DEVELOPER MANUAL

G-PM

WP2.4

Document Filename:	**CG2.4.1-v1.2-CYF-G-PMDeveloperManual.doc**
Work package:	**WP2.4**
Partner(s):	**CYFRONET, TUM**
Lead Partner:	**CYFRONET**
Config ID:	**CG2.4.1-v1.2-CYF-G-PMDeveloperManual**
Document classification:	**PUBLIC**

Document Log

Version	Date	Summary of changes	Author
0.1	22/08/2004	Adaptation from earlier prototype templates	Piotr Nowakowski
0.3	29/08/2004	Inclusion of TB comments, formatting	Ariel Garcia, Piotr Nowakowski
1.0	20/12/2004	First draft of G-PM docs	Wlodzimierz Funika, Tomasz Arodz, Marcin Kurdziel
1.1	22/12/2004	Inclusion of HLAC docs	Hamza Mehammed
1.2	05/01/2005	Updates and corrections	Roland Wismueller
	26/01/2005	Verified by the QE	Robert Pajak

CONTENTS

COPYRIGHT NOTICE 4

1. INTRODUCTION..... 5

1.1. ABBREVIATIONS AND ACRONYMS 5
1.2. REFERENCES AND SOURCE CODE 6

2. IMPLEMENTATION STRUCTURE 7

2.1. PRODUCT COMPONENT MODEL 7
2.2. DETAILED IMPLEMENTATION MODEL..... 8
2.2.1. The measurement interface..... 8
2.2.2. Performance Measurement component (PMC) class diagrams 20
2.2.3. The HLAC component 24
2.2.4. The class diagram of the UIVC component..... 47

3. PRODUCT TESTING 49

3.1. PMC..... 49
3.2. HLAC 49

4. CONTACT INFORMATION AND CREDITS..... 50

5. GNU GENERAL PUBLIC LICENSE, VERSION 2, JUNE 1991 51

COPYRIGHT NOTICE

Copyright (c) 2005 by ACC CYFRONET AGH, Krakow, Poland; AGH University of Science and Technology, Krakow, Poland; Technische Universität München, Germany; Universität Siegen, Germany. All rights reserved.

Use of this product is subject to the terms and licenses stated in the GPL license agreement. Please refer to Section 5 for details.

This research is partly funded by the European Commission IST-2001-32243 Project "CrossGrid".

1. INTRODUCTION

The G-PM is the CrossGrid performance monitoring tool for parallel grid applications. Its purpose is to monitor run-time behaviour of applications and detect possible performance bottleneck. The tool was designed to provide the user with performance data in an on-line fashion. Consequently it is not necessary to wait for the end of application's execution in order to analyse its performance. Rather, the analysis is being performed constantly and the user can react to improper application behaviour as soon as it occurs. This is particularly important in case of applications that have long execution times, which are common in the grid environment. The tool works in X-Windows environment and provides convenient graphical interface for defining measurement and visualizing of performance data.

One of the most important features of the G-PM is its flexibility. The tool can be customized to support a huge range of monitoring scenarios. First, the G-PM provides a number of predefined performance metrics for MPI applications. New metrics can be added by the user with an aid of the dedicated Performance Measurement Specification Language (PMSL). The PMSL also provides a probe mechanism that can be used to monitor control flow during application execution and to retrieve content of internal application variables. These features are especially useful for the application developer. Using them the developer can design monitoring scenarios usefully in detection of origins of possible performance bottlenecks. Finally, the measurement of both predefined and PMSL-based metrics can be narrowed to any subset of computing sites, nodes, processes or application functions. The measurement of the PMSL metrics can be further narrowed to intervals in application execution that are specified by enclosing probes.

The G-PM monitors the application by appropriate programming of another CrossGrid service, i.e. OCM-G. The main task of OCM-G is to perform low-level monitoring of application in a distributed way. Also, the initial processing of performance measurement results (e.g. partial aggregation of measured values) is preformed by OCM-G. This approach significantly reduces the computational overhead associated with the monitoring, thus minimizing the influence of the G-PM on the grid application behaviour. Furthermore, it reduces the overhead on the user workstation, which is important in case of monitoring large number of processes.

The main novelty of our approach is the combination of on-line monitoring with the support for user-defined, application specific performance metrics. Other on-line monitoring tools either focus on hardware infrastructure (e.g. Network Weather Service) or does not provide user-defined metrics whose functionality is comparable to the one of G-PM (e.g. GRM, Paradyn, Autopilot, TAU).

1.1. ABBREVIATIONS AND ACRONYMS

G-PM	The CrossGrid performance analysis tool
HLAC	High Level Analysis Component
OCM-G	OMIS Compliant Monitoring system for the Grid
OMIS	On-line Monitoring Interface Specification
PMC	Performance Measurement Component
UIVC	User Interface and Visualization Component
UML	Unified Modelling Language

1.2. REFERENCES AND SOURCE CODE

The source code of the tool is available at the CrossGrid CVS at:

https://savannah.fzk.de/cgi-bin/viewcvs.cgi/crossgrid/crossgrid/wp2/wp2_4-perf/wp2_4_1-perfmon/

The G-PM classes are located in the *src* subdirectory within the bundle and is arranged in the directory tree structure as follows:

`src` – source code and header files

- `uivc` – source and header files for the UIVC classes
- `pmc` – source and header files for the PMC classes
- `hlac` – source and header files for the HLAC classes
  - `dag` – source and header files for the intermediate representation of HLAC metrics
  - `dfn` – source and header files for the data flow graphs used to evaluate HLAC metrics
  - `ocm` – source and header files for an OCM-G extension supporting distributed evaluation of HLAC metrics
- `support` – source and header files for the classes that are shared among the above modules:
  - `include` – header files for the `support` classes
    - `measurement` – header files for the measurement interface classes
    - `auxilary` – header files for the auxiliary classes
  - `measurement` – source code files for the measurement interface classes
  - `auxilary` – source code for the auxiliary classes
- `ocmg-dummy` – source code and header files for a simulation of OCM-G used for testing

`doc` – user's manual and installation manual

`man` – manual pages

`test` – input files and scripts used for testing

`apidoc` – automatically generated documentation

`gtk+extra-0.99.17` – a copy of the GTK+extra library source code

2. IMPLEMENTATION STRUCTURE

2.1. PRODUCT COMPONENT MODEL

The module decomposition of G-PM is shown in Figure 2-1:

- UIVC implements the user interface and the visualizers for performance data,
- HLAC implements configurable (user-definable) performance metrics,
- PCM implements standard performance metrics,
- OCM-G provides application related monitoring services and also interfaces to CrossGrid components that provide infrastructure related performance data as well as benchmark data.

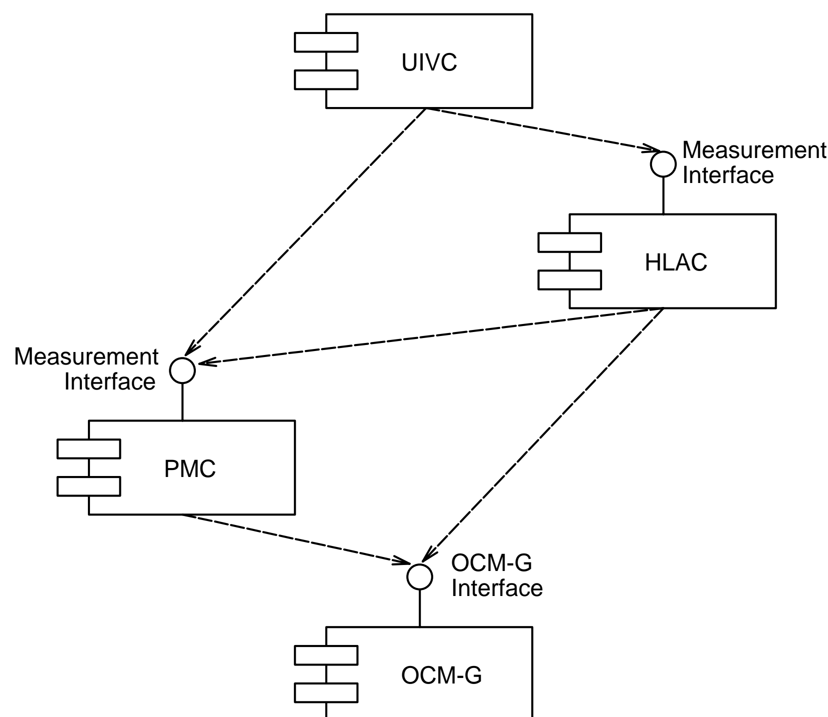


Figure 2-1 Module decomposition of the G-PM tool

There are two major interfaces between these modules:

- *Measurement Interface*:

  This interface allows to define performance measurements and to read their results. As can be seen from Figure 2-1, both the HLAC and the PMC provide the same interface.
- *OCM-G Interface*:

  The interface to the monitoring system is an external interface to Task 3.3. It is based on OMIS.

