagnostic( ) method. Passing a null reference will instruct the service to use its default settings.

4.3.1.2.2 Capabilities

Capabilities are described as “abilities and/or potential for use” and, for the diagnostic model, are defined by the DiagnosticServiceCapabilities class. Capabilities are the means by which a service publishes its level of support for key components of the diagnostic model. CIM clients use capabilities to filter settings and execution controls that are made available to users. For example, if a service does not publish a capability for the setting “Quick Mode”, the client application might “gray out” this option to the user.

Clients use the ElementCapabilites association to obtain instances of DiagnosticServiceCapabilities.

4.3.1.2.3 Characteristics

Characteristics[ ] is a property of the DiagnosticTest class which publishes certain information on the inherent nature of the test to the client. It is a statement of the operational modes and potential consequences of running the service. For example, "IsDestructive" indicates that, if this service is started, it will cause some negative system consequences. These consequences can usually be deduced by considering the service, the device upon which the service is acting, and the "affected resources" (below).

Clients should examine the Characteristics[ ] array and use this information to configure the user session and avoid situations that would obviate the desired problem determination goals.

4.3.1.2.4 Affected Resources

CDMV1 relies on the ResourcesUsed property of the DiagnosticTest class to publish the system resources that will be affected or consumed by invoking the test. CDMV2 uses the ServiceAffectsElement association to indicate the managed element(s) affected by the diagnostic service and the ElementEffects[ ] property of this class to describe the actual effect.

Clients should traverse this association to determine the system consequences of starting the service.

4.3.1.2.5 Dependencies

It is common for a service to depend on other system activity for its successful execution. A diagnostic test example is a NIC device under test that depends on TCP/IP being started. It also may be important to "order" certain tests (SCSI interface test prior to SCSI device test). The ServiceServiceDependency association and its TypeOfDependency property are used to publish these dependencies.

4.3.1.3 Execute the Service

Once a service is chosen for running (considering all the system ramifications discussed in the previous section), it is started by invoking the RunDiagnostic( ) method of the DiagnosticService class. The diagnostic service method provider receives references to the settings and managed element objects to be used in running the service, creates an instance of ConcreteJob and returns a reference to it.

4.3.1.3.1 Starting a Job

A diagnostic Job is launched in the following manner:

- The diagnostic service provider creates an instance of ConcreteJob when its RunDiagnostic( ) method is called and creates a globally unique InstanceID key (see Section 4.3.2.1). A reference to the job object is returned as an output parameter.

- The diagnostic service provider creates the associations AffectedJobElement and OwningJobElement so that the client can identify which diagnostic service owns the job and what effects the job will have on various managed elements.
- The Job.DeleteOnCompletion property may be initialized to the value "False" to prevent fast-executing Jobs from being deleted before a client can query for results.

4.3.1.4 Monitor/Control the Service

The client can use the job object to monitor and control the running of the service with the following properties and methods:

- ConcreteJob.JobState – property that communicates the current state of the job. Values are: "New", "Starting", "Running", "Suspended", Shutting Down", "Completed", "Terminated", "Killed", "Exception" and "Service".
- ConcreteJob.RequestStateChange( ) – method used to change the JobState. Options are "Start", "Suspend", "Terminate", "Kill" and "Service".
- Job.PercentComplete – property that communicates the progress of the job.
- Job.KillJob( ) - A method to kill this job and any underlying processes, and to remove any 'dangling' associations. This method is deprecated in V2.8 in favor of RequestStateChange( ).
- Job.ElapsedTime: The time interval that the Job has been executing or the total execution time if the Job is complete.
- Job.ErrorCode: A vendor specific error code. This will be set to zero if the job completed without error.

4.3.1.5 Complete the Service

A service may be terminated using the above controls, or may complete normally, when its work is done. The above controls are monitored by the client to determine when the service is completed.

