+ **White Paper**

DSP2000 ***Status: Informational***

CIM Diagnostic Model White Paper

CIM Version 2.9

Document Version 1.0 December 14, 2004

Abstract

Diagnostics is a critical component of systems management. Diagnostic services are used 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 on and consistent with the DMTF Common Information Model (CIM), that enable seamless integration of vendor-supplied diagnostic services into system and SAN management frameworks.

In this paper, the motivation behind the 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 (CIM version 2.3) is presented, with extensions introduced beginning with CIM 2.7. 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 computer 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	Reorganized & 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 CIM 2.8 review.
Version 0.93	Ray Pedersen	Cleanup for CIM 2.8 review.
Version 0.94	Ray Pedersen	Cleanup for CIM 2.8 review. Incorporate new CRs for LogOptions and ConcreteJob.
Version 0.95	Ray Pedersen	Update for CIM 2.9 New Logging mechanism.
Version 1.0	Ray Pedersen	Submitted to SysDev for Review

TABLE OF CONTENTS

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 version 2.3 as a simple set of classes representing tests, test settings, and results. Through subsequent implementation, a number of opportunities for improvement were identified in the original model. In addition, CIM has matured, the schema has been extended, and some of these changes are being applied to the CDM to improve versatility and extendibility. A number of major changes occurred as part of the CIM 2.9 release, and phasing over to the new schema will be completed with CIM 3.0.

The CDM version introduced in CIM 2.3 is being referred to as CDMV1 to distinguish it from new concepts introduced in CIM 2.9. CDMV2 encompasses these concepts and will be the only version supported in CIM 3.0.

The purpose of this paper is to describe the CDM schema as it appears in CIM 2.9, distinguish between CDMV1 and CDMV2, and describe future development. The paper provides guidance, where appropriate, to client and provider implementers to reinforce the standardization goal. Guidance for diagnostic test developers is not within the scope of this paper and is being documented by the CDM Industry Group[1].

1.1 Overview

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

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

1.2 Goals

The goals of the CDM are:

- Manageability through Standardization
- Interoperability

[1] The CDM Industry Group is 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.

- Diagnostic Effectiveness
- Global Access
- 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 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 results in a "one size fits all" diagnostic package. Diagnostic management applications can obtain information about which diagnostic services are available, configure and invoke diagnostics, monitor diagnostic progress, control diagnostic execution, and query CIM for information that the diagnostic service gathers. If the CDM methodology is followed, these standard diagnostic packages can be seamlessly incorporated into applications that are implemented as CIM clients. The diagnostic programmer, relieved from the effort associated with satisfying multiple interfaces, can spend more time improving the effectiveness of the tools.

1.2.2 Interoperability

CIM is platform-neutral. Although implementations of CIM (clients, object managers, and providers) do not have to be platform-neutral, that is the goal. To the extent that CIM implementations promote interoperability, so does the CDM. CDM clients and providers can be portable, not only between customers, but also across platforms and in heterogeneous environments.

1.2.3 Diagnostic Effectiveness

Surrounding the CDM infrastructure are the diagnostic tools themselves. When developed to the CDM, the tools become less difficult to deploy and the effectiveness of the entire package can improve. Several factors are at play. Ease of deployment through standardization and interoperability increases availability, thus expanding coverage. Tool developers have the entire WBEM instrumentation database to draw on 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 more details of the device being diagnosed.

1.2.4 Global Access

WBEM provides a framework for managing system elements across distributed environments, enabling the CDM to potentially service systems without regard to locale. This potential facilitates cost-effective serviceability scenarios and warranty-expense reduction.

1.2.5 Life-Cycle Applicability

The CDM is designed to be applicable at all stages in a product's life cycle, from system development and testing to manufacturing, end users, service, and warranty repair.

1.3 Who Should Read This Paper

This paper was prepared to help CDM client and provider developers understand the CIM components of the Common Diagnostic Model and other areas of the model that fulfill the requirements of a comprehensive problem-determination methodology for modern computer systems. Anyone planning to create CDM-compliant diagnostic tools should read it.

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

This paper deals primarily with the CDM architecture. The CDM also includes implementation standards to promote OEM/vendor interoperability and code reuse. 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 also addresses issues related to compliance. Tools are being developed to verify CDM compliance, and procurement processes will likely include such testing.

1.4 CDM Versions

CDM version 1.0 (CDMV1) was introduced in CIM 2.3. It has been enhanced in subsequent versions of the CIM schema. The model components peculiar to CDMV1 will be deprecated prior to the introduction of CIM 3.0, at which time support for CDMV1 clients and providers will be discontinued.