In the following section, we provide an overview on the different classes used in the Measurement Interface and their interrelations.

2.2. DETAILED IMPLEMENTATION MODEL

The components and interfaces of the G-PM, as specified above, are discussed in this section.

2.2.1. The measurement interface

2.2.1.1. Main classes for defining measurements

There are three main classes related to a performance measurement: *MeasurementSpec*, *ActiveMeasurement* and *Metrics*. These classes along with their relationships with other G-PM classes are depicted in **Figure 2.**2. These classes are described in more details below. The description focuses on the external interface which can be used to define and read measurements. It does not include internal methods which are only used for implementing the measurements.

The *MeasurementSpec* class is a definition of a single measurement. It includes:

1. Metrics definition represented by an instance of the *Metrics* class.
2. List of measured (involved) objects along with an optional list of partner objects. Both are represented by an instance of *AppObjectList* class.
3. List of measured code regions represented by an instance of *RegionList* class.

The *MeasurementSpec* class is used, with help of the *Metrics* class, to create instances of the *ActiveMeasurement* class, which represent an ongoing measurement process.

Instances of the *ActiveMeasurement* class represent actual measurements that are performed by the monitoring system. The class provides a functionality to create, start, stop and destroy a measurement, in general with help of a corresponding *Metrics* class. Each instance of the *ActiveMeasurement* class contains a *MeasurementSpec* object that defines the measurement, and via this class also a pointer to a *Metrics* object defining the data to be measured. Measurements performed by *ActiveMeasurement* instances are mutually independent. In particular this applies to the instances that refer to identical *MeasurementSpec* objects.

The third of the above-mentioned classes, the *Metrics* class describes the properties to be measured. It does not define the involved objects or measured code regions, but is responsible for defining the way in which the data should be measured. The definition of metrics can be hierarchical, that is a single metrics can be defined as a set of other submetrics. In addition the *Metrics* class itself contains a static class variable that represents a list of all top-level metrics that are available in the G-PM. A simple metrics is for example: communication volume, computation time etc. In general, since a *Metrics*-derived class knows how to measure and an *ActiveMeasurement*-derived class contains the monitoring service's object identifiers associated with that specific measurement, for each class inherited from *Metrics,* there should exist a corresponding class inherited from *ActiveMeasurement*, i.e. these classes always come in pairs.

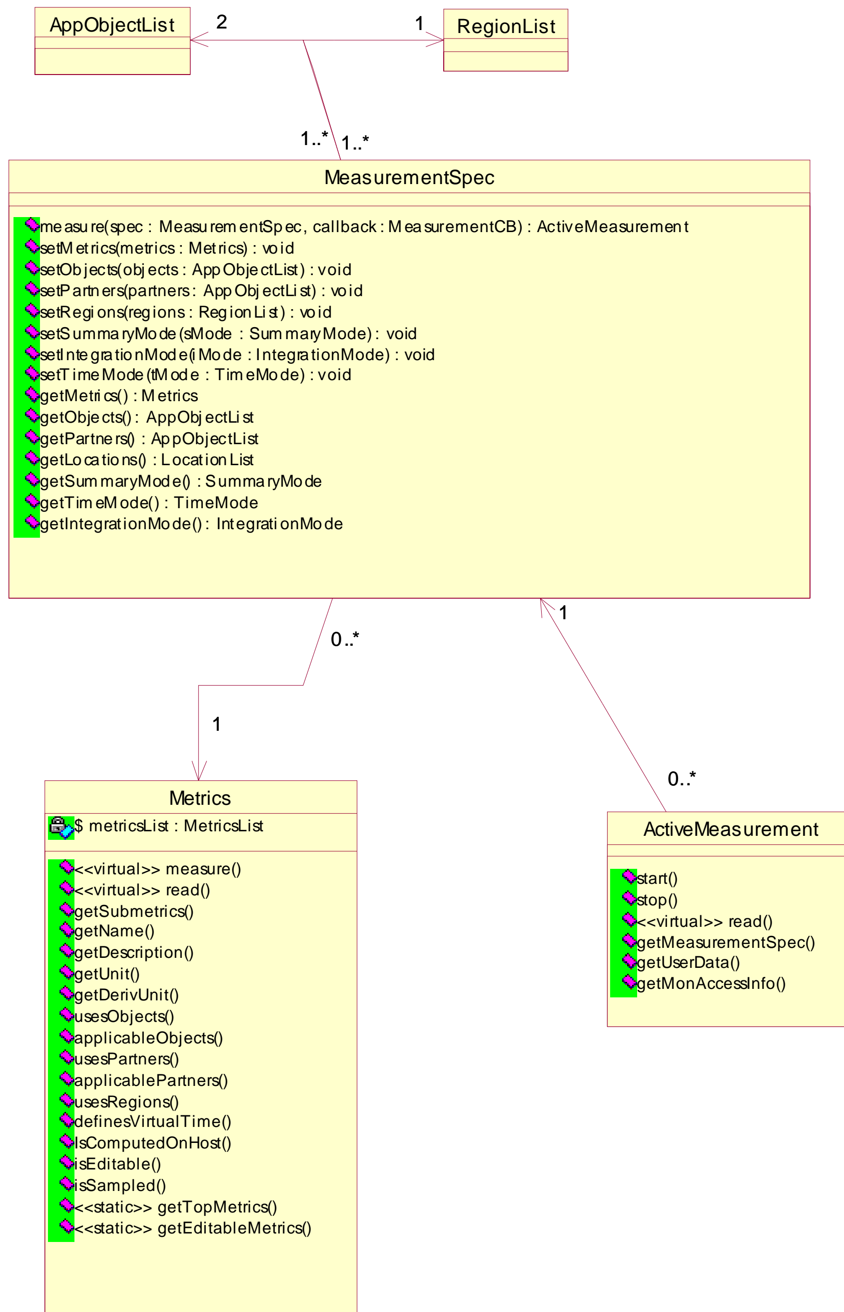


Figure 2.2 Class diagrams of performance measurement-related classes *MeasurementSpec*, *ActiveMeasurement* and *Metrics*

2.2.1.1.1. Metrics

An object of (an implementation class derived from) this abstract base class represents a metrics, i.e. a property that can be measured. The metrics just defines what is to be measured, it does not define

where or when to measure. The latter is controlled by additional parameters which must be supplied when an actual measurement is defined.

There are two very different ways to measure a performance value:

a. The value may be sampled from some source whenever the measurement is read. This situation is typical for infrastructure related metrics. For example, consider the CPU load of a node, which may be sampled from the OS.
b. The value may be determined by monitoring events, which occur whenever the value changes. This is typical for application related metrics. E.g. the amount of communication can be determined by monitoring all "message send" events and summing up the lengths of the messages.

In case *b*. we have an exact integral and thus, we can exactly determine the value of the metrics. In case *a*. we only have a current value. We call these metrics *sampled metrics*.

In application-specific metrics, we also want to measure the value of a metrics for an interval of time bounded by the execution of special procedure calls, called *probes* (i.e. generating special events) in the application. Probes are usually used to constrain the beginning and end of a code region. A typical example of this scenario is measuring the amount of communication between a user's request and the application's reply.

Metrics provides the following methods:

1. `string getName() const`

   Returns the name of the metrics. The only purpose of this name is that the user of G-PM can identify the metrics.

2. `string getDesc() const`

   Returns a concise description of this metrics, i.e. its documentation. Like the name, this description is intended to be displayed to the user of G-PM.

3. `string getIntegralUnit() const`
4. `string getNormalUnit() const`

   These methods return the unit (dimension) of the result of a measurement using this metrics. There are two methods, since when a measurement is defined, a parameter allows to select either the integral value of the metrics or the 'normal' value (i.e. the time derivative of the integral value). E.g. for the "amount of data sent" metrics *getIntegralUnit()* might return the string "KByte", while *getNormalUnit()* might return "KByte/s".

   Rationale: This feature is needed to support the implementation of the user-defined metrics.

