RC22840 (W0307-042) July 7, 2003
Computer Science

IBM Research Report

Implementing flexible Data Collection and Aggregation for Performance Management with the CIM Metrics Model

Alexander Keller [1], Oliver Benke [2], Markus Debusmann [3], Andreas Köppel [4], Heather Kreger [5], Andreas Maier [2], Karl Schopmeyer [6]

[1] IBM T.J. Watson Research Center, Yorktown Heights, NY, USA,
[2] IBM Germany Lab, eServer Systems Management, Böblingen, Germany,
[3] FH Wiesbaden, Dept. of Computer Science, Wiesbaden, Germany,
[4] SAP AG, Systems Management, Walldorf, Germany,
[5] IBM Corporation, Research Triangle Park, NC, USA,
[6] The Open Group,

Implementing flexible Data Collection and Aggregation for Performance Management with the CIM Metrics Model

Alexander Keller[1], Oliver Benke[2], Markus Debusmann[3],
Andreas Köppel[4], Heather Kreger[5], Andreas Maier[6], Karl Schopmeyer[7]

[1] IBM T.J. Watson Research Center, Yorktown Heights, NY, USA, *alexk@us.ibm.com*
[2,6] IBM Germany Lab, eServer Systems Management, Böblingen, Germany, {*benke*|*maiera*}*@de.ibm.com*
[3] FH Wiesbaden, Dept. of Computer Science, Wiesbaden, Germany, *m.debusmann@computer.org*
[4] SAP AG, Systems Management, Walldorf, Germany, *andreas.koeppel@sap.com*
[5] IBM Corporation, Research Triangle Park, NC, USA, *kreger@us.ibm.com*
[7] The Open Group, *k.schopmeyer@opengroup.org*

***Abstract*— We describe new extensions to the CIM Metrics Model, termed *BaseMetrics Submodel*, whose scope is to define schema extensions capable of specifying and subsequently instantiating new performance measurement data at the runtime of a system. The model has been developed by the Metric Extensions Working Group of the Distributed Management Task Force (DMTF) in which the authors actively participate. The BaseMetrics submodel has been recently adopted by the CIM Technical Committee and is part of the new version 2.7 of the CIM schema. In addition, we present an extension to the BaseMetrics submodel that allows the definition and aggregation of arbitrary performance data at runtime to address the requirements of service level agreements and workload management systems. Two examples illustrate the applicability of the model to real-life data collection and aggregation scenarios in distributed computing environments.**

I. Introduction and Motivation

The Common Information Model (CIM) [1], [2] is a conceptual framework for describing managed resources and management information in enterprise and service provider environments. CIM consists of over 1000 classes and associations defining managed resources that have been developed over the last 5 years by the Distributed Management Task Force (DMTF). In its various *Common Schemas*, CIM defines a taxonomy of various types of managed resources, such as systems, applications, networks and network elements. Management information that applies to all types of managed resources is defined in the *Core Schema* from which all the Common Schemas are derived. CIM provides a consistent definition and structure of data, using object oriented techniques, and it is based on UML.

While the vast majority of the CIM standardization effort has been devoted to defining and improving the CIM models of resource instrumentation, additional work on *management services* is now taking place. Management services are generic, domain-independent functionality that should be defined and implemented only once; the best-known examples to date are the ISO–OSI Management Functions [3] for Network and Systems Management. In CIM, examples of this kind of functionality are services for specifying and applying management policies, Service Level Agreements (SLAs), Service Level Objectives (SLOs), and capturing and manipulating managed resource metrics. Metrics are numerical information (counters and gauges) that indicate how a managed resource is performing.

Generally, CIM support for performance management comprises the information that developers have defined at the design time of a managed resource model, typically as properties in subclasses of `CIM_StatisticalData`. Having such static definitions of performance measurement data makes it difficult to add new attributes to a class without disrupting existing management systems because they assume that the interface of a managed object (its properties and operations) remains constant once it is defined. On the other hand, there is a need to dynamically define new metrics during the runtime stage of managed resources, as illustrated by the following scenarios:

Dynamic service provisioning: IT resources are allocated and provisioned for specific applications and resource aggregations. Autonomic and Grid technologies [4] are pushing provisioning to be done on a per-request basis. This means that the actual IT resources used by an application may vary throughout the lifetime of the application. Consequently, the metrics from different IT resources need to be associated with a service when it is provisioned.

Service Level Management: Once a Service Level Agreement (SLA) has been negotiated between a customer and a service provider, its definitions – comprising, among other, the service parameters and their allowable value ranges – are deployed to the managed resources and subsequently instantiated. To ensure a maximum of accuracy, an SLA often contains the precise definitions on how lower-level resource data is aggregated into higher-level service parameters. This information needs to be sent to already deployed managed resources whenever a new SLA has been negotiated. In a second step, the SLA monitoring infrastructure needs to be automatically provisioned to enforce the SLA.

Metric aggregation for Workload Management: New virtual IT resources may be defined that aggregate other IT resources: database clusters, connection pools and portal servers wrapping web application servers are examples for such virtual resources. It is necessary to promote some of the metrics defined for the underlying resources to the new virtual resource to appropriately describe the performance of a virtual resource.

All of these examples have in common that new performance management information is defined *after* the underlying resources (and the management agents that surface their data) have been deployed and provisioned. More specifically, new information is defined – and withdrawn – at the runtime of managed resources, when the static instrumentation schemas cannot be modified to accommodate new data.

With the availability of the CIM Metrics Model in the recently released version 2.7 of CIM, there is now a means to introduce such new performance management information at the runtime of a managed resource. In this paper, we describe the underlying concepts and design principles of the CIM Metrics Model and give implementation examples that have served as proof-of-concepts for the adoption of the model.

The paper is structured as follows: Section II describes the design principles and underlying concepts of the current CIM Metrics Model, which is presented in section III. Section IV presents the IBM z/OS Resource Management Facility, a performance data collection system for mainframe clusters, and how the CIM Metrics Model could be used to expose its data via CIM. Section V gives an overview of the Metric Aggregation and Summarization Model, an extension to the CIM Metrics Model, targeted for inclusion into the upcoming CIM version 2.9, along with a description of its prototype implementation. In section VI, we discuss the applicability of the CIM Metrics Model for monitoring and enforcing SLAs. Section VII concludes the paper and presents future work items.

II. Requirements and Core Concepts

The CIM Metrics Model is built on the concept of the dynamic definition, manipulation, and use of arbitrary metric information for all kinds of objects in the CIM class hierarchy. In this section, we describe the requirements the Metrics Model needs to address.

A. Late Binding of Metrics to any Type of Resource

The development of the BaseMetrics model started with the assumption that metrics could effectively be defined as objects rather than attributes in classes or associated statistical classes. This enables modelers to associate a metric definition and its corresponding values with a managed resource (services, applications, systems, devices, etc.) either at design time or at runtime. In addition, the model should support the definition of arbitrary metrics at the runtime stage of a resource, thus permitting administrators to define new metric data in addition to the measurement data defined by the developer of a managed resource, which is typically represented by properties of the class `StatisticalData`[1] or one of its subclasses. It should be possible to introduce new metrics, metric aggregations and complex calculations as new separate definition and value classes, which are then associated with a resource while it is executing. Extending an existing CIM statistics class with a new property would change the interface of the class, which is not possible at runtime in CIM once the class is loaded into the CIM Object Manager. The Metrics Model, in contrast, separates the definition of the metric from the instance representing the value of the metric.