CDM version 2.0 (CDMV2) was introduced with CIM 2.9. The settings/test/results concept is still present, but it is modeled using services, jobs, and logs.

1.5 Background Reference Material

The following background reference material is available:

- Original white paper, *A Diagnostic Model in CIM*, at http://www.dmtf.org/educ/whit.html
- CIM Tutorial at http://www.dmtf.org/spec/cim_tutorial/
- CIM Schema at http://www.dmtf.org/standards/index.php
- 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 2.3)
CDMV2	Version 2 of the CDM (based on CIM 2.9)
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 (contrast with "managed element," which is the generic form).

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

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

Arrays include brackets [ ] 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_" is often omitted from class names for brevity and readability.

2 Modeling Diagnostics

The Common Diagnostic Model (CDM) extends the CIM schema to cover the management of diagnostics, including diagnostic tests, executives, monitoring agents, and analysis tools. The objective of diagnostic integration into CIM is to provide a framework in which industry-standard building blocks that contribute to the ability to diagnose and predict the system's health can seamlessly integrate 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 CDM clients) and diagnostic providers. Diagnostic providers register the classes, properties, methods, and indications that they support with the CIM object manager (CIMOM). 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 action establishes communication between the discovered diagnostic providers and the management 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 assume 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 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 the diagnostic provider supports indications

2.3 Backward Compatibility

The CIM 2.9 diagnostic model (CDMV2) creates some parallel semantics to the CDMV1 and extends the model to provide additional semantics. To make these extensions cleanly, some parts of the diagnostic model were deprecated in favor of more scalable approaches.

These deprecations will be supported in the CIM schema until version 3.0. Provider developers should implement the new CIM 2.9 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 CIM 3.0 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. Because of its diverse uses, diagnostics is best 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. “Applied” means that a test checks a managed element, a diagnostic daemon monitors a managed element, diagnostic instrumentation is built into the managed element, and so on. One of the goals of CIM-based diagnostics is the packaging of diagnostics with the vendor’s deliverable or Field Replaceable Unit (FRU). Thus diagnostics are often applied at the FRU level of granularity.

Diagnostic services are commonly applied to:

- **Logical Devices**: Most vendor-supplied diagnostics are for add-on peripherals such as adapters and storage media. In this case clear correspondence exists 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. Because 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 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:
    - — System stress tests and monitors that measure aggregate system health
    - — Miscellaneous, non-modeled, or baseboard devices that are often best viewed as part of a system-level FRU
    - — Controllers that are part of an internal system bus structure and may not be independently diagnosable but 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 ensure the reliability of the associated service.

2.6 Generic Framework

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

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

2.6.1 Diagnostic Control

CDM clients may need to control and monitor the status and progress of the diagnostics elements that the service provider launches to implement a service request. Clients achieve this control and monitoring capability in a generic manner by using the CIM job and process model. The elements launched by the diagnostic service can be collectively controlled and monitored through an instance of ConcreteJob that is returned by the diagnostics start method in the diagnostic service.

2.6.2 Diagnostic Logging and Reporting Assumptions

Diagnostics require the ability to record information about detected faults, state, status, performance, and configuration of the diagnostic components and the managed elements. This information can 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 2.9, diagnostics use a log to record relevant information from diagnostic service applications, agents, and instrumentation.

In the future, the diagnostic model will connect with planned service models that standardize error codes, indications, and trouble tickets in order to integrate CIM 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. Inherent differences, such as language, phraseology, and currency, must be considered.