5. `UseInfo usesObjects() const`

   Indicates whether the *objects* parameter can / must be used when defining a measurement based on this metrics (c.f. Section 2.2.1.1.2). The result type of this method is an enumeration consisting of three values:

REQUIRED	The parameter must be specified.
POSSIBLE	The parameter can be specified
NOT_APPLICABLE	The parameter must not be specified

Rationale: Although the method defining a measurement will always receive the parameter (and possibly ignore it), this information is needed for user guidance in the user interface: If the parameter must be specified, the user must be forced to do so. If it must not be specified, the user should not be given an opportunity to specify it.

6. `int applicableObjects() const`

   The *objects* parameter of a measurement definition can have different types (c.f. Section 2.2.1.1.2). This method returns the allowed types for this metrics. The result type of this method is a bitwise OR of the values: SYSTEM, SITE, NODE, PROCESS, FILE.

   Rationale: Like *usesObjects()* this method is mainly included for user guidance in the user interface.

7. `UseInfo usesPartners() const`

   Like *usesObjects()*, however, this method refers to the partner objects of a measurement.

8. `int applicablePartners() const`

   Like *applicableObjects()*, however, this method refers to the partner objects of a measurement.

9. `UseInfo usesRegions() const`

   Like *usesObjects()*, however, this method refers to the *regions* parameter of a measurement.

10. `int applicableSummaryMode() const`

    This method returns the summary modes which can be used for a measurement with this metrics. The result is a bitwise OR of the values: SINGLE_MEASUREMENT, FOR_EACH_OBJECT, FOR_EACH_PARTNER, FOR_EACH_OBJECT_AND_PARTNER (c.f. Section 2.2.1.1.2).

11. `int applicableIntegrationMode() const`

    Returns the integration modes which can be used for a measurement with this metrics. The result is a bitwise OR of the values: SINCE_START, SINCE_LAST_READ_DIVIDE_BY_TIME, SINCE_START_DIVIDE_BY_TIME, SINCE_LAST_READ (c.f. Section 2.2.1.1.2).

12. `bool definesVirtualTime() const`

    Returns *True*, if this metrics defines a virtual time. See Section 2.2.1.1.2 for an explanation of this concept.

13. `bool isEditable() const`

    Returns *True*, if the definition of this metrics can be edited by the user. In G-PM, this is the case for the metrics defined by the HLAC.

14. `MetricsList getSubMetrics() const`

    Since the number of different metrics may be quite high, metrics can be organized in a hierarchical way. This method returns a list of all metrics that are direct children of a metrics in this hierarchy.

Rationale: We may have a metrics "delay in communication". Its sub-metrics would e.g. be "delay in sending", "delay in receiving", "delay in global operations", etc. The metrics "delay in sending" again could have sub-metrics like "delay in MPI_Bsend", "delay in MPI_Isend", etc.

15. `bool isSampled() const`

    Returns *True*, if this metrics is based on sampling an instantaneous value, rather than computing it from events in the target system.

16. `static MetricsList getTopMetrics()`

    This static method returns the list of all top-level metrics (i.e. metrics which are not sub-metrics of other metrics) known in the system.

    Implementation Note: The constructor of class *Metrics* will insert the new object in a static list, if it is a top-level metrics. This method just returns this list. Note that the constructor as well as the static list are not part of the public interface!

17. `static MetricsList getEditableMetrics()`

    This static method returns the list of all editable metrics (i.e. user-defined metrics) known in the system.

18. `static Metrics* findMetrics(string name)`

    This static method searches for a metrics with the given name and returns a pointer to it.

2.2.1.1.2. MeasurementSpec

This class comprises all data that is necessary to specify a measurement, i.e. the metrics to be measured and all other parameters. It offers the following methods:

1. `MeasurementSpec()`
2. `~MeasurementSpec()`

    Constructor and destructor.

3. `MeasurementSpec(const MeasurementSpec& from)`

    Copy constructor

4. `MeasurementSpec& operator=(const MeasurementSpec& rhs)`

    Assignment operator

5. `void setName(string measurementName)`
6. `string getName() const`

    Set and get the measurement name. The name can be arbitrary and has no influence whatsoever on the measurement itself.

    Rationale: The measurement name is provided solely for the user interface. It helps the user to organize and remember his measurements.

```
7.  void setMetrics(Metrics& metrics)
8.  Metrics* getMetrics() const
```

Set and get the metrics. The object will store a reference to the metrics object.

Rationale: Metrics are regarded as global, unchangeable objects. Thus, it doesn't really make sense to copy the metrics.

```
9.  void setObjects(const AppObjectList& objects)
10. AppObjectList getObjects() const
```

Set and get the objects (sites, nodes, processes) where the measurement should be performed. An empty list denotes the whole system. The list is copied by both methods.

```
11. void setPartners(const AppObjectList& partners)
12. AppObjectList getPartners() const
```

Set and get the partner objects (sites, nodes, processes, files) for a measurement. An empty list denotes a wildcard, i.e. any partner object. The list is copied by both methods.

Rationale: A partner object can e.g. be used for metrics like "amount of communication". When a list of partners is specified, only the communication with these partners is taken into account for the measurement.

```
13. void setRegions(const RegionList& regions)
14. RegionList getRegions() const
```

Set and get the code regions (modules, functions) where the measurement should be performed. An empty list denotes the whole program. The list is copied by both methods.

Rationale: If a list of regions is specified, the measurement will be restricted to these code regions. E.g. this allows measuring the "amount of communication" for a specific function in a program.

```
15. void setSummaryMode(SummaryMode sMode)
16. SummaryMode getSummaryMode() const
```

The summary mode specifies to which extent measurement values should be summarized for the different elements of the objects, partners and regions lists. *SummaryMode* is an enumeration type containing the following values:

SINGLE_MEASUREMENT	The result of the measurement is a single value, which is the sum of the results for all objects, partners and regions.
FOR_EACH_OBJECT	The result is an array with one element for each object. Each element is the sum of the measurement results for all partners and regions.
FOR_EACH_PARTNER	The result is an array with one element for each partner. Each element is the sum of the measurement results for all objects and regions.
FOR_EACH_OBJECT_AND_PARTNER	The result is a two-dimensional array with one element for each pair (object, partner). Each element is the sum of the results for all regions.

For example, with the “amount of communication” metrics, *SINGLE_MEASUREMENT* will result in the total amount of communication between the specified objects and their partners, while *FOR_EACH_OBJECT_AND_PARTNER* will result in the communication matrix.

```
17. void setIntegrationMode(IntegrationMode iMode)
18. IntegrationMode getIntegrationMode() const
```

Set / get the integration mode. *IntegrationMode* is an enumeration with four values, which encode the following (independent) choices:

i. whether the measurement interval is the definition interval (i.e. the time interval between the definition of the measurement and the current read of the measurement) or the update interval (i.e. the time interval between the last read of the measurement and the current read)

ii. whether the measurement value should be divided by the length of the measurement interval or not.

The enumeration values are:

SINCE_START:	In this case, the result of the measurement is an integral over time, starting with the start of the measurement. E.g. for the 'amount of communication' metrics, the result would be the total number of bytes communicated since the start of the measurement.
SINCE_LAST_READ_DIVIDE_BY_TIME:	For this mode, the result is the difference between the current integral value and the integral value at the time of the previous reading of the measurement, divided by the time difference. E.g. for the 'amount of communication' metrics, the result would be the obtained communication bandwidth.
SINCE_START_DIVIDE_BY_TIME:	In this mode, the returned value is the aggregated value since the measurement was started, divided by the length of the measurement interval. I.e. the result is something like a mean derivative.
SINCE_LAST_READ:	In this mode, the result is the difference between the current integral value and the integral value at the time of the previous reading of the measurement. It is useful only for internal purposes (mostly for used-defined metrics).

The explanation above assumes a metrics with *isSampled() = False*. For sampled metrics, the integration mode should be ignored.

Rationale: This feature is mainly needed to support the implementation of the user-defined metrics outlined in Section 2.2.3.

```
19. void setTimeMode(TimeMode tMode)
20. TimeMode getTimeMode() const
```

Set / get the time mode. *TimeMode* is an enumeration with the values *REAL_TIME* and *VIRTUAL_TIME*.

In *REAL_TIME* mode, when reading the measurement, the result(s) are the value(s) for the current time (or for the interval between the time of the last read and the current time).

In *VIRTUAL_TIME* mode, the result of reading the measurement is a sequence of values, one for each tick of the virtual time, since the last read operation. The virtual time ticks whenever an event (which depends on the measured metrics) occurs in the monitored application. For metrics that are computed from a start and an end event, usually the end event defines virtual time. The virtual time then just is the number of executions of this end event. E.g. the "amount of communication" metrics is computed by monitoring the beginning and end of each communication call. A *VIRTUAL_TIME* measurement of this metrics thus results in the sequence of message lengths for each communication call.