Another major objective was to create a model so that metrics can be dynamically defined and attached to any kind of CIM object representing a managed resource. Whereas the `StatisticalData` class and its subclasses require their static association to a specific CIM class representing a managed resource (naming properties are propagated from the class representing the managed resource to the class holding its statistical data), the Metrics Model should allow metrics to be dynamically associated with any kind of resource in the CIM class hierarchy. Such late binding offers a number of advantages in the ability to attach metrics to components of the model after these components have been defined: It provides for the definition of flexible, dynamically extensible meta-data with very fine granularity. In addition, it allows publishing the semantics of the measurement data, i.e., providing a definition for the management application to help it understand what the data means.

[1] For CIM classes, we omit the prefix 'CIM_' from their description.

In addition, there are a number of reasons for separating the definition aspects of a metric from its value(s):

A metric definition may apply to several different types of resources. Reusing the same definition class ensures that the metric has the same semantics across different resources.

A management application must be able to understand what all the possible metrics for a resource are, even if the resource is not yet installed, running, or some metrics do not have values.

The meta-data for a metric definition is different from the meta-data for a metric value. In analogy to the `ObjectType` macro in SNMP-based management [5], a metric definition needs an identifier and name, as well as a datatype, a unit, and a flag to indicate which kinds of calculation can be applied to a metric when it is further processed by a management application. Each metric value, in contrast, needs at least an identifier, a name, a link to its definition, as well as a timestamp.

Finally, there may be many values of a metric over time, but the same definition applies to all of them. There is no need to replicate the definition for each value instance. In fact, replicating the information causes doubt that the semantics are the same for all instances of a metric across different resource types.

In addition to these basic principles, we have identified further requirements that existing resource management systems pose on the way metrics should be defined. They are detailed below.

B. Address different Performance Data Access Types

For performance monitoring, we identified three important access types that typical systems management applications might want to use:

1) *Volatile* or *current data*, where a management application would like to see only the most current performance value;
2) *Long-term monitoring*, where a data collector stores collected performance values for future use.
   This data access type typically requires a performance repository, optionally with wrap-around buffers to automatically discard older values, or a mechanism to systematically condense the data over time so that the available granularity is reduced over time.
3) *Asynchronous access* to performance data, where the application subscribes for *events* and gets notified later with an event if the given criteria are met (e.g., threshold exceeded, or application state change). Such a subscription mechanism is currently being designed, but not yet part of the model.

For synchronous access, a *Volatile* property of the class representing a metric value could be used to distinguish between access to current data and long-term monitoring. If it is set to `true`, there is only one metric value instance for a given pair of managed resource and metric definition. If *Volatile* is set to `false`, a time series of performance values can be constructed by creating a separate metric value object for each individual measurement.

C. Drill-down Capabilities for Metrics

An important task in performance management is the ability to "drill down" to the root cause of a given problem. Systems management software can either help the human administrators significantly in navigating to the root cause of the problem, or it may analyze and solve the problem without human intervention. Such problem analysis can be done in four different ways [6]:

1) *Changing the selected metric*: In order to look at the problem from a different view, like asking for CPU utilization instead of memory utilization numbers.
2) *Changing the analyzed resource*: For having a closer look at the resource object representing the database management system instead of looking at the resource object representing the application server which makes use of the DBMS.
3) *Navigating through the time dimension*: For identifying when exactly the problem started to occur and finding out what happened at a specific point in time.
4) *Navigating through a user-defined (”breakdown”) dimension*: For drilling-down on the metric by an arbitrary dimension. Examples are drill-down by delay/wait reason and drill-down by user or user group.

Making use of the breakdown dimension capabilities is comparable to the drill-down operation in data warehouses, where one wants to analyze a given situation in more detail. A simple example is illustrated in figure 1.

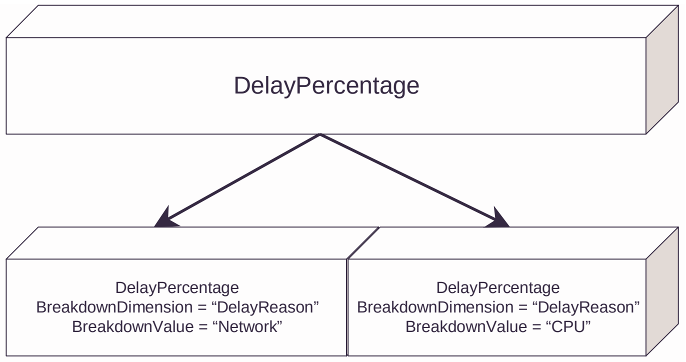
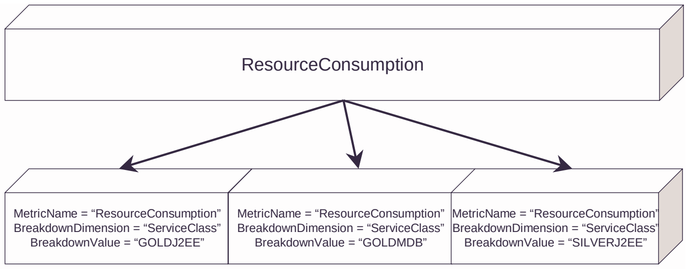

DelayPercentage
DelayPercentage
BreakdownDimension = “DelayReason”
BreakdownValue = “Network”
DelayPercentage
BreakdownDimension = “DelayReason”
BreakdownValue = “CPU”

Fig. 1. Breakdown of a delay metric

Here, the analysis starts with a "delay" metric for a given resource. If, for example, a DBMS experiences delays during 60% of a time interval, the performance analyzer of a workload manager needs to find out why the DBMS is delayed, and which resources should be allocated for the DBMS to speed it up. By using the CIM Metrics model, it should be possible to ask for metrics with breakdown dimension "DelayReason". With a metric definition object, a CIM instrumentation should be able to specify which breakdown dimensions it is able to offer. The instrumentation may reply that the DBMS was delayed 40% of the time due to network congestion and 20% of the time due to CPU resource contention.



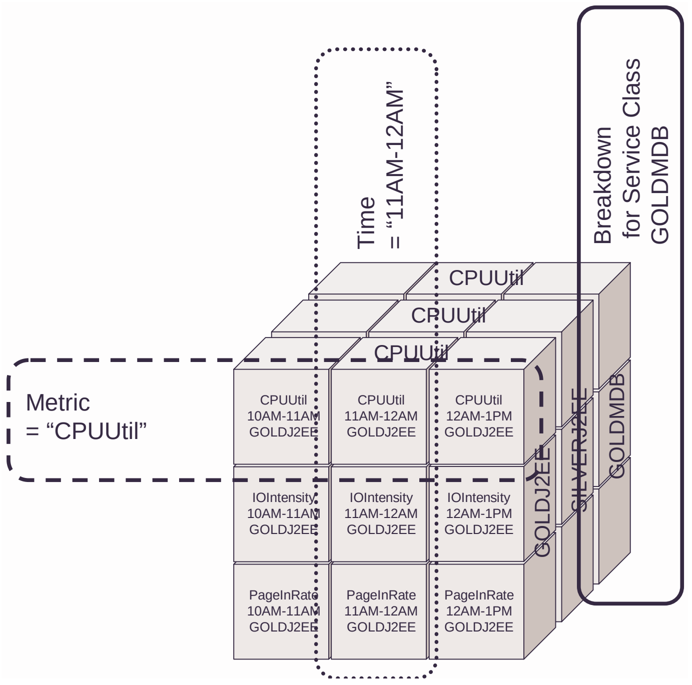
ResourceConsumption
MetricName = "ResourceConsumption" BreakdownDimension = "ServiceClass" BreakdownValue = "GOLDJ2EE"
MetricName = "ResourceConsumption" BreakdownDimension = "ServiceClass" BreakdownValue = "GOLDMDB"
MetricName = "ResourceConsumption" BreakdownDimension = "ServiceClass" BreakdownValue = "SILVERJ2EE"

Fig. 2. Breakdown of a resource consumption metric

For a second example (figure 2), we assume a system capable of grouping transactions by a mixture of user groups and business applications, and where one can define the transaction group goals for response times as service level objectives. If a group of very important customers is using a J2EE application server with a relational database back end, their transactions would be grouped in service class `GOLDJ2EE`.