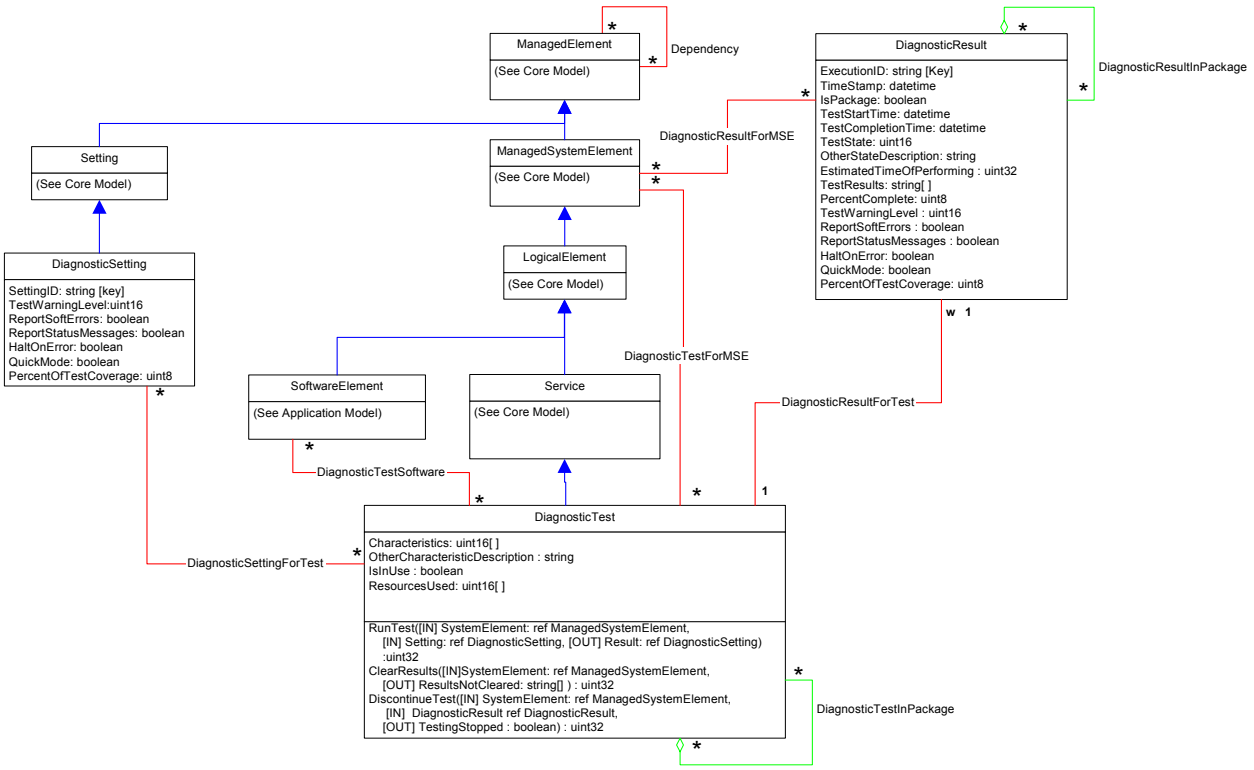
Prior to version 2.9, CIM provided no localization support. Because diagnostics relies on precise reporting of system status and problem data in a user-centric environment, localization is critical. CIM 2.9 introduced schema extensions to allow a client to query a diagnostic service for supported locales and to specify a locale through 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 was 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 2.3. This section describes this original CDM model, which will be supported until CIM 3.0, 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 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 2.3. Diagnostic tests pass in DiagnosticSettings, 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. The classes have, however, been enhanced with new

features and some minor changes to support integration with the Job model (CDMV2). These changes are included in the following descriptions.

3.2 Model Components

This section contains descriptions of the classes added to CIM to support version 1 of the diagnostic model.

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. Several deprecations are noted in the following paragraphs. These deprecated features, which are part of the CDMV1 model, will be supported until CIM 3.0.

> **Note:** Diagnostic services are always considered to be enabled and started. The state controls provided in EnabledLogicalElement are not supported and the service state attributes will be set to their default values. RunTest() can always be invoked and will return the appropriate non-zero return code if the service is not available.

A CDM client uses the properties included in the DiagnosticTest class to determine the general effects 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 methods defined for the test class are included to start the test, stop it prior to normal completion, and clear any stored results that are no longer needed

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 many modeling problems and was abandoned. This association class is deprecated in CIM 2.7.

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 CDM clients 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 particular setting, the client must maintain the default for that setting.

CIM 2.7 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.

CIM 2.9 added support for specification of the nature of data being logged by the test through the addition of the LogOptions enumeration. This support 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 monitors test progress and receives 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 quickly or if it must run synchronously, the result object is not useful for monitoring test progress (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 (for example, HaltOnError). After the test is complete, the client can read the TestResults property and format the outcome of the test for the user.

> **Note:** Because 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 distinguishes between multiple executions of a test on the same managed element.

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 determine the 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 CIM 2.7, properties were added to record error codes and the number of times that a 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, and so on. 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 CIM 2.7, the addition of looping controls led to the need to count the number of loop iterations that passed and failed. This information 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 refer to model components that were added for CDMV2 but can also be viewed as CDMV1 extensions. These extensions were added over a period covering multiple CIM versions, and early implementations may not have applied these scenarios because the model was not complete. All of the referenced components appear finally in CIM 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 2.9) 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 CDM 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 2.9.

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 and any modifications made by the user, taking into account the published capabilities for that test.

Either the preferred RunDiagnostic( ) method in DiagnosticService (added in CIM 2.9) 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 as a parameter to the method call. If a setting reference is not passed, the CDM provider should use the default setting values.