Rationale: The *VIRTUAL_TIME* mode has been introduced mainly for application-specific metrics. It will be very useful to measure e.g. the amount of communication or the CPU time of each single iteration of the application's main loop, rather than providing only averaged values.

21. `ActiveMeasurement* measure(MeasurementCB& callback)`

Defines a measurement with the metrics and parameters stored in the *MeasurementSpec* object. The method returns an *ActiveMeasurement* object that represents the ongoing measurement.

The *callback* parameter is a reference to an object implementing the interface

```
class MeasurementCB {
    virtual void processMeasurementResults(
            ActiveMeasurement measurement,
            ValueList values,
            void *data) = 0;
}
```

The method *processMeasurementResults()* of this object will be called when the results of the measurement are available during or after a call to the *read* method of the returned *ActiveMeasurement* object. For measurements in *REAL_TIME* mode, the *ValueList* passed to this method always contains only one element. The *data* parameter passed to this method contains the pointer passed to the *data* parameter of the *ActiveMeasurement::read()* method.

Implementation Notes:

a. This method calls a method *ActiveMeasurement* Metrics::measure (MeasurementSpec&, MeasurementCB&)*, which does all the work. This is necessary, since the knowledge on how a measurement is implemented is encapsulated in the *Metrics* classes.

b. The *MeasurementSpec* object stored in the *ActiveMeasurement* object returned by this method is a copy of the current object. Reason: The current object can be modified after a measurement is defined. In this case, the information in the *ActiveMeasurement* object would be inconsistent.

2.2.1.1.3. ActiveMeasurement

An object of (an implementation class derived from) this abstract base class is created whenever a concrete measurement is defined. Thus, the object represents the ongoing measurement. Methods of this object allow to manipulate (start / stop / delete) the measurement and to read its results.

The interface of *ActiveMeasurement* defines the following methods:

1. `~ActiveMeasurement()`

   This destructor not only deletes the object, but also deletes the measurement in the target (i.e. the monitored application).

   Note: there is no constructor in the interface, since the only way to create an *ActiveMeasurement* object is via *MeasurementSpec::measure ()*.

2. `void start(bool flush)`

   Starts a measurement. When a measurement is defined, all necessary instrumentation is performed in the target, i.e. in an object being monitored, however, the measurement is not yet started. It is started only after the *start* method is called.

   The *flush* parameter defines the buffering behaviour of the necessary communication with the monitoring system. If set to *True*, all communication buffers are flushed before the method returns.

   Rationale: The separate *start* method results in a two-phase commit when defining measurements. During the definition, all possible sources of errors are checked (esp. the parameters of the measurement). If an error occurs, the definition can be deleted without any effect. The *start* method itself normally can not result in errors.

   Implementation Note: Somewhere we should store the exact information when a measurement was started / stopped, since this information is needed for computing correct performance values. This does not happen in the current implementation yet.

3. `void stop(bool flush)`

   Stops a measurement. This is useful to temporarily disable measurements.

   The *flush* parameter defines the buffering behavior of the necessary communication with the monitoring system. If set to *True*, all communication buffers are flushed before the method returns.

   Implementation Note: As with *start()*, we should somewhere store the exact time.

4. `void read(bool block, bool flush, void *data)`

   Read the results of a measurement. The results are not returned by this method. Rather, the callback specified during the measurement definition is invoked when the results are available. In this way, asynchronous receipt and processing of measurement results is possible.

   The *block* parameter defines the blocking behavior of this method. If set to *True*, the method blocks until all results have been passed to and processed by the callback. Otherwise, the method returns immediately and the callback is called asynchronously at a later time. Note that until the call to *read* is actually finished, further calls to this method have no effect other than probably blocking until the read operation is finished.

   The *flush* parameter defines the buffering behavior of the necessary communication with the monitoring system. If set to *True*, all communication buffers are flushed before the method returns.

   The *data* parameter is an arbitrary pointer that is passed to the callback. It can be used to pass context information to the callback.

5. `MeasurementSpec* getMeasurementSpec()`

Returns the measurement specification of this measurement, i.e. the metrics and all its parameters used for defining the measurement. See Section 2.2.1.1.2 for a documentation of *MeasurementSpec*.

Implementation Note: This method should really return either a copy of the *MeasurementSpec* object or a *const* reference, since the object must not be changed!

6. `bool getMonAccessInfo(MonAccessInfo& info) const`

   Returns information on whether and how the results of this measurement can be read from within an OCM-G extension. The result is *False*, if the measurement result can not be read within an OCM-G extension.

   Rationale: This method is provided mainly for the HLAC, but may also be useful if the Measurement Interface is used as an API for performance measurements. When measurement values have to be combined with other information (e.g. other measurement values, or events occurring in the monitored application), it is more efficient to combine the information near to its source. I.e. if we have to combine to measurement values originating from the same compute node, we should do this on that node. The OCM-G allows implementing such a scheme, because it can be (dynamically) extended with tool-specific services. The method *getMonAccessInfo()* will return the necessary information for such a service.

2.2.1.1.4. ActiveMeasurementList

This class implements a list of *ActiveMeasurement* objects. Besides the normal methods for list classes, it also implements the methods *start()*, *stop()*, and *read()* of class *ActiveMeasurements*, properly extended for lists of measurements. As far as possible, the implementation should ensure that the measurements are started, stopped, or read at the same time.

2.2.1.2. Auxiliary classes

There are the following auxiliary classes used in the Measurement Interface: *AppObject*, *FileObject*, *TokenizedObject*, *Token*, *Region* and *Value*. In addition the following classes are used to represent lists: *ActiveMeasurementList*, *MetricsList*, *AppObjectList*, *TokenList*, *LocationList*, and *ValueList*. All these classes along with their relationships are depicted in **Figure 2.**3.

The *AppObject* class is used to represent different application-related objects in G-PM. By its subclasses, it can represent a disk file, a process, a host or a site. Additionally, the *AppObject* class provides functionality to obtain the list of all available sites, the list of all nodes in a particular site (or set of sites), and the list of all processes on a particular node (or set of nodes).

The *FileObject* is a class derived from *AppObject*. It is used to represent a single disk file.

The *TokenizedObject* is a class derived from *AppObject*. It is used to represent a monitoring service object, i.e. tokens, such as a host identifier. It is an encapsulation of a token class to provide a consistent interface to access both files and tokens as *AppObjects*.

The *Token* class contains a single monitoring service object identifier. The available object types include *site*, *host*, *process* and *monitoring system conditional service request* (CSR).

The *Region* class represents a continuous code region in a process executable. This region is specified by its beginning and ending address. Usually, regions are used to represent functions or modules in the process executable. For this reason, the *Region* class provides functionality to retrieve code regions of all functions or modules in the process executable.

The *Value* class represents the result of a measurement. In general, all results are two dimensional matrices although each dimension may contain just a single value, thus allowing for results that are one dimensional arrays or single numbers. In addition, the *Value* class provides a functionality to retrieve the timestamp of the beginning or end of the measurement and, if applicable, to retrieve the virtual time that is associated with the measurement.

The ActiveMeasurementList, *MetricsList*, *AppObjectList*, *TokenList*, *LocationList*, and *ValueList* classes are used to represent lists of objects of the following classes: *ActiveMeasurement*, *Metrics*, *AppObject*, *Token*, *Region*, and *Value*. Except for *ActiveMeasurementList* all these classes implement simple list functionality and will be based on list classes from the Standard Template Library [STL]. The *ActiveMeasurementList* class additionally provides a functionality to simultaneously start, stop, and read all measurements that it contains.