Breakdown dimensions yield an N-dimensional data hypercube of performance data with dimensions *time*, *resource*, *metric* and various user-defined *breakdown dimensions* (see figure 3). Very simple performance monitors only implement two dimensions: *resource* and *metric*. The *time* dimension is implemented if the performance monitor supports long-term analysis. This requires a performance data repository, which may be a relational database, or another kind of reliable data store. For long term monitoring, there is often support to condense or discard older performance values. The *breakdown dimension* is currently only implemented by some advanced monitors as it is complicated to implement it in an efficient way so it can be used for daily systems management tasks without compromising the performance.

The simplified data cube in figure 3 represents the dimensions of performance management data for a managed resource, in this case a web server. The service classes are called `GOLDJ2EE`, `SILVERJ2EE` and `GOLDMDB`. The available metrics are called `CPUUtil`, `IOIntensity` and



Time = "11AM-12AM"
Breakdown for Service Class GOLDMDB
Metric = "CPUUtil"
CPUUtil
CPUUtil
CPUUtil
CPUUtil 10AM-11AM GOLDJ2EE
CPUUtil 11AM-12AM GOLDJ2EE
CPUUtil 12AM-1PM GOLDJ2EE
IOIntensity 10AM-11AM GOLDJ2EE
IOIntensity 11AM-12AM GOLDJ2EE
IOIntensity 12AM-1PM GOLDJ2EE
PageInRate 10AM-11AM GOLDJ2EE
PageInRate 11AM-12AM GOLDJ2EE
PageInRate 12AM-1PM GOLDJ2EE
GOLDJ2EE
SILVERJ2EE
GOLDMDB

Fig. 3. Breakdown dimensions of performance management data

`PageInRate`. Some of the data values are associated to the time interval `10AM-11AM`, some to the time interval `11AM-12AM` and some to the interval `12AM-1PM`.

For a more detailed discussion of an implementation that uses breakdown dimensions, the reader is referred to [7] and [8].

III. The CIM Metrics Model v2.7

The CIM Metrics Model, recently released as part of CIM version 2.7, is partitioned into two distinct submodels:

- The **Unit of Work** submodel: It provides a means to carry out response time measurements against an application that is instrumented according to the Application Response Measurement (ARM) specification [9]. The corresponding CIM Unit of Work submodel was standardized as part of CIM version 2.5.
- The **BaseMetrics** submodel: This model was recently adopted by the DMTF and released as part of CIM version 2.7. It is the subject of this paper. The Metric Aggregation and Summarization extensions to this model, described in section V, are currently being discussed in the DMTF Metric Extensions Working Group for possible inclusion in CIM version 2.9.

The BaseMetrics model comprises two classes:

- `BaseMetricDefinition`: This is the meta-data for CIM Metrics. An instance of this class is created for each metric that is to be defined.

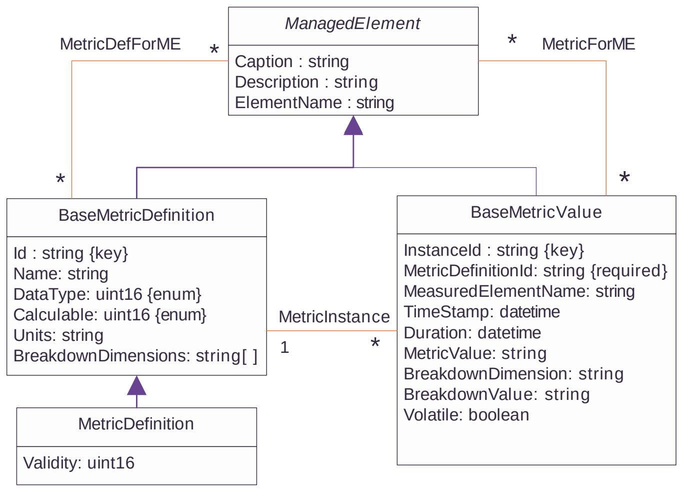


Fig. 4. BaseMetrics Submodel of the CIM 2.7 Metrics Model

- `BaseMetricValue`: This is the value container class for instances of `BaseMetric` information. One metric value is contained in each instance of this class and every one of its instances is associated with an instance of `BaseMetricDefinition`.

In addition, several associations are defined. They relate any type of CIM managed resource (defined as subclasses of `ManagedElement`), metric definitions, and metric values:

- `MetricDefForME` defines which `BaseMetricDefinition` objects are available for a given resource (i.e., `ManagedElement`).
- `MetricInstance` allows the traversal from a `BaseMetricDefinition` to one or more `BaseMetricValue` objects that may exist.
- `MetricForME` allows finding all `BaseMetricValue` objects for a given `ManagedElement`.

In addition to these classes and associations, figure 4 also depicts a `MetricDefinition` class, which has a property *Validity*. This class, however, is not part of the BaseMetrics model, but belongs to the Unit of Work model, which had been specified before work on the BaseMetrics model had started. One of the properties of this class was *Validity*, which indicates when a metric can be considered valid: Some metrics are valid only at the beginning of a unit of work (e.g., bytes to transfer), while the unit of work is running (e.g., percent complete), or when the unit of work is finished (e.g., pages printed). However, this information is only relevant in the context of units of work (such as batch jobs, user-initiated interactive operations, transactions executed under the control of a TP Monitor) and not needed by a general purpose metric definition. In addition, general purpose metrics are not associated to a unit of work, but to a `ManagedElement`. In CIM, the way to remove attributes from existing classes without breaking existing implementations and compatibility of the model is to create a new superclass. Thus, `BaseMetricDefinition` is defined as the new base class of the Metrics model; it contains all the previously defined properties, without the *Validity* property.

A. Details of the `BaseMetricDefinition` *class*

The purpose of `BaseMetricDefinition` is to provide a mechanism for introducing a new metric definition at runtime. An instance of `BaseMetricsDefinition` defines a single metric with the following properties:

Id: an identifying property (a key) that has no semantics. The usage scenario is that a CIM client application may ask for all `BaseMetricDefinitions` associated to a given resource.

Name: a descriptive name of the metric (e.g., ”Request-Rate”).

DataType: a standard CIM data type like ”string”, ”uint32”, ”real64”.

Calculable: an enumeration that defines the characteristics of calculations that may be performed on this metric. The possible values are: 1) ”Non-calculable”: arithmetic makes no sense, 2) ”Non-summable”: it makes no sense to sum this value over many instances of `BaseMetricValue`, 3) ”Summable”: It is reasonable to sum this value over many instances, such as the number of errors.

Units: identifies the specific units of a value, like Bytes or Packets.

BreakdownDimensions: an array of strings that defines the breakdown dimensions for this metric definition. See section II-C for more details on *BreakdownDimensions*.

B. Details of the `BaseMetricValue` *class*

Once the meta-data for a metric has been defined by means of an instance of `BaseMetricDefinition`, the values for a metric are gathered in instances of the `BaseMetricValue` class. One value is stored in each instance. The properties in this class are:

InstanceId: an identifying property for the object that has no semantics.

MetricDefinitionId: the reference to the `BaseMetricDefinition` instance for a `BaseMetricValue` instance.