The diagnostic model uses settings to specify the parameters that are standard to all CIM diagnostic services. The diagnostic setting model does not use any of the methods defined

in the Setting class. Instead, 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, thus preserving 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, no test looping capability was included in the model. Looping was left to the client to repeatedly execute the RunTest method. CIM 2.7 added 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. Four types of controls are specified in CIM 2.7:

- 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 to achieve the logical OR of these controls. In another example, if a client wants to run a test 1000 times or until 5 hard errors occur, then two elements are set in this array, one of 'Count' and one of 'ErrorCount'. In the LoopControlParameter array, "1000" would be in the first element and "4" in the second element.

If the LoopControl array is empty, no looping takes place. Also, if one element is 'Continuous,' no other array elements have any effect, and the client 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). Following 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, which was left as an implementation detail. Some situations (such as abnormal termination) 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, CDM 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 2.7 allows a CDM client to specify how long a provider must persist DiagnosticResults 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:

- 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.
- 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.
- Persist results forever (ResultPersistence = 0xFFFFFFFF): A maximum timeout value prohibits the provider from deleting the referenced result. The client is responsible for deleting them.

Note: No default timeout value exists for this property. However, a five-minute (300-second) timeout, for example, might 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 2.3, a client could 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 2.9, the DiagnosticSetting.LogOptions property was added to greatly extend the list of message types that the client could specify. The set of supported message types is extensible; see the MOF for the most current list.

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, dynamically and upon test completion, by polling the DiagnosticResult class and looking at this property to see the messages coming from the test. This approach requires the diagnostic provider to create a unique instance of the DiagnosticResult class and return a reference to that instance to the client. This instance permits the client to query the DiagnosticResult class while the diagnostic test is running.

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

1. The CDM provider 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.
2. The CDM provider 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.
3. The CDM provider sets the property TestState to InProgress and sets the current date and time in the TestStartTime property just prior to calling the test.
4. For tests that run more than a few seconds, an internal communication mechanism between the test and the provider is 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.
5. 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, the provider sets the TestState property to the appropriate value: “Passed”, “Failed”, or “Stopped”.

3.3.5.2 Using Typed Messages in TestResults[ ]

The CIM 2.3 System MOF specifies that each string entry in the TestResults array in the DiagnosticResult class should be prefixed with a “message header.” In CIM 2.9, the description of the TestResults array is modified to specify that the message type must be prefixed to each message header. This modification 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 of the particular message. The CDMV1 message header has the following format:

LogOption|DateTime|TestName|TestMessage

The parts of this header are defined as follows:

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

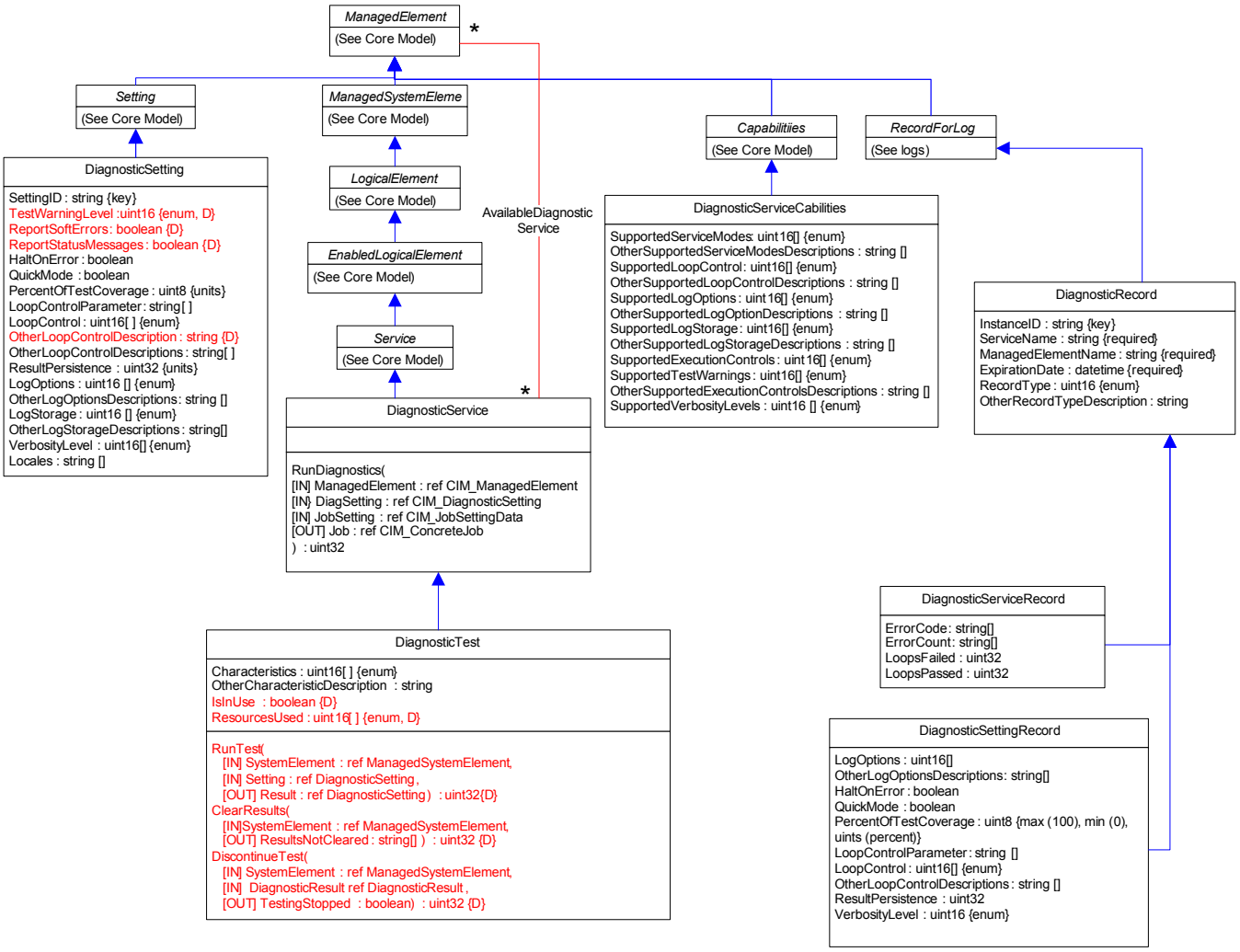
4 CDMV2

Version 1 of the CDM provides a very simple structure for discovering diagnostic tests, running and monitoring them, and reporting results. This simplicity introduced limitations by not using the functionality built into other areas of CIM. 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.

The following UML diagram represents the model components unique to the CDM. You can find other components (for example, ConcreteJob and Log) by searching the online documentation at www.dmtf.org.

This document corresponds to CIM 2.9. Always refer to the latest online 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 (discussed in section 4.3).
- Diagnostic services allow extending beyond DiagnosticTest.
- Jobs provide control and monitoring of the diagnostic services launched. (The ConcreteJob class is not shown in the preceding diagram because it has 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 in the preceding diagram because it has no diagnostics-specific subclasses. Diagnostics use an aggregation of records to record results.)

4.2 Model Components

This section contains descriptions of the classes added to CIM to support version 2 of the diagnostic model.

4.2.1 Diagnostic Service

The CIM_DiagnosticService class was introduced in CIM 2.9 to accommodate the anticipated extension of the CDM to include additional diagnostic service types. Diagnostic services that are distinct in their intent and requirements 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.

The AvailableDiagnosticService : ServiceAvailableToElement class associates the diagnostic service with the managed element that it tests, monitors, or exercises. The managed elements most often targeted by diagnostic 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 the device(s) that it services and the effects that running the service has on the rest of the system.

The diagnostic service publishes the following information:

- Name and 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 for the specified ManagedElement (which is defined using the ManagedElement input parameter). How the test should execute (that is, 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 indicate what settings and other options are supported.

> **Note:** Diagnostic services are always considered to be enabled and started. The state controls provided in EnabledLogicalElement are not supported and the service state attributes will be set to their default values. RunDiagnostic() can always be invoked and will return the appropriate non-zero return code if the service is not available.

The ServiceAffectsElement class (not shown in the CDMV2 diagram) represents an association between a service and the managed elements that may be affected by its execution. This association indicates that running the service will pose some burden on the managed element that may affect performance, throughput, availability, and so on. 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 2.7.

ServiceServiceDependency (not shown in the CDMV2 diagram) 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 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 2.7, provides the properties and methods needed for controlling a diagnostic component (for example, 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.

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 through the OwningJobElement association. ConcreteJob contains the following functionality:

1. The job state can be queried.
2. Jobs can be suspended and resumed by invoking the RequestStateChange method of the ConcreteJob class (added in CIM 2.8).
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 –Job may be "OK" but another element, on which it depends, 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 running a diagnostic service is to collect information about the health of a managed element. Clients specify how this information needs 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 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 CIM 2.9, 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 RecordForLog class.