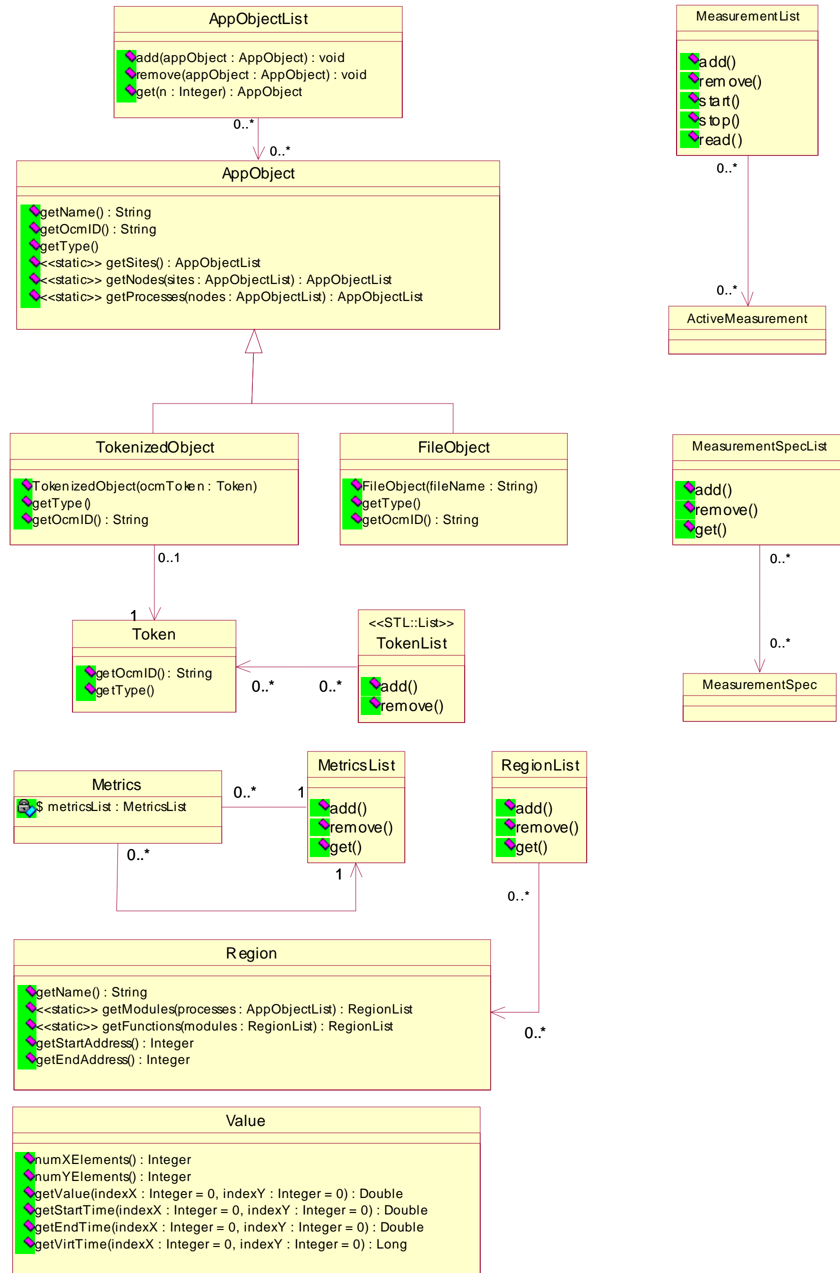


Figure 2.3 Class diagrams of auxiliary classes

The PMC and HLAC provide an identical interface that allows to define performance measurements and to read their results. This interface consists of two C++ abstract base classes (*Metrics* and *ActiveMeasurement*) and a number of non-abstract classes. The PCM and HLAC provide different implementations for the abstract base classes, depending on the way how measurements are realized. The other, non-abstract classes are independent of the realization of measurements.

2.2.2. Performance Measurement component (PMC) class diagrams

Performance Measurement Component consists of a set of classes that are used to carry out a pre-defined set of fixed measurements. Therefore, it consists of a level of abstract classes representing various common aspects of measurements and a level of inherited classes, defining broad categories of above-mentioned pre-defined measurements.

In general, each measurement can be described as 3 complementing parts. The first one is the specification of what values to measure, with corresponding information on how to measure them, e.g. information on what “communication volume” is to be measured, and that for that measurement the monitoring service “counter” type object is to be used. This information is specified in the *Metrics* class.

Then a set of objects, on which the measurement is carried out, is specified in the *MeasurementSpec* class. This can be information e.g. on host name, process identifier etc., thus specifying the measurement as “*communication volume on host foo.bar within process 883*”.

Finally, for each measurement a set of low-level monitoring service identifiers, such as an identifier of the above-mentioned counter, has to be stored throughout the time of measurement. This data is contained in the *ActiveMeasurement* class. This class is also an interface to a running measurement for other parts of G-PM, such as a “Visualisation Window”.

2.2.2.1. Specific measurements classes

It should be noted that classes *ActiveMeasurement* and *Metrics*, being abstract classes for, respectively, a specific measurement and measurement type, are both overloaded in the PMC module. Each inherited class is responsible for handling a broad range of measurement types. In the PMC module, there are two such generic types: sampled measurement and function-based measurements.

2.2.2.1.1. Sampled measurement classes

Sampled measurements consist of those attributes of the application and its environment that change in a continuous manner independently from the application-generated events, thus allowing only for obtaining values sampled at some interval. An example of such metric could be a host's CPU load.

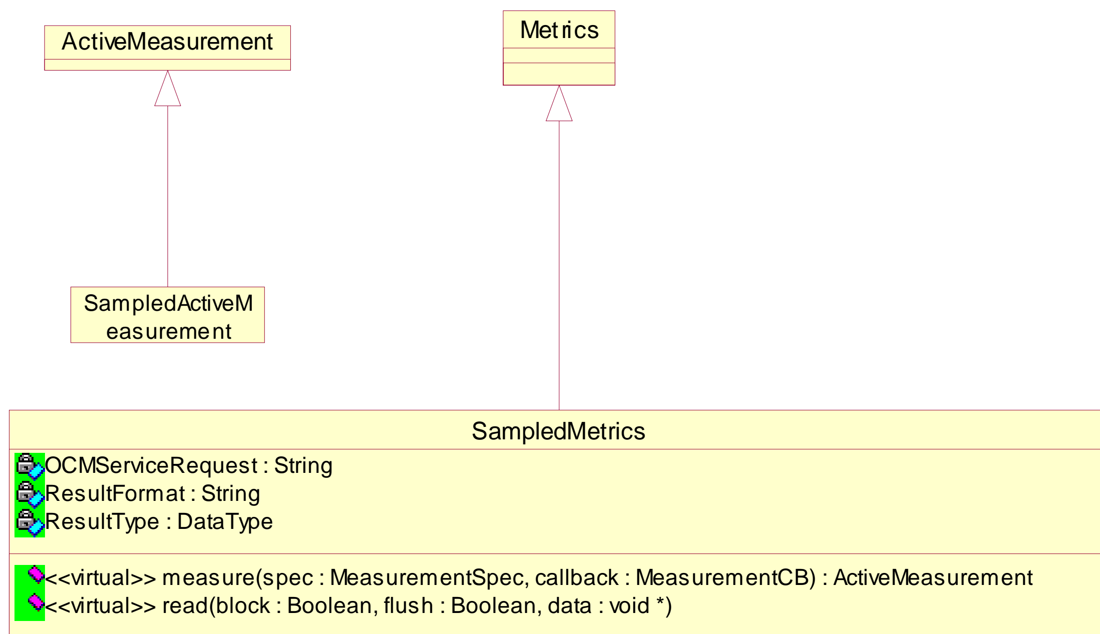


Figure 2-2 Class diagrams of sample-related metrics and sample-related active measurements

Sampled measurements are handled by *SampledMetrics* class, which itself knows how to measure the metrics. There will exist several instances of that class, one per each sampled metrics type offered by the monitoring services. These instances differ only in private attribute members specifying a command string for monitoring services and properties of a metric, e.g. for a CPU load metric there will exist an instance of the class with an OCM-G command string containing service request appropriate for measuring the CPU load.

It should be noted that such a measurement does not require initialisation of any type and does not need to store any monitoring service objects identifiers. Therefore, since each sampled measurement can be carried out solely by *SampledMetrics* class, the accompanying *SampledActiveMeasurement* class does not overload any members of *ActiveMeasurement* class. Both of those classes are depicted in Figure 2-2.

2.2.2.1.2. Function-based measurement classes

Function-based measurements are another broad class of metrics. Each function-based measurement is built on conditional events associated with a function call or a set of function calls: starting a call to the function and finishing the call to the function. The value of the metric is, in principal, based on the parameters passed or returned by the measured function.