MeasuredElementName: a descriptive name for the managed element being measured. This property is required if no association to a `ManagedElement` is defined, but may be used in other cases to provide supplemental information. It allows metrics to exist independently of a `ManagedElement`. In addition, keeping the name of the measured element as a property may be more efficient than traversing the `MetricInstance` associations if a very large amount of `BaseMetricValue`s exist for a given `BaseMetricDefinition`.

TimeStamp: the time when the value of a metric instance is computed or retrieved from the instrumentation. For a given `BaseMetricValue` instance, its *TimeStamp* changes whenever a new measurement snapshot is taken if the property *Volatile* (see below) is true. A management application may establish a time series of metric data by retrieving the instances of `BaseMetricValue` and sorting them according to their *TimeStamp*.

Duration: the time duration over which this metric value is valid. This property should not exist for time stamps that apply only to a point in time but should be defined for values that are considered valid for a certain time period (e.g., sampling).

MetricValue: the measured value itself, stored as a string. The value can be converted into a numeric CIM data type by looking up the *DataType* property of the associated class `BaseMetricDefinition`.

BreakdownDimension: if present, specifies one BreakdownDimension from the *BreakdownDimensions* array property, defined in the associated `BaseMetricDefinition` class. This is the dimension along which this set of metric values is broken down. See section II-C for more details.

BreakdownValue: the value of the *BreakdownDimension* property defined for this metric value instance.

Volatile: a boolean value indicating that the value for succeeding points in time may use the same object and just change its properties (such as the *MetricValue* or *TimeStamp*). If false, the existing objects remain unchanged and a new object is created for each new measurement. A more detailed discussion is provided in section II-B.

IV. Implementation Example: z/OS RMF

The z/OS *Resource Measurement Facility (RMF)*, is the strategic IBM tool for performance monitoring of the z/OS mainframe operating system. Having a CIM interface to RMF provides the following benefits:

- The RMF data store can be accessed by remote, platform-independent management applications. To support this, standardized metric definitions with common semantics need to be defined.
- For integration of systems management applications of various disciplines – like a performance monitor for DB2, a performance monitor for WebSphere, and a high-availability software package – having a common data model and sharing parts of the conceptual model enables a mixture of "best of breed" solutions, and it is easier for software vendors to enter the market as they do not need to implement a complete software framework anymore.

While the CIM model is independent of any implementation architecture, the DMTF interoperability protocols implement operations to manipulate CIM objects. A CIM Client sends operation requests to a CIM Server, which processes the operations. In typical implementations, the CIM Server is split into two major components: the *CIM Object Manager (CIMOM)* and one or more *CIM Providers*. The CIMOM is responsible for protocol and operation processing, security, and respositories; the CIM Provider is the glue code between the resource instrumentation and its CIM object representation.

RMF already has a persistent data store for performance monitoring values that offers powerful searching capabilities and which is highly optimized for performance and minimal memory footprint. CIM compliant access to this performance data can be implemented by encapsulating the proprietary access methods for the existing z/OS RMF performance repository with CIM providers. Note that this does not create a new CIM based performance data repository, but uses the existing one. Whenever a CIM client application asks for performance data, the RMF Metrics provider would forward this request to the existing z/OS RMF performance infrastructure and return the result.

As z/OS RMF supports breakdown dimensions to query the repository by job, workload or service class, the CIM BaseMetrics model maps directly to corresponding RMF metrics.

A. RMF Architecture with CIM Provider

IBM zSeries mainframe computers are typically used in a cluster environment using Parallel Sysplex[2] technology. On every z/OS operating system image, one RMF data gatherer is running. The data gatherer gets data periodically and writes data samples once per data collection cycle (default: 100 seconds). The performance data samples are stored for some period of time in a VSAM hierarchical database, which is an integral part of the z/OS operating system. As the performance of this data store is critical to the overall system and accordingly optimized, there is no motivation to replace it with a CIM Object Manager. By using the CIM BaseMetrics model, one is able to apply an object-oriented model while keeping a hierarchical data store underneath. Figure 5 depicts this layered

[2]A Sysplex is a cluster of mainframes, tightly coupled using shared storage in the Coupling Facilities.

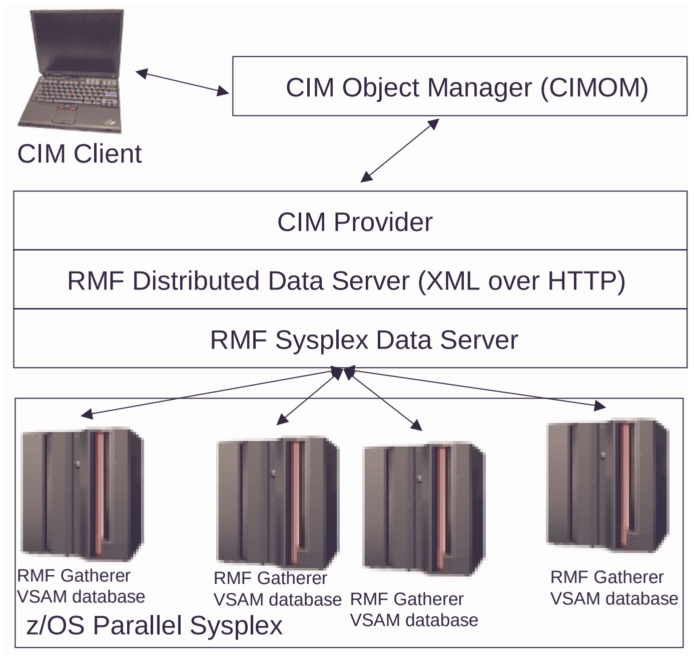

CIM Client
CIM Object Manager (CIMOM)
CIM Provider
RMF Distributed Data Server (XML over HTTP)
RMF Sysplex Data Server
RMF Gatherer VSAM database
RMF Gatherer VSAM database
RMF Gatherer VSAM database
RMF Gatherer VSAM database
z/OS Parallel Sysplex

Fig. 5. Providing a CIM interface to the z/OS RMF data store

architecture. Using the RMF Sysplex Data Server, every RMF instance in the Sysplex has access to all performance data sysplex-wide. On top of the Sysplex Data Server is the Distributed Data Server, which maps the RMF data into XML over HTTP format. This interface could be exploited by a CIM Metrics Provider on top of z/OS RMF.

B. Important CIM Resource Classes in the RMF Context

For the implementation of a RMF CIM Metrics provider, the most important resource classes are `ComputerSystem` as a representation of the logical partition in which the operating system is running, `OperatingSystem`, `Process`, `UnixProcess`, `NetworkPort` and `FileSystem`.

The following use cases illustrate the usage of the CIM BaseMetrics model in the z/OS RMF context:

a) Use Case 1: Get current free physical memory of z/OS Operating System: In this simple case, a management application has to enumerate `BaseMetricValue` instances that are associated to `OperatingSystem` for a given `BaseMetricDefinition` instance. Ideally, the key of the base metric definition instance can be used in the association call. If no specific time stamp is given, the RMF provider assumes that the most current metric value is to be returned.

b) Use Case 2: Get time series of free physical memory metrics for z/OS Operating System: This is very similar to use case 1; in addition, the application also specifies the timestamp it is looking for in the associator call.

c) Use Case 3: Get OS CPU consumption for a given service class: The extension needed in order to execute such a query is the specification of a breakdown dimension for a z/OS service class.

Dealing with opaque keys: The `BaseMetricValue` object instances are not actually persisted inside the CIM Object Manager, but created only when a management application asks for them. Therefore, a unique mapping between the *InstanceId* key property of the `BaseMetricValue` objects and the RMF identification of the metrics values is needed, in both directions. To solve this problem, the *InstanceId* key property of `BaseMetricValue` can be created by the CIM provider from the proprietary RMF query string. This allows the CIM provider to determine the RMF query string from the *InstanceId*.