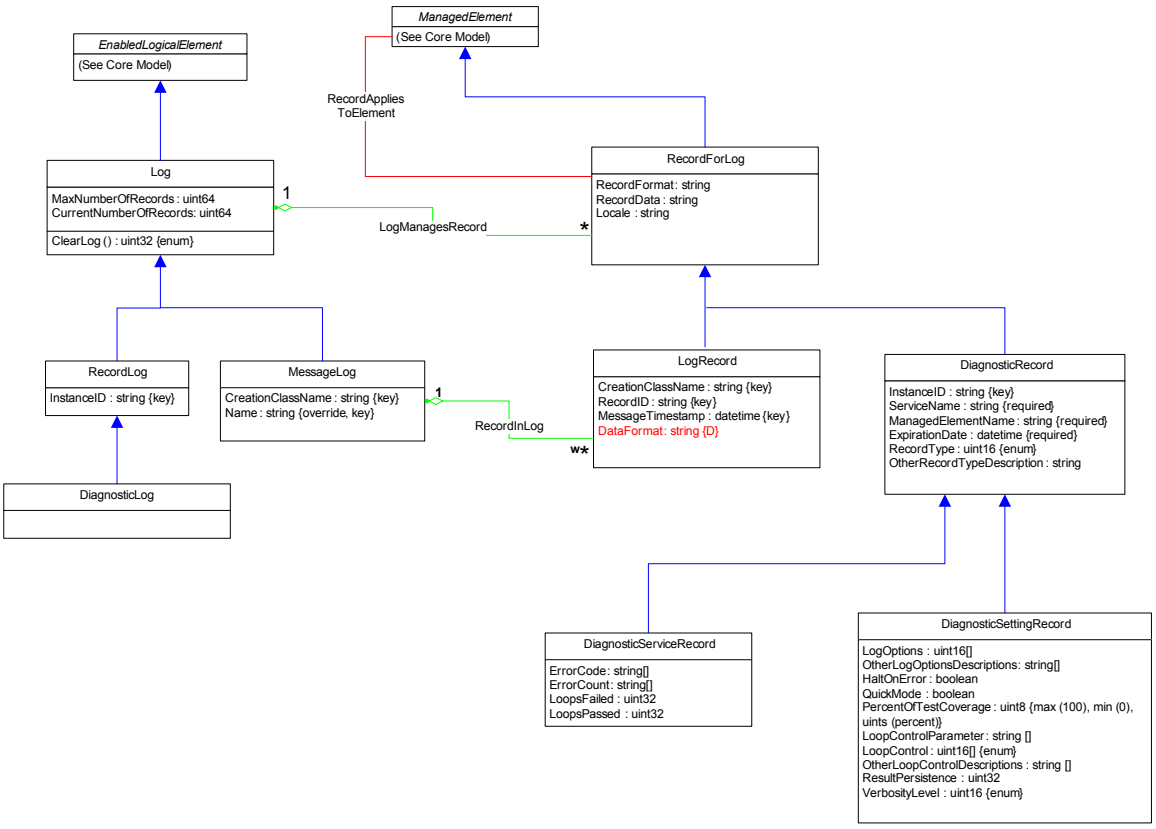
CIM_MessageLog (now a child of CIM_Log) was designed to act both as a container for freeform records with methods for managing them and as an aggregation point for LogRecord objects. Having separate classes for these two log mechanisms was more object-oriented, so RecordLog was introduced as a peer of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

An empty subclass of RecordLog, DiagnosticsLog, was added 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

CIM_RecordForLog : CIM_ManagedElement is an abstract parent of LogRecord and DiagnosticRecord, introduced in CIM 2.9 to allow these record classes to have a different key structures. LogRecord remains Weak to MessageLog (via the RecordInLog aggregation) and has the propagated keys from MessageLog. DiagnosticRecord has the simpler, preferred, InstanceID key and uses the (non-Weak) LogManagesRecord aggregation, defined at the CIM_Log level.

The CDMV2 Logging Model



CIM 2.9 subclasses RecordForLog with DiagnosticRecord and its 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:

- **InstanceID** is the only key for this class. 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 because 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.
- **ServiceName** is a required string property that identifies which service created the record. To ensure that ServiceName is unique, its value should be set to the value of the Name property of the DiagnosticService that caused the record to be created.
- **ManagedElementName** is a required string property that identifies which managed element is related to the record. To ensure that ManagedElementName

is unique, its value should be set to the value of the ElementName property of the ManagedElement that caused the record to be created.