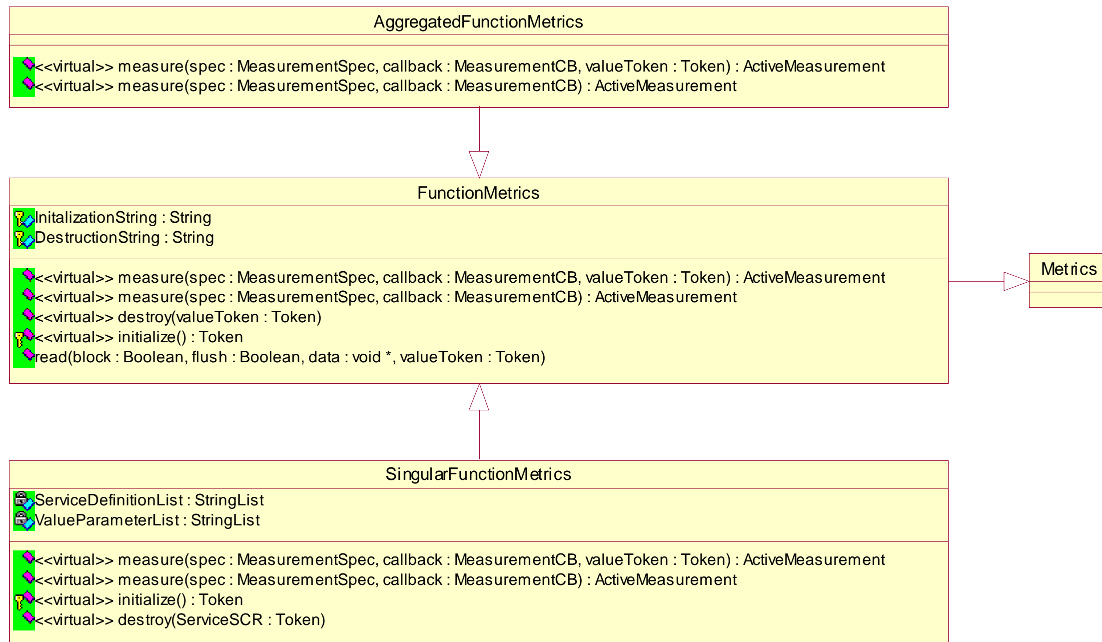


Figure 2-3 Class diagram of function-related measurements

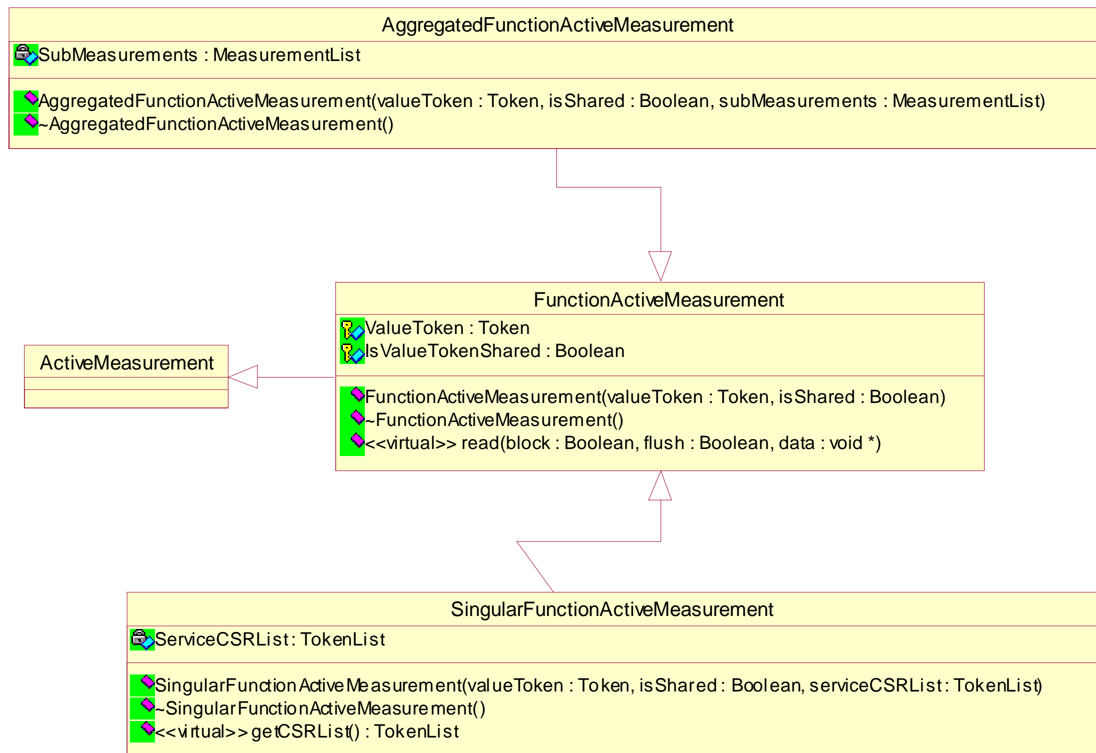


Figure 2-4 Class diagram of function-related active measurements

The first of a set of classes associated with function-based metrics is the *SingularFunctionMetrics* class, which is inherited form *FunctionMetrics* class described below. It specifies how to measure a property, i.e. a singular function and function parameters the metric is based on. It also contains information on how to create and destroy some objects in the monitoring service, which are necessary for carrying out the measurement. On the other hand, the accompanying *SingularFunctionActiveMeasurement* is a placeholder for the identifiers of these objects, thus allowing to have several measurements based on one *SingularFunctionMetrics* class object. These objects are passed to *SingularFunctionMetrics* class object when a communication with monitoring services is to occur.

It should be noted that there is often a need to treat several functions as one class of functions. The set of MPI send functions consisting of e.g. blocking and non-blocking send functions is one example. To handle such a case in an easy manner, the classes *AggregatedFunctionMetrics* and *AggregatedFunctionActiveMesurement* have been specified. In the above-mentioned example, the “*MPI_Isend*” and “*MPI_Issend*” would be singular function metrics, while “non-blocking send” metric would be an aggregated function metric, which itself would be aggregated within “send” metric.

The *AggregatedFunctionMetrics* class allows to treat several *FunctionMetrics* as a single metrics. It specifies how to initialise and measure the metrics with the help of the underlying *FunctionMetrics*

class objects. The *AggregatedFunctionActiveMeasurement* class is, in principal, responsible for managing the underlying *FunctionActiveMeasurement* class objects.

The classes *AggregatedFunctionMetrics* and *SingularFunctionMetrics* are both inherited from a Metrics class via a *FunctionMetrics* class. This class combines common interfaces of function-based metrics, thus allowing to aggregate not only singular, but also aggregated function metrics. Therefore it is possible to create multi-level hierarchy of function-based metrics. Within the same approach, the classes *AggregatedFunctionActiveMeasurement* and *SingularFunctionActiveMeasurement* are inherited from *FunctionActiveMeasurement* class, which itself is inherited from *ActiveMeasurement* class. The classes described above are depicted in Figure 2-4.

2.2.3. The HLAC component

The HLAC component consists of classes which support to specify the user defined and application specific metrics. Using the metrics specification Editor in UIVC the developer can write the specification of metrics using PMSL (Performance Metrics Specification Language), which is part of this component. This specification will be parsed and as a result, all the necessary information of the defined metrics will be stored in the *UDSpecifcation* class in the form of a DAG (Directed Acyclic Graph). When the measurement is defined, the DAG will be evaluated and at the same time, a DFG (Data Flow Graph) will be created to collect the result values which will be received asynchronously. The DFG will be then analysed and the sub-dataflow graphs which can be evaluated locally on their appropriate host will be distributed using the OCM-G system. Since we need new OMIS services which deal with the dataflow graphs, a HLAC-specific OCM-G extension is provided.

2.2.3.1. Specific HLAC classes

First of all the class diagrams for all classes in HLAC will be presented in this section. To have a user friendly presentation of the class diagrams, we omit methods and attributes in these diagrams. For more details, please see the documentation of HLAC which is generated by DOXYGEN. This documentation is available in the *apidoc* directory of the G-PM sources. If necessary, generate it by invoking '*make apidoc*' in the top-level directory of the G-PM sources.

Next to the class diagrams, some sequence diagrams as examples will be presented to show the interaction of the objects of this component. After this, a detailed description of how those metrics are processed and how the distributed evaluation of the performance measurement are realised will be shown.

2.2.3.1.1. Specific classes for user defined metrics

*UDSpecification (*User Defined Specification)

This class contains all the necessary information about a user defined metrics. It is constructed with the help of a parser for the Performance Metrics Specification Language PMSL. The parser has been constructed using the tools Flex for scanning and Bison for parsing.

In general, when a new metrics is defined, the class will parse the PMSL specification, create a Directed Acyclic Graph (DAG) as an internal representation of this specification, and optimize it.