The useful outcome of running a service is generally presented as a series of messages and data blocks that can be used by the client in the problem determination process. In CDMV1, these were returned in the TestResults[ ] array of the DiagnosticResult class. This has been made more flexible in CDMV2 by using the Log class. Service providers will instantiate subclasses of DiagnosticRecord for logging data returned from the service executable. These are aggregated to a log with the LogManagesRecord association. A client may attempt to read these records by traversing the UseOfLog and LogManagesRecord associations.

4.3.2 Correlation of Records

When information is recorded in a shared log, the life cycle of objects and the ability to distinguish related objects through keys, tags, and instances of associations becomes critical. The diagram below illustrates the relationships between objects in a re-entrant

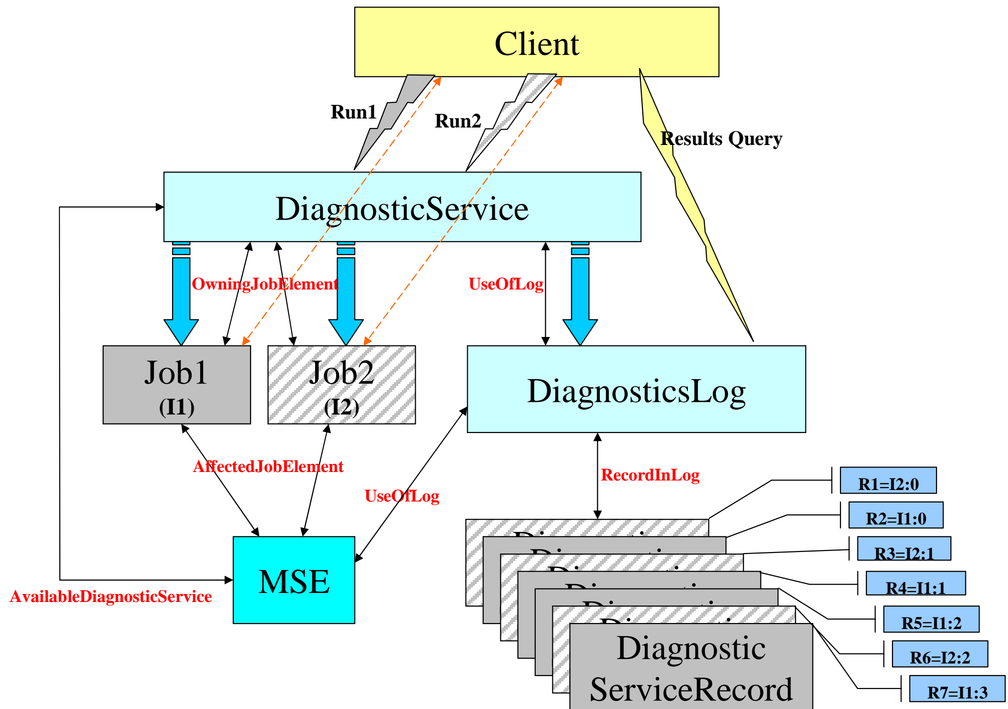
CDM service provider environment using a shared log. It shows a single client initiating two diagnostic services on the same device.

Legend:

Solid Arrows – Instantiations

Line Arrows – Associations

Callouts – Properties (I = InstanceID(key), R = RecordID(key))



The instances for the first request are shown using solid boxes, while the instances for the second request are shown using striped boxes. This diagram also depicts a shared log. Note that the diagnostic model does not dictate whether the message log is shared, unique to each request, or external to the diagnostic service provider. It is recommended, however, that a diagnostic log be firmly associated with the managed element and service that caused it to exist. In this way, a client can more easily query for all records that persist for a particular managed element or service.

The process flows as follows:

1. The client queries for available services and decides to run two instances of a service on a managed element.

2. The client invokes RunDiagnostic( ) with the appropriate settings and receives a reference to Job1 (InstanceID = I1).
3. The service traverses the UseOfLog association to find which log to use. The service is started and Job1 is used for client/service communication.
4. Similar actions take place for the second service instance and Job2 (InstanceID = I2) is created.