- **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 value should be set to the current datetime. After the date has expired, the instance should be deleted as soon as possible.
- **RecordType** specifies the nature of the data being entered into the DiagnosticServiceRecord. The value of 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).
- **Locale** specifies the language used in creating the log data.

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

DiagnosticSettingRecord captures the settings that were used in running the service (equivalent to DiagnosticResult in CDMV1).

4.2.4 HelpService

HelpService was added in CIM 2.8 to fill a need for diagnostic online help. It was added to the schema as a child of CIM_Service so that it is readily useful to other parts of the model -. HelpService has properties that describe the nature of the available help documents and a method to request needed documents. Diagnostic services may publish any form of help, but some implementation recommendations are being developed by the CDM Industry Group.

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

4.3 CDMV2 Usage

The following sections describe, at a high level, how the CDMV2 classes should be used to develop effective diagnostic applications in a CIM environment.

4.3.1 CDM Client Protocol

This section describes the process by which a CDM client configures, initiates, monitors, controls, and completes a diagnostic service.

4.3.1.1 Query for Services

Clients query the CIMOM for the diagnostic services that are 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.

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

- the service, traversing the association to find the managed element
- the association, then retrieving the antecedent and dependent classes
- the managed element, traversing the association back to the service

4.3.1.2 Configure the Service

After the applicable services are enumerated, the client discovers the configuration parameters for each service. (This discovery can occur for all services up front or individually when a service is 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 the following section).

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 instructs the service to use its default settings.

4.3.1.2.2 Capabilities

Capabilities are “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 ElementCapabilities association to obtain instances of DiagnosticServiceCapabilities.

4.3.1.2.3 Characteristics

Characteristics[ ] is a property of the DiagnosticTest class that publishes certain information about 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 test 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" (see the following section).

Clients should examine the Characteristics[ ] array and use this information to configure the user session and avoid situations that would undermine the 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 elements 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 test.

4.3.1.2.5 Dependencies

A service may 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 (for example, an SCSI interface test prior to an SCSI device test). The ServiceServiceDependency association and its TypeOfDependency property are used to publish these dependencies.

4.3.1.3 Execute the Service

After the client considers all the system ramifications discussed in the preceding section and chooses a service to run, it commences the service action by invoking the RunDiagnostic( ) method of the DiagnosticService class. The diagnostic service 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:

1. When its RunDiagnostic( ) method is called, the diagnostic service provider creates an instance of ConcreteJob, creates a globally unique InstanceID key

(see section 4.3.2.1), and returns a reference to the job object as an output parameter.

2. 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.
3. 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 and 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 that kills this job and any underlying processes, and removes any dangling associations. This method is deprecated in CIM 2.8 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 property is set to zero if the job completes without error.

4.3.1.5 Complete the Service

A client can use the preceding controls to terminate a service or the service may complete normally when its work is done. The client monitors the controls to determine when the service is completed.

The outcome of running a service is generally presented as a series of messages and data blocks that the client can use in the problem-determination process. In CDMV1, this information was returned in the TestResults[ ] array of the DiagnosticResult class. In CDMV2, the Log class is used. Service providers instantiate subclasses of DiagnosticRecord for logging data that the service executable returns. 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