The class also is responsible for the symbol table management during the PMSL parsing. It offers methods to create entries for local variables which are declared in the body of the metrics specification, as well as parameters of the specified metrics. Each symbol table entry is a pointer to an identifier node in the DAG (see class *IdentifierNode* below).

Once a measurement of the metrics is requested, the *evaluateDAG* method of this class does a partial evaluation of the DAG, optimizes it, creates a dataflow graph controlling the measurement's evaluation, defines all the child measurements and OMIS conditional requests necessary for the measurement, and creates the *ActiveMeasurement* object. When metrics are specified to receive scalar parameters, but the user defined a measurement using **lists** of objects, several DAGs will be created, one for every combination of the parameters in the lists. These DAGs are then combined into a single DAG using an "ADD" operator (i.e., the result of the measurement is an aggregated value, comprising all possible combinations of the parameters).

The constructor of this class is called from the PMSL parser and receives the name of the user defined specification, which will be used as the name of the corresponding metrics in the navigation part of the user interface of G-PM.

UDMetrics (User Defined Metrics)

After creating an object of this class, which is a subclass of *Metrics*, the proper PMSL specification must be attached to it via the *setSpecification* method. This method invokes the PMSL parser to create an *UDSpecification* object, which is then stored in the *UDMetrics* object. The method also receives a callback object, which will be called whenever an error is occurring in the process of parsing. Only when parsing was successful, the *UDMetrics* object is appended to the list of available metrics.

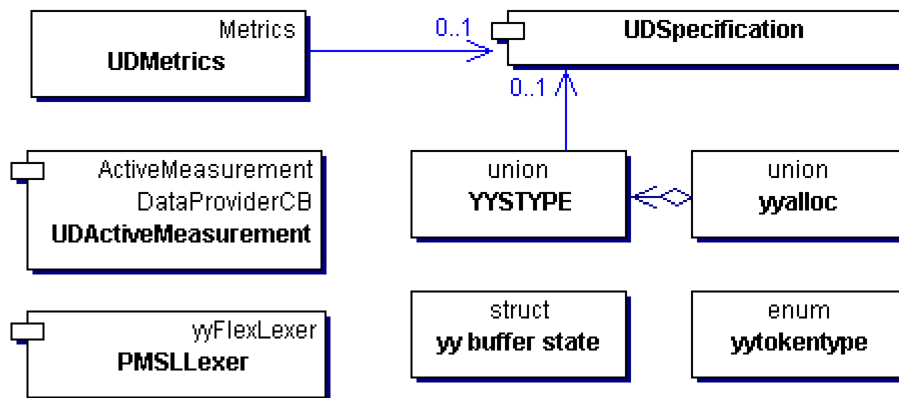


Figure 2-5: Class diagram for the top-level HLAC classes

During the definition of a measurement based on this user defined metrics, the actual measurement specification object (*MeasurementSpec*) and measurement callback object (*MeasurementCB*) will be given as parameter for the *measure* method of this class. This method then prepares the measurement with the help of the *UDSpecification* class and its method *evaluateDAG* (see above).

UDActiveMeasurement (User Defined Active Measurement)

The objects of this class represent an active measurement of a user defined metrics. This is a subclass of the abstract class *ActiveMeasurement* as mentioned in Section 2.2.1.1.3. This active measurement is returned as a result of the *measure()* method of *UDMetrics*.

This class contains all the information, which is needed for enabling/disabling, reading, and deleting the measurement:

- all active measurement objects created for the inferior measurements, i.e. measurements of metrics used in the specification of the user defined metrics,
- a list of OMIS conditional request tokens for all conditional requests sent to OCM-G for monitoring probe events, which are used for enabling, disabling, and deleting these requests,
- all active measurements which must be read in order to get a result for this measurement,
- a list of aggregation data flow nodes (see Section 2.2.3.1.3) which must be read after the active measurements above

Besides the top-level classes documented above, the HLAC consists of two packages, which implement directed acyclic graphs (as an intermediate representation for metrics specifications) and data flow graphs (for the evaluation of measurements). These packages are described below.

2.2.3.1.2. Specific classes for the directed acyclic graph (DAG)

The DAG representing a metrics specification, which is created from the PMSL parser, consists of different types of nodes, whose common behaviour is implemented in the base class *Node*. The methods provided by this class allow to build the DAG and to perform some manipulations on the DAG, like replacing nodes with others, common sub expression elimination (CSE), or simply printing the DAG. Since for most of these manipulations, a recursive traversal of the DAG is necessary, the *Node* class is a quite complex one, with a lot of methods of rather high complexity as was remarked by the CrossGrid Quality Assurance, however, it is a necessity caused by the complex DAG algorithms, which cannot easily be changed.

In addition, the class implements methods to partially evaluate the DAG, which results in the creation of a dataflow graph. This graph is used both to define the actual measurements and the necessary OMIS requests for monitoring the probes, as well as for the evaluation of the measurements' results.

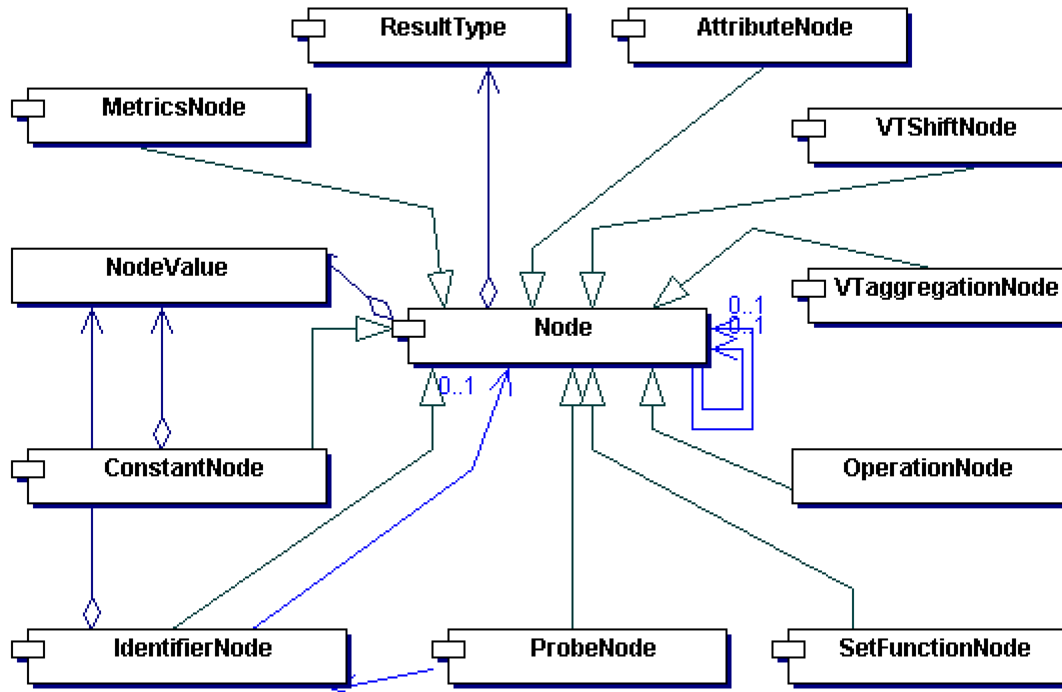


Figure 2-6*:* Inheritance diagram for the DAG classes

The subclasses of this class are briefly described below. They represent the different kinds of DAG nodes.

ConstantNode**:**

This class represents constants values in the DAG.

A constant mainly consists of its value and a type. At the moment, during construction of the DAG, only constants of types *double, int*, *bool*, and *TimeInterval* are needed.

IdentifierNode**:**

This class represents identifiers in the DAG.

An identifier has a name, a type, and an optional value. During the construction of the DAG, this value is again a DAG. Only during the evaluation, real values are assigned to identifiers (namely parameters and global variables).

ProbeNode**:**

This class implements DAG nodes representing the use of probes.

Similar to a *MetricsNode*, a *ProbeNode* just stores information on the probe and its parameters. However, instead of the name, a pointer to the *IdentifierNode* of the probe is stored.

***OperationNode*:**

This class implements DAG nodes representing arithmetic and logical operations. At the moment, we have the following unary and binary operations:

- Plus, minus, times, div, mod, and unary minus for operands of type *double*, *int*, and *Value*,
- Equality and inequality for all scalar data types,
- the other comparison operators only for operands of type *double*, *int*, and *Value*,
- AND, OR, and NOT for Boolean operands,
- the IN operator, which checks whether an element is in a list,
- the special AT operator, which determines the value of an expression when an event occurs.