   Note that it is an implementation detail whether there are two instances of the service provider running or the provider is able to handle multiple requests of this kind.
5. Now two keyed Jobs are running, generating keyed log records. The next section addresses these keys and how they should be constructed.
6. When a service completes, the Job associated with it should be deleted. The results of the tests are obtained from the log and it's aggregated DiagnosticServiceRecords.

4.3.2.1 CDM Key Structure

Keeping object references distinct is critical in this environment. Object references include key values for uniqueness, and a convention for key construction is often required to guarantee this uniqueness.

4.3.2.1.1 ConcreteJob Keys

The ConcreteJob class contains a single opaque key, InstanceID. The MOF description provides the following guidance for its construction:

"The InstanceID must be unique within a namespace. In order to ensure uniqueness, the value of InstanceID SHOULD be constructed in the following manner: <Vendor ID><ID>. <Vendor ID> MUST include a copyrighted, trademarked or otherwise unique name that is owned by the business entity or a registered ID that is assigned to the business entity that is defining the InstanceID. (This is similar to the <Schema Name>_<Class Name> structure of Schema class names.) The purpose of <Vendor ID> is to ensure that <ID> is truly unique across multiple vendor implementations. If such a name is not used, the defining entity MUST assure that the <ID> portion of the Instance ID is unique when compared with other instance providers. For DMTF defined instances, the <Vendor ID> is 'CIM'. <ID> MUST include a vendor specified unique identifier."

4.3.2.1.2 DiagnsoticRecord Keys

The DiagnsoticRecord class inherits two keys from LogRecord that are useful for distinguishing log records: RecordID and MessageTimestamp. The MOF description provides the following guidance for RecordID construction:

"RecordID, with the MessageTimestamp property, serve to uniquely identify the LogRecord within a MessageLog. Note that this property is different than the RecordNumber parameters of the MessageLog methods. The latter are ordinal values only, useful to track position when iterating through a Log. On the other hand, RecordID

is truly an identifier for an instance of LogRecord. It may be set to the record's ordinal position, but this is not required."

DiagnosticRecord overrides the LogRecord.RecordID property in order to specify its construction for diagnostic logs. The MOF description provides the following guidance for RecordID construction:

> *"In order to ensure uniqueness and provide for efficient mining of DiagnosticRecords that correspond a particular diagnostic ConcreteJob, the RecordID key SHOULD be constructed using the following 'preferred' algorithm: <ConcreteJob.InstanceID>:<n>, where <InstanceID> is <OrgID>:<LocalID> as described in ConcreteJob and <n> is an increment value that provides uniqueness. <n> SHOULD be set to "0" for the first record created by the job and incremented for each subsequent record."*

4.3.3 Using the Physical Model for FRU Identification

The diagnostic CIM client has the ultimate responsibility for obtaining and analyzing state and FRU information. Providers can help through local schema extensions, giving in-house providers a boost in performance and possibly fault analysis capabilities. Such extensions are nonstandard so they cannot be depended upon when leveraging industry providers. The client must be prepared to provide the minimum capabilities of error analysis and FRU reporting. This means providing the ability to trace the test associations to the physical model as far as it is implemented on the system.

The client first queries the association DiagnosticServiceForME which associates the diagnostic test to either UnitaryComputerSystem (in which case you are done) or a type of logical device that could be under NetworkAdapter, Controller, MediaAccessDevice, or StorageExtent.

If the Diagnostic test is associated to a type of logical device, the client needs to query the Realizes association that associates the given logical device to an instance of the static class PhysicalElement that contains the part information.

Finally, the client queries the aggregation associations FRUPhysicalElements that associates the PhysicalElement to the field replaceable unit, FRU class, which contains field replaceable unit information.