V. Extending the Metrics Model for Metric Aggregation and Summarization

Figure 6 depicts the CIM Metric Aggregation and Summarization model, an extension to the CIM BaseMetrics Model, introduced in section III. It is currently being discussed in the DMTF Metric Extensions Working Group for inclusion in CIM version 2.9. Section V-A gives a brief overview of the main classes and associations; section V-B illustrates how the model works by means of a detailed example. Finally, we describe in section V-C our experiences with developing a CIM *Measurement Provider*, a special kind of CIM provider that implements a metric aggregation and summarization service, based on the model. The functionality of the provider is related to the *Summarization Function* [10] of the ISO–OSI Management Framework [11].

A. Aggregation and Summarization Model: Overview

The purpose of this model is to facilitate the computation of arbitrary *Composite Metrics*, which are aggregated from other metric types. These types are derived from the `BaseMetricDefinition` class as follows: We distinguish between resource metrics (e.g., counters, gauges) that are directly retrieved from a managed resource, composite metrics, and time series. Thus, the model comprises three classes `ResourceMetricDefinition`, `CompositeMetricDefinition` and `TimeSeriesDefinition` – depicted in the left part of Figure 6 – that hold the definitions for these metric types.

`ResourceMetricDefinition` contains, in addition to the properties inherited from `BaseMetricDefinition`, properties that allow the specification of how counters and gauges are retrieved from managed resources. Its

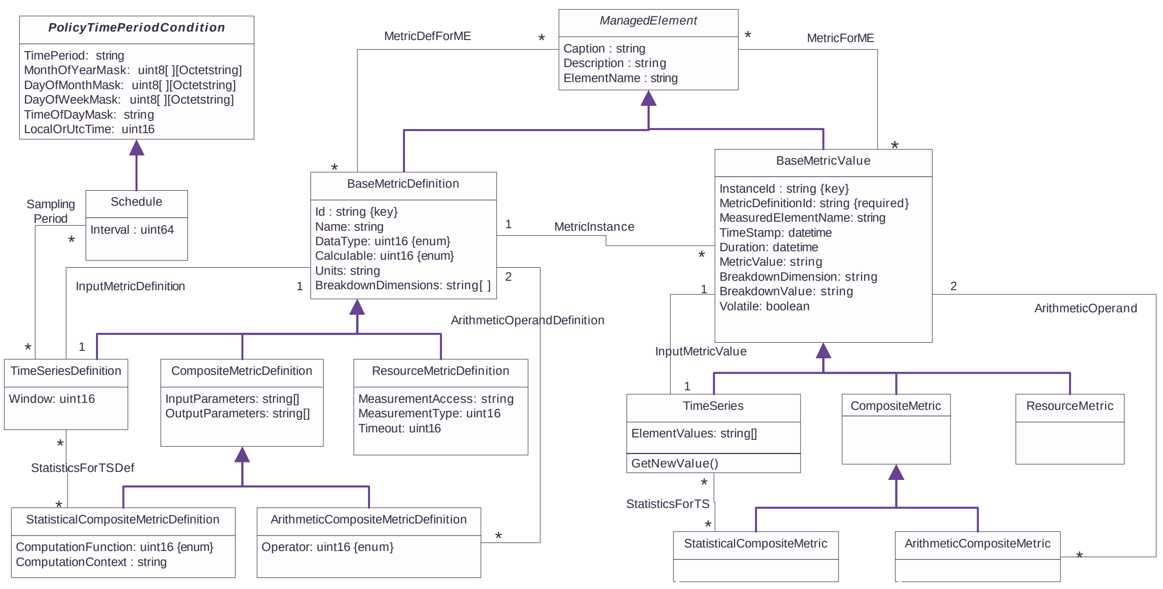


Fig. 6. Metric Aggregation and Summarization Model

additional properties *MeasurementAccess*, *Measurement-Type* and *Timeout* contain data needed for interfacing with local or remote managed resources that do not have a CIM interface, such as SNMP-managed devices (cf. the example in section V-B for more details).

The class `CompositeMetricDefinition` is further refined to address two types of complex metrics that need to be computed in a different way by the measurement provider. The `ArithmeticCompositeMetricDefinition` class represents an arithmetic operator (e.g., +,-,*,/), that aggregates two `MetricDefinitions` by following the association `ArithmeticOperandDefinition`. The `StatisticalCompositeMetricDefinition` class captures the definitions of statistical functions that apply to times series. Note that, in contrast to the OSI Metric Objects and Attributes [12], we do not define statistical functions as object classes, but as enumerated integer values of the property *Computation-Function*. As our model contains close to two dozen statistical function definitions, using a property for defining them prevents cluttering of the model with an excessive number of classes. Some statistical functions (e.g., minimum, maximum, mean, median etc.) can execute as-is on a given time series, while others (e.g., round, percentage greater/less than threshold etc.) require an administrator to provide additional context (e.g., the precision of the round function, the threshold against which values are to be compared) to be carried out. This context must be given by the administrator in the property *Computation-Context*.

A `TimeSeriesDefinition` comprises the number of metrics (*Window*) that will be stored for further computation. The intervals during which metrics are collected and placed into a time series (cf. the association `SamplingPeriod`) are represented by the class `Schedule`, which extends the class `PolicyTimePeriodCondition` of the CIM Policy Model [13] by a property 'Interval'.

The three metric types, along with the function definitions and schedules allow the definition of arbitrarily complex metrics, such as the average utilization of network interfaces or the maximum response time of a system within the last hour, sampled every five minutes. Note that all the classes discussed until now are used to represent the definitions — and not the actual measurement values. They are instantiated whenever a new measurement algorithm is deployed to the Measurement Provider. This can happen either manually by an administrator, or automatically, e.g., when a new SLA containing these definitions is deployed.

Once the Measurement Provider has access to the instances of the definition classes, it uses these definitions to retrieve and compute the actual values by instantiating the subclasses of `BaseMetricValue` – depicted in the right part of Figure 6 – and assigning values to their properties.

The computation of composite metric values requires the automatic retrieval of metric values by the Measurement Provider. During runtime, instances of `Schedule` are used to trigger the retrieval of current metric values and to perform metric computation and aggregation. The input metrics and the (intermediate or final) results are represented by the classes discussed below.

In regular time intervals, a `Schedule` instance initiates the collection of a new metric value for