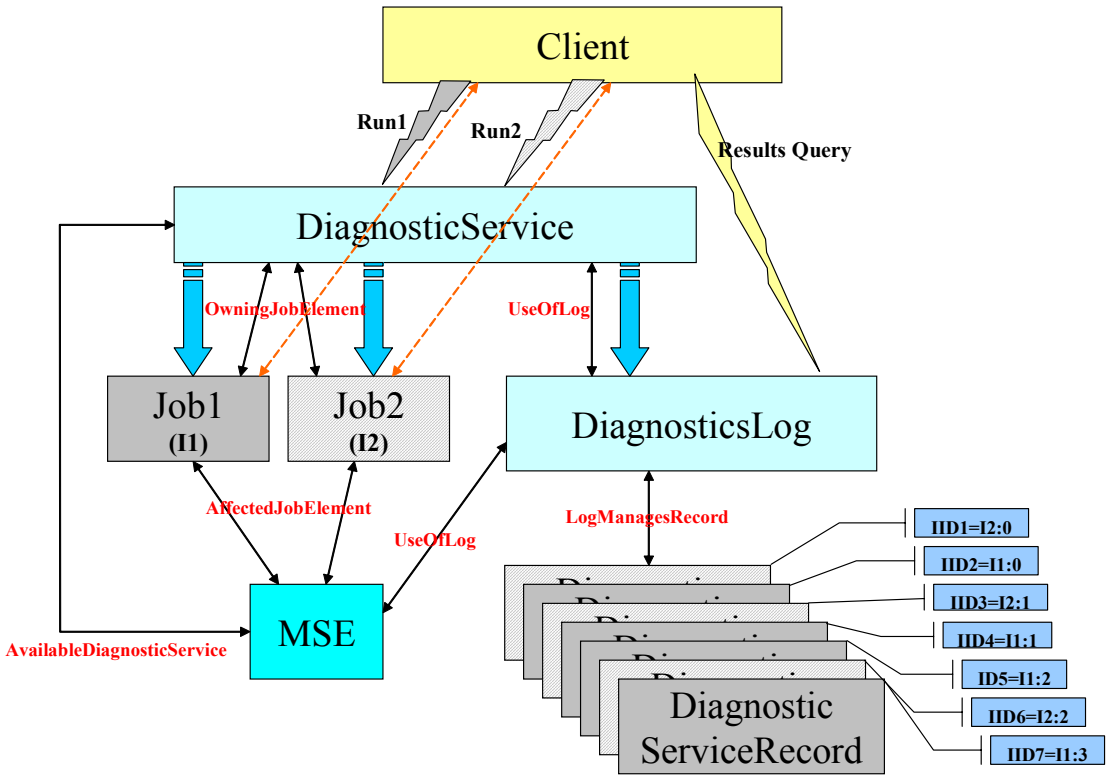
When information is recorded in a shared log, the life cycle of objects and the ability to distinguish related objects through keys, tags, and instances of associations becomes critical. The following diagram illustrates the relationships between objects in a re-entrant CDM service provider environment that uses a shared log. It shows a single client initiating two diagnostic services on the same device.

Legend:

Solid Arrows – Instantiations

Line Arrows – Associations

Callouts – I*n* = ConcreteJob.InstanceID*n*, IID*n* = DiagnosticRecord.InstanceID*n*



The instances for the first request are shown using solid boxes, and 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 provider traverses the UseOfLog association to find which log to use. The service is begun 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. 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 is deleted. The results of the tests are obtained from the log and its 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 Key

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 DiagnosticRecord Key

The DiagnosticRecord class has a single key, InstanceID. It is constructed identically to the ConcreteJob InstanceID key. It is further specified in the Diagnostics Profile Specification that:

"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 < ConcreteJob.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 CDM client is ultimately responsible 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, the client is done tracing the association) 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 association 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 3.0. CDMV2 will continue to be extended and enhanced; no plan exists at this time for a CDMV3.

5.1 Interoperability

Diagnostic services are often complex and could require some client awareness of the nature of the service and the best way to make use of the results obtained by running the service. This situation can be an impediment to the level of interoperability that the DMTF would like to achieve.

First, the DMTF must develop a mechanism for making the services self-describing so that a general, unaware client can (programmatically) understand the purpose and scope of the service. Then, the results obtained by running the service must be able to be interpreted without any prior knowledge of the meaning of messages and codes. Standards need to be developed and integrated with new and existing schema (for example, CIM Error) to achieve these goals.

5.2 CIM Indications

Indications are useful for generating accurate progress indications, communicating current status, alerting on error, and so on. Because 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, DMTF will add the functions that rely on indications into the CDM.

5.3 Interactive Testing

Some diagnostic use cases require interactive job control. For example:

- A test that requires operator intervention (for example, “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” value but does not define a mechanism for a client to communicate with the test through the diagnostic service provider. Without such a mechanism, implementing interactive tests that could be managed by WBEM standard client applications is impossible. DMTF has discussed extending jobs to provide support for such interactive tests in a future version of the CIM schema.

Because 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 message in which the user can select only 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 displays instructions to the user to set a value to pass back to the test. The user types in the value and selects OK, which causes 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, which would allow debug command parameters to be passed back to the tests through a message box.

5.4 Diagnostics DTD/XSL

The CDM client and its GUI program currently handle formatting of the data that is 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) or and eXtensible Style Language (XSL) style sheet.

5.5 Services

CIM 2.8 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.

5.5.1 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 initiate some corrective action.

5.5.2 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.

5.5.3 Executives

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

5.6 Logging

Enabling the direction of selected message types to one or more destinations with various message-logging mechanisms should be more efficient and versatile than the current practice. 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 delayed until a future version of the CIM schema to fully comprehend the issues of directing messages to third-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.

5.7 Self Healing and Autonomic Healthcare

The ultimate goal of the CDM is to provide an infrastructure that supports “self-healing” systems. Using the base built in the first and second versions, an AI-based data consumer could use the diagnostic results with other CIM data to provide a “self-healing” function.

End of Document