It is also recommended that diagnostic services assist with FRU reporting and additional fault information when the test knows about the physical device under test or can obtain fault data. The "Hardware Configuration" record type may be used to post known FRU information to the message log.

5 Future Development

At the time of this writing, CDM has defined and mostly implemented two versions, CDMV1 and CDMV2. Support for CDMV1 will be dropped with CIM V3.0. CDMV2 will continue to be extended and enhanced; there is no plan for a CDMV3 at this time.

5.1 CIM Indications

Indications are useful for generating accurate progress indications, communicating current status, alerting on error, etc. Since CIM Indications are not supported in WMI, the diagnostic modeling group delayed consideration of these features. As CIMOMs that support the current model become more pervasive, we will add the functions that rely on indications into the CDM.

5.2 Interactive Testing

Some diagnostic use cases require interactive job control, for example:

- A test that requires operator intervention (e.g., "Insert loopback plug.").
- Special cases where a diagnostic might want to request information from the diagnostic service before executing the start diagnostic method.
- Interactive debug sessions requiring prompts and responses.

The diagnostic model currently contains a DiagnosticTest.Characteristics = "Interactive", but does not define a mechanism for a client to communicate with the test through the diagnostic service provider. Without such a mechanism it is impossible to implement interactive tests that could be managed by WBEM standard client applications. We have discussed extending jobs to provide support for such interactive tests in a future version of the CIM schema.

Since not all diagnostic providers are expected to support interactivity, a mechanism is necessary to publish the interactive capabilities that the diagnostic service supports, such as:

- No interactivity
- Simple query - the query is a simple display of a message in which the user can only select OK or cancel. An example use case would be a message to the user to insert a loop-back plug before proceeding with running the test.
- Query with data - the query would display instructions to the user to set a value to pass back to the test. The user would type in the value and select OK which would cause the client to write the parameter into the message box property and resume execution of the test. An example use case is an interactive debug mode within the diagnostic test. This would allow debug command parameters to be passed back to the tests through a message box.

5.3 Diagnostics DTD/XSL

The diagnostic CIM client and its GUI program currently handle formatting of data returned from running diagnostic services. It would be desirable, in some environments, to produce a standard form of the output, regardless of the source and user interface. This is most easily achieved by defining a Data Type Definition (DTD) and/or and eXtensible Style Language (XSL) style sheet.

5.4 Services

CIM V2.8 added a superclass to DiagnosticTest in anticipation of additional services that would have different property and method requirements from a standard test. New service types have been identified, but not yet added to the model.

5.4.1 Daemons

In the problem determination environment, daemons are monitor programs that run in the background and look for the existence or emergence of a problem. They are interested in resource contention and over-consumption, predictive failure analysis indications, and any published support for system health events. When the daemon discovers a potential or existing problem, it can alert an administrator and/or initiate some corrective action.

5.4.2 Exercisers

Exercisers are programs written to stress system components to either expose early failures or to cause intermittent problems to occur more often for the purpose of problem determination. They are useful in manufacturing, burn-in, and active debug session environments.

5.4.3 Executives

An executive service is a means to start up external control programs through CIM.

5.5 Logging

Eventually, enabling the direction of selected message types to one or more destinations with various message-logging mechanisms should also prove to be more efficient and versatile. The ability to specify not just a LogType but also its "logical/physical" destination will be a major improvement over the present schema. However, this change has been put off until a future version of the CIM schema to fully comprehend the issues of directing messages to 3rd party providers such as a system event log. It is anticipated that any of the LogOption values (message types) will be able to be specified more than once, in order to direct the same message to more than one message log destination.

5.6 Self Healing and Autonomic Healthcare

The ultimate goal of the CDM is to provide an infrastructure that supports “self-healing” systems. Utilizing the base built in the first and second versions, an AI-based data

consumer could use the diagnostic results along with other CIM data to provide a "self-healing" function.

End of Document