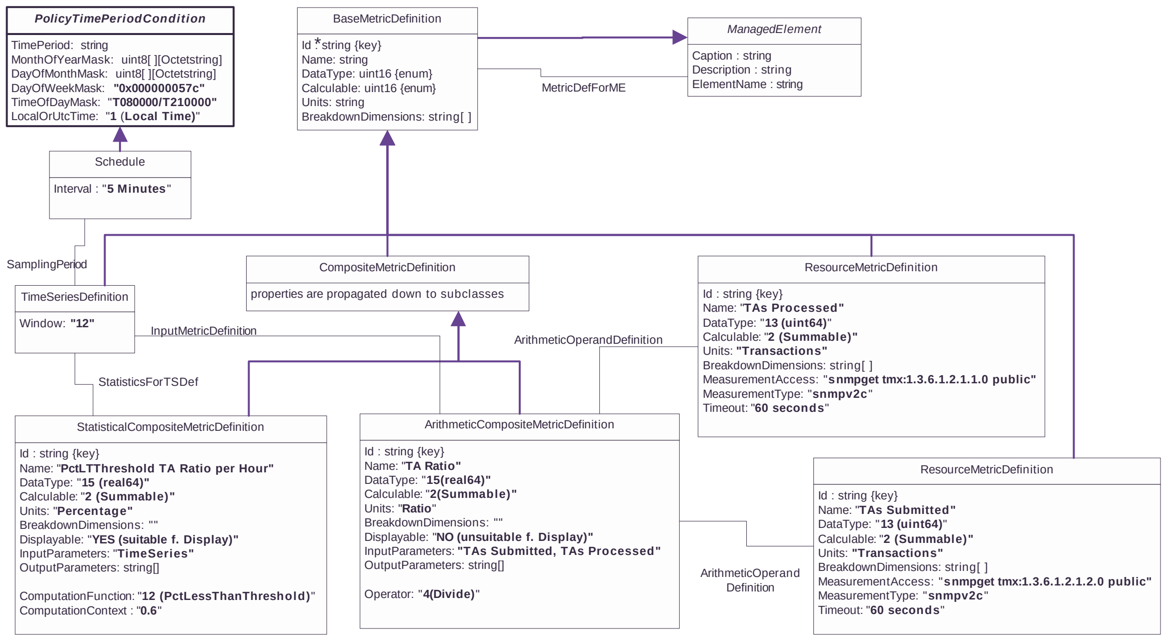

PolicyTimePeriodCondition
TimePeriod: string
MonthOfYearMask: uint8[ ][Octetstring]
DayOfMonthMask: uint8[ ][Octetstring]
DayOfWeekMask: "0x000000057c"
TimeOfDayMask: "T080000/T210000"
LocalOrUtcTime: "1 (Local Time)"
BaseMetricDefinition
Id :*string {key}
Name: string
DataType: uint16 {enum}
Calculable: uint16 {enum}
Units: string
BreakdownDimensions: string[ ]
ManagedElement
Caption : string
Description : string
ElementName : string
MetricDefForME
Schedule
Interval : "5 Minutes"
SamplingPeriod
TimeSeriesDefinition
Window: "12"
InputMetricDefinition
CompositeMetricDefinition
properties are propagated down to subclasses
ArithmeticOperandDefinition
StatisticsForTSDef
ResourceMetricDefinition
Id : string {key}
Name: "TAs Processed"
DataType: "13 (uint64)"
Calculable: "2 (Summable)"
Units: "Transactions"
BreakdownDimensions: string[ ]
MeasurementAccess: "snmpget tmx:1.3.6.1.2.1.1.0 public"
MeasurementType: "snmpv2c"
Timeout: "60 seconds"
StatisticalCompositeMetricDefinition
Id : string {key}
Name: "PctLTThreshold TA Ratio per Hour"
DataType: "15 (real64)"
Calculable: "2 (Summable)"
Units: "Percentage"
BreakdownDimensions: ""
Displayable: "YES (suitable f. Display)"
InputParameters: "TimeSeries"
OutputParameters: string[]
ComputationFunction: "12 (PctLessThanThreshold)"
ComputationContext : "0.6"
ArithmeticCompositeMetricDefinition
Id : string {key}
Name: "TA Ratio"
DataType: "15(real64)"
Calculable: "2(Summable)"
Units: "Ratio"
BreakdownDimensions: ""
Displayable: "NO (unsuitable f. Display)"
InputParameters: "TAs Submitted, TAs Processed"
OutputParameters: string[]
Operator: "4(Divide)"
ArithmeticOperand Definition
ResourceMetricDefinition
Id : string {key}
Name: "TAs Submitted"
DataType: "13 (uint64)"
Calculable: "2 (Summable)"
Units: "Transactions"
BreakdownDimensions: string[ ]
MeasurementAccess: "snmpget tmx:1.3.6.1.2.1.2.0 public"
MeasurementType: "snmpv2c"
Timeout: "60 seconds"

Fig. 7. Instance Diagram of the Metric Aggregation and Summarization **Definition** Classes. The definitions are created by an Administrator or taken from an SLA, and deployed to the Measurement Provider. The inheritance hierarchy is shown for illustrative purposes.

a `TimeSeries` object by invoking its `GetNewValue()` method. This causes the collection of the `ArithmeticCompositeMetric` associated with the `TimeSeries`, which is done by means of the CIM protocol operation `getInstance`, defined in [14]. Before the `ArithmeticCompositeMetric` instance can be calculated, its associated `ResourceMetrics` have to be retrieved. After the calculation is done, the result is given back and stored within the *ElementValues* property of a `TimeSeries` object, a string array whose size is defined in the property `TimeSeriesDefinition`.*Window*. `TimeSeries` instances may be used as input for any number of statistical composite metrics (cf. the association `StatisticsForTS`). This reduces redundancy and ensures the integrity of measurement data by providing a shared basis for statistical calculations.

The second possible activation mechanism is a CIM request, issued by a CIM client to a CIMOM, and dispatched to the Measurement Provider. If a request for a `StatisticalCompositeMetric` is received, the associated `TimeSeries` instance has to be retrieved. Once this is done, the average value is calculated based on the values retrieved from the `TimeSeries` object.

B. Usage Example: Scheduled Measurements

We will now illustrate the usage of the model with a workload management scenario. In order to distribute workload, we are interested in finding out if a database server is overloaded by examining its transaction ratio (defined here, for the sake of simplicity, as the ratio of processed and submitted transactions). This is done on an hourly basis, with measurements taken every 5 minutes, during business days from 8am to 9pm. We consider a database server overloaded whenever its transaction ratio is less than 0.6. If, over the course of an hour (12 measurements), a server is overloaded for 30% of the time, the workload management system needs to direct new incoming load to a different server. We further assume that the database management system exposes its performance data through an SNMPv2c (Community based SNMP version 2) interface, out of which we are interested in the number of submitted (*TAsSubmitted*) and processed transactions (*TAsProcessed*).

Figure 7 depicts how this measurement request is expressed in the Metric Aggregation and Summarization Model. The information can be either provided by an administrator, or generated from a Service Level Agreement. We begin with the two `ResourceMetricDefinition` objects, depicted on the right side of figure 7 and gradually proceed to the left. Both objects, named ”TAs Submitted” and ”TAs Processed”, are accessed through SNMPv2c with the command specified in the property *MeasurementAccess*. A timeout value of 60 seconds is specified, as the counters are retrieved from a remote system called ”tmx”.

The `ArithmeticCompositeMetricDefinition` object specifies that the transaction ratio (”TA Ratio”) is computed by applying the *Operator* ”Divide” to

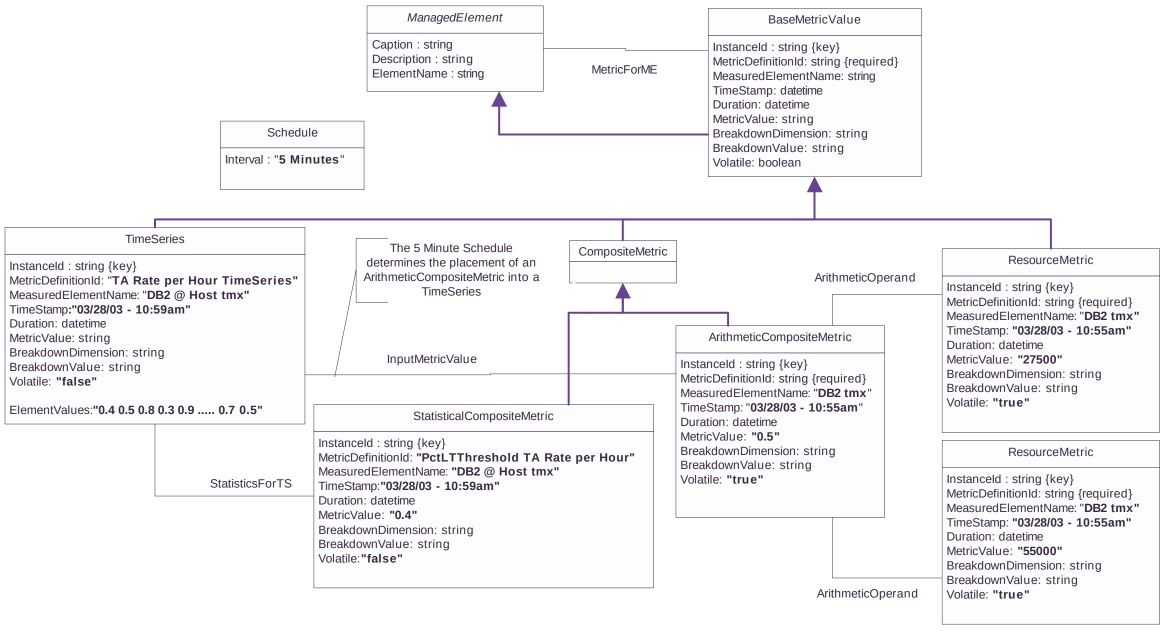

ManagedElement
Caption : string
Description : string
ElementName : string
MetricForME
BaseMetricValue
InstanceId : string {key}
MetricDefinitionId: string {required}
MeasuredElementName: string
TimeStamp: datetime
Duration: datetime
MetricValue: string
BreakdownDimension: string
BreakdownValue: string
Volatile: boolean
Schedule
Interval : "5 Minutes"
TimeSeries
InstanceId : string {key}
MetricDefinitionId: "TA Rate per Hour TimeSeries"
MeasuredElementName: "DB2 @ Host tmx"
TimeStamp:"03/28/03 - 10:59am"
Duration: datetime
MetricValue: string
BreakdownDimension: string
BreakdownValue: string
Volatile: "false"
ElementValues:"0.4 0.5 0.8 0.3 0.9 ..... 0.7 0.5"
The 5 Minute Schedule determines the placement of an ArithmeticCompositeMetric into a TimeSeries
CompositeMetric
ArithmeticOperand
ResourceMetric
InstanceId : string {key}
MetricDefinitionId: string {required}
MeasuredElementName: "DB2 tmx"
TimeStamp: "03/28/03 - 10:55am"
Duration: datetime
MetricValue: "27500"
BreakdownDimension: string
BreakdownValue: string
Volatile: "true"
InputMetricValue
ArithmeticCompositeMetric
InstanceId : string {key}
MetricDefinitionId: string {required}
MeasuredElementName: "DB2 tmx"
TimeStamp: "03/28/03 - 10:55am"
Duration: datetime
MetricValue: "0.5"
BreakdownDimension: string
BreakdownValue: string
Volatile: "true"
StatisticalCompositeMetric
InstanceId : string {key}
MetricDefinitionId: "PctLTThreshold TA Rate per Hour"
MeasuredElementName: "DB2 @ Host tmx"
TimeStamp:"03/28/03 - 10:59am"
Duration: datetime
MetricValue: "0.4"
BreakdownDimension: string
BreakdownValue: string
Volatile:"false"
StatisticsForTS
ResourceMetric
InstanceId : string {key}
MetricDefinitionId: string {required}
MeasuredElementName: "DB2 tmx"
TimeStamp: "03/28/03 - 10:55am"
Duration: datetime
MetricValue: "55000"
BreakdownDimension: string
BreakdownValue: string
Volatile: "true"
ArithmeticOperand

Fig. 8. Instance Diagram of the Metric Aggregation and Summarization **Value** Classes. They are instantiated by the Measurement Provider.

the objects associated by means of an `ArithmeticOperandDefinition` association. As the order of the parameters is important for the divide operation, the *InputParameters* array lists the operands in the proper order.

Once the transaction ratio is specified, one needs to define the validity period and the sampling interval, as well as the number of values that need to be kept for further processing. `PolicyTimePeriodCondition`.*DayOfWeekMask* is set to ”Monday through Friday” and `PolicyTimePeriodCondition`.*TimeOfDayMask* is set to ”from 8am to 9pm”. These values are encoded according to the mapping specified in the Policy Core Information Model [15]. The *Interval* property of `Schedule` explicitly states that measurements of ”TA Ratio” need to be taken every 5 minutes (cf. the association `InputMetricDefinition`) and stored in a `TimeSeries` object whose *Window* size is 12. By implementing the class `TimeSeries` as a ring buffer, an hour-long set of measurements, taken every 5 minutes, is available for further processing.

Finally, one needs to determine the percentile of how long a database server was subject to an overload condition by comparing the values stored in the `TimeSeries` object against a threshold (0.6 in our case). This is expressed in the class `StatisticalCompositeMetricDefinition` as follows: The *ComputationFunction* property is set to ”PercentageLessThanThreshold”, which determines how many elements of a time series are below a given threshold, expressed as a percentage; the threshold itself is expressed by setting *ComputationContext* to ”0.6”. Note that all the definition objects described above remain constant for every measurement type and are defined once.

Figure 8 depicts the instances of the value objects, whose object classes are derived from the class `BaseMetricValue`. The purpose of the `Schedule` class is to trigger the insertion of a new measurement into a `TimeSeries` object every 5 minutes. This leads to the traversal of the `InputMetricValue` association to obtain the `ArithmeticCompositeMetric` value that needs to be inserted, which, in turn, triggers the retrieval of the `ResourceMetric` objects that are associated with `ArithmeticCompositeMetric`. The two `ResourceMetric` objects hold the values of ”TAs Processed” (27500) and ”TAs Submitted” (55000) at the time the measurement is taken (recorded in the *TimeStamp* property). According to the specification in `ArithmeticCompositeMetricDefinition`, they need to be divided, which yields an `ArithmeticCompositeMetric`.*MetricValue* of ”0.5”. This value is inserted in the `TimeSeries`.*ElementValues* array property for further processing. In our case, the PercentageLessThanThreshold function is used to determine how many elements of the `TimeSeries`.*ElementValues* are below the threshold of 0.6, and express this value as a percentage in the property *MetricValue* of the class `StatisticalCompositeMetric`. In our case, the result is ”0.4”, which means that a database server has been overloaded 40% of the time. Based on the locally stored threshold of 30%, a workload manager would have to issue corrective actions, e.g., dispatching incoming requests to other servers.

The example also shows how a time series can be used to decouple the computation of a statistical composite metric from the process of aggregating a potentially large amount of arithmetic composite or resource metrics. The advantage of using a time series as a data store is that one does not need to keep all its associated metric objects as separate instances, but can reuse object instances by overwriting their values periodically. This is done by setting the *Volatile* property in these classes to ”true” (cf. the discussion in section II-B). In our example, having a time series with a window size of 12 eliminates the need of keeping a total of 36 resource and arithmetic composite metric objects in memory, because the values of the 3 objects get overwritten whenever a new measurement is taken.

C. *Implementation of the Measurement Provider*

The Measurement Provider was successfully implemented with an open-source CIMOM implementation, the SNIA (Storage Networking Industry Association) CIMOMversion 0.7 [16] and the Java Development Kit 1.3.1. One of the major novelties of our approach is the use of active CIM providers. While active management agents have been known in the network management community for some time, CIM providers are, until now, stateless resource providers. They are passive and surface information from managed resources without providing sophisticated processing capabilities. Usually, the retrieval of information is initiated by the CIM client, a management application. Stateless providers may cause a considerable overhead, e.g., by requiring polling of new values from management applications.

As described in section V-B, the CIM providers we implemented (described in detail in [17]) actively compute the values of high-level metrics by autonomously retrieving low-level resource metric values from managed resources and aggregating them, without being triggered by a management application. The retrieval of new metrics is automatically requested by the provider implementing the `Schedule` class of the model. Making use of this delegated management functionality reduces the overhead introduced by polling significantly.

To minimize the implementation costs, instances of simple classes like `BaseMetricDefintion` are created and statically stored in the CIM Repository. The repository is an internal database of a CIMOM implementation that keeps class definitions and static instances of a specified set of these classes. For more complex classes, such as `Schedule` and the subclasses of `BaseMetricValue`, static creation and storage is not sufficient. For example, metric value classes have to perform calculations every time they are retrieved by a CIM client (aka the management application). In addition, the `Schedule` class has to independently trigger the calculation of the metrics. The following classes are implemented by individual providers: `Schedule`, `ResourceMetric`, `TimeSeries`, `ArithmeticCompositeMetric`, and `StatisticalCompositeMetric`. These providers not only deliver the data that instantiates the aforementioned classes, but also implement the logic needed for processing activities like scheduling and metric aggregation.

VI. SLAs and the CIM Metrics Model

Along with organizational data describing the involved parties, a Service Level Agreement (SLA) [18] comprises the description of a service, the definitions of the metrics used to measure the quality of service (SLA parameters), and the service levels that define the allowable value ranges for each service parameter along with (monetary) penalties for violating service levels. Today, a complete SLA is unlikely to be represented in CIM, because this would imply that both service provider and service customer use CIM compliant management systems. However, at some point in the SLA monitoring process, SLA parameters need to be mapped onto the performance data that (CIM based) managed resources expose so that the managed resources can interpret and consume the SLA document that is sent to them. What is needed is an automated provisioning of the SLA monitoring environment, according to the service parameter defintions in an SLA.

This is where the CIM BaseMetrics model (cf. section III) and the Metric Aggregation and Summarization Model (described in section V) can be applied: The SLA parameter definitions and the way how they are aggregated from resource metrics are often made explicit in today’s SLAs and follow exactly the same pattern as the workload management scenario in section V-B: First, in the SLA deployment phase, SLA parameter definitions are mapped onto `BaseMetricDefinition` and its subclasses. In a second step, the measurements are carried out, whose objects are represented as subclasses of `BaseMetricValue`. This approach, along with a prototype implementation, is described in detail in [17]. Similar work [19] has been carried out for mySAP CRM 3.0, SAP’s customer relationship management software.

It should be noted that recent work in the Policy Working Group aims at extending CIM to express SLAs. Once an `SLA` class is defined in the CIM Policy Model, the aggregation and summarization model can be fully reused by introducing just two new associations `SLAParameterDefinition` and `SLAParameter` that relate the `SLA` class to `BaseMetricDefinition` and `BaseMetricValue`, respectively.

VII. Conclusions and Outlook

We have described the CIM BaseMetrics Model, which has recently been released as part of CIM version 2.7. To illustrate the applicability of the model to real-life environments, we have described two typical usage scenarios: the first scenario deals with instrumenting an existing performance management data store of a mainframe cluster, while the second, more complex scenario describes a model for the aggregation and summarization of metrics, based on workload management policies or SLAs.

During this work, it was recognized that the advantages of the late binding concept, which the Metrics Model is based upon, apply to a wide range of managed elements (services, devices, systems, networks, SLAs, etc.). In addition, our work has shown that fairly complex operations, such as breaking down performance data according to user-defined criteria and the creation of time series can be carried out with a small set of information, defined as properties in the two classes of the BaseMetrics Model. Finally, we have demonstrated that it is possible to implement reusable management services in CIM by specifying their data model in the same way as one would do when a resource needs to be instrumented. The architecture of a general-purpose CIM Object Manager, along with the CIM Provider concept, is capable of supporting the needs of autonomous management services without further modification. We consider the results of the work described in this paper as a first step towards the goal of extending CIM from a resource instrumentation framework to a management architecture for truly distributed management.

Based on the experiences gained during this work, the CIM Metric Extensions Working Group is currently extending the model to facilitate the integration of performance metrics that have been defined in different management environments, such as SNMP-based management, or proprietary data stores. In addition, a hysteresis model and a subscription-based notification mechanism based on the CIM Event Model are currently being developed, so that management applications get notified whenever a predefined threshold is crossed. The emerging Service Level Agreement model, which is currently being developed by the CIM Policy Working Group, is another area where the concepts underlying the Metrics Model could be applied.

References

[1] "Common Information Model (CIM) Version 2.2," Specification, Distributed Management Task Force, June 1999, http://www.dmtf.org/standards/cim_spec_v22/.

[2] W. Bumpus, J.W. Sweitzer, P. Thompson, A.R. Westerinen, and R.C. Williams, *Common Information Model: Implementing the Object Model for Enterprise Management*, Wiley, 2000.

[3] "Information Technology – Open Systems Interconnection – Systems Management – Management Functions," IS 10164-x, ISO/IEC, 1991-94.

[4] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke, "The Physiology of the Grid: An Open Grid Service Architecture for Distributed Systems Integration," Draft, Globus Project, July 2002.

[5] M. Rose and K. McCloghrie, "Concise MIB Definitions," RFC 1212, IETF, Mar. 1991.

[6] R. Berry and J. Hellerstein, "A Flexible and Scalable Approach to Navigating Measurement Data in Performance Management Applications," in *Proceedings of the Second International Conference on Systems Management*, June 1996.

[7] "z/OS Resource Measurement Facility: Report Analysis," Manual, IBM Corporation, July 2003.

[8] "z/OS Resource Measurement Facility: Performance Management Guide," Manual, IBM Corporation, Oct. 2001.

[9] "Application Response Measurement, Issue 3.0 - Java Binding," Open Group Technical Standard, Document Number: C014, The Open Group, Oct. 2001.

[10] "Information Technology – Open Systems Interconnection – Systems Management – Part 13: Summarization Function," IS 10164-13, International Organization for Standardization and International Electrotechnical Committee, 1995.

[11] "Information Processing Systems – Open Systems Interconnection – Basic Reference Model – Part 4: Management Framework," IS 7498-4, ISO/IEC, Nov. 1989.

[12] "Information Technology – Open Systems Interconnection – Systems Management – Part 11: Metric Objects and Attributes," IS 10164-11, International Organization for Standardization and International Electrotechnical Committee, 1994.

[13] "CIM Policy White Paper," Version 2.7, Distributed Management Task Force, June 2003, http://www.dmtf.org/standards/documents/CIM/DSP0108.pdf.

[14] "Specification for CIM Operations over HTTP, Version 1.1," Specification, Distributed Management Task Force, May 2002, http://www.dmtf.org/standards/documents/WBEM/DSP200.html.

[15] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen, "Policy Core Information Model - Version 1 Specification," RFC 3060, IETF, Feb. 2001.

[16] "SNIA CIM Object Manager Version 0.7," Open Source Java CIMOM, The Open Group, June 2002, http://www.opengroup.org/snia-cimom/.

[17] M. Debusmann and A. Keller, "SLA-driven Management of Distributed Systems using the Common Information Model," in *Proceedings of the 8th IFIP/IEEE International Symposium on Integrated Network Management*, G.S. Goldszmidt and J. Schönwälder, Eds. Mar. 2003, Kluwer Academic Publishers.

[18] L. Lewis and P. Ray, "Service Level Management: Definition, Architecture, and Research Challenges," in *Proceedings of the IEEE Global Communications Conference (GlobeCom)*, 1999.

[19] B. Pauze, "Service Level Management for mySAP CRM 3.0," M.S. thesis, Universität Karlsruhe, Aug. 2002.