In addition, there are n-ary operations, which are only defined for types *double*, *int*, and *Value*: ADD, MULT, MIN, MAX, MEAN, and STDEV.

SetFunctionNode:

This class implements DAG nodes representing aggregation functions, like 'sum over all elements of a specified set'.

Currently, seven aggregation functions are defined:

- SUM: the sum of the elements. Elements must be of type *double*, *int*, or *Value*
- PROD: the product of the elements. Elements must be of type *double*, *int*, or *Value*
- MIN: the minimum of the elements. Elements must be of type *double*, *int*, or *Value*
- MAX: the maximum of the elements. Elements must be of type do*u*ble, *int*, or *Value*
- MEAN: the mean value of the elements. Elements must be of type *double*, *int*, or *Value*
- STDEV: the standard deviation of the elements. Elements must be of type *double*, *int*, or *Value*
- COUNT: the number of elements. Elements can be of any scalar type.
- UNIQUE: returns an arbitrary element. Elements can be of any scalar type. The idea is that typically, the set is defined in such a way that it contains only one element.

A *SetFunctionNode* has one or two children. The first one is the expression producing the elements of the set when the iterators iterate over their value range. The second one is a *Boolean* expression over the iterators. An element is taken into account for the computation only if this expression evaluates to `true`. If the expression is missing, all elements are used for the computation, i.e. there is no further restriction on the iterators.

AtrributeNode:

This class represents DAG nodes which compute attributes of objects.

At the moment, attributes:

- `.time` can be applied to a *Value* and returns its time stamp.
- `.duration` can be applied to a Value and returns the length of its measurement interval.
- `.upper` can be applied to a *TimeInterval* and returns its upper bound.
- `.site` can be applied to a *Node* or *Process* and returns its site.
- `.node` can be applied to a *Process* and returns its node.
- `.rank` can be applied to a *Process* and returns its rank, i.e. its global ID.

Nodes of this class always have exactly one child node.

VTaggregationNode:

This class implements DAG nodes representing the aggregation over time.

This operation is defined only for types *double*, *int*, and V*a*lue. The possible aggregation operators are: SUM, PROD, MIN, MAX, MEAN, and STDEV.

MetricsNode :

This class implements DAG nodes which represent uses of (``calls to'') already defined metrics.

The information stored is just the name of this metrics and the parameter expressions, which are the child nodes of the *MetricsNode*.

2.2.3.1.3. Specific classes for the dataflow g*raph* (DFG)

A dataflow graph consists of *DataflowNode* objects, which are linked together via *DataflowDataProvider* objects, which act as a data channel. The *DataflowNode* class represents the operations in the dataflow graph, while the *DataflowDataProvider* class delivers the operators. Both classes together implement a dataflow driven evaluation: *DataflowNode* objects wait until all their child data providers have their results ready and then collect and remove all the corresponding values from them. The dataflow node then calculates its result, based on the node type and the collected values, and pushes this result to the data provider of its parent node(s). The *DataflowDataProvider* class implements a callback mechanism that is used to notify the proper dataflow nodes when an operand is available.

The *DataflowNode* class possesses several subclasses for the different kinds of operations. Each of these classes corresponds to a subclass of the DAG *Node* class.

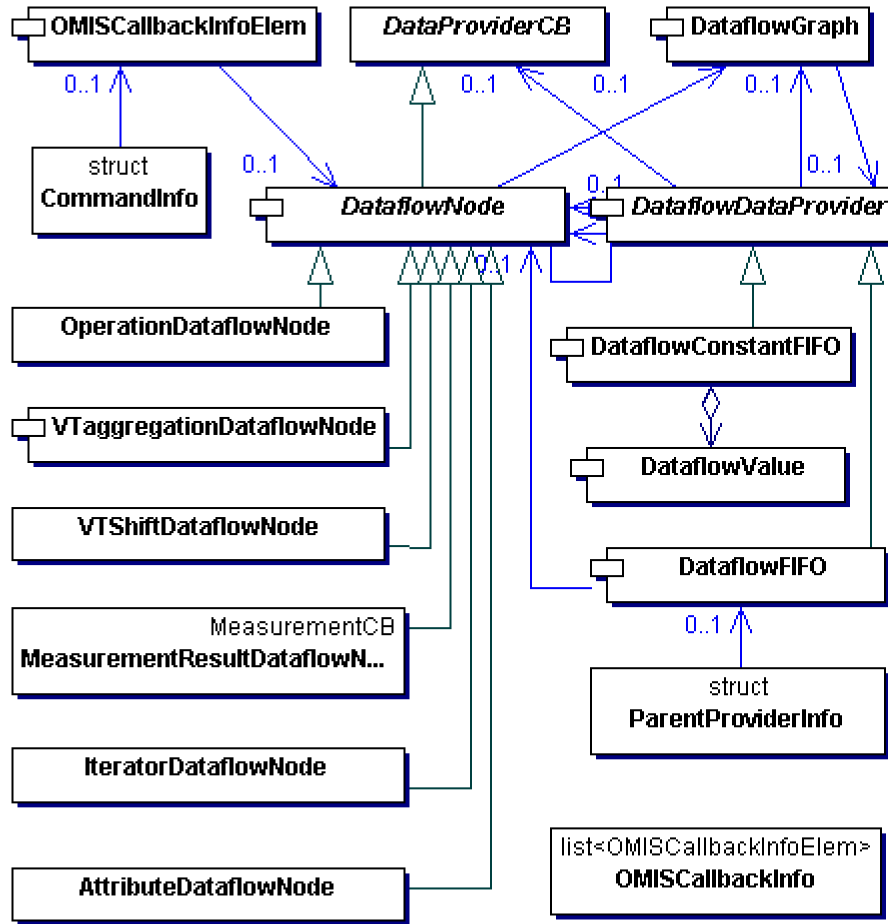


Figure 2-7: Inheritance diagram for DFG classes

AttributeDataflowNode:

This class implements the dataflow node for the corresponding *AttributeNode* in the DAG. Since all attributes except the `.duration` attribute are evaluated when the measurement is defined, the `.duration` attribute is the only one implemented by this class.

OperationDataflowNode:

This class represents the dataflow node for its counterpart *OperationNode* from the DAG. This class collects values from the child data providers, executes the defined operation on the list of the results and writes the result value to the parent data provider(s).

MeasurementResultDataflowNode :

This contains the counterpart of the *MetricsNode* from the DAG. The node receives measurement results from the callback of its corresponding active measurement and forwards these results to its parent data provider(s).

IteratorDataflowNode:

This class implements a dataflow node, whose value can be explicitly set via a method call. The value is then forwarded to its parent data provider(s). This class is used when data from a probe needs to be injected into the dataflow graph.

VTaggregationDataflowNode:

This class implements a dataflow node for its corresponding *VTaggregationNode* in the DAG, which deals with the aggregation of measurement values over time. This node aggregates its operand values in an internal variable and forwards its value to the parent data provider(s) only when explicitly requested.

The *DataflowDataProvider* class has two subclasses:

DataflowFIFO:

This class implements a first-in-first-out queue (FIFO) with the help of the Standard Template Library STL. It is used to ``connect'' the *DataflowNode*s in a dataflow graph, implementing the necessary buffering.

DataflowConstant:

This class implements a special data provider for constant values. Conceptually, it can be viewed as a FIFO which contains an unlimited number of identical, predefined values. The class allows introducing constants in the dataflow driven evaluation scheme, without a need to handle this case especially in all the dataflow nodes.

2.2.3.1.4. OCM-G extension for distributed evaluation

G-PM tries to evaluate the DFGs corresponding to a measurement of a user-defined metrics in a distributed way. In order to achieve this, an extension to the OCM-G is needed. This extension provides services to deal with data flow graphs on the hosts different from the one where G-PM is executing. These services deal with the following tasks:

- Allocating a remote DFG and returning a DFG token (i.e. a location-transparent object reference)
- Adding nodes and data providers to a remote DFG
- Handling the events of data providers in a remote DFG, which should inform the appropriate dataflow node in G-PM
- Reading and writing results from/to nodes in a remote DFG
- Deleting a remote DFG when it is no longer needed.

The OCM-G extension consists of the following source files which implement the new OMIS services.

DFGtoken.C:

This file implements the token (i.e. a location-transparent object reference) for DFGs in OCM-G. It implements the functions required by OCM-G to create, localize and delete the DFG token. The token is constructed by simply combining a sequence number and the local node token. Thus this token is globally unique and easy to localize.

GenerateDFG.C:

This file implements the following OMIS services (see the OMIS document for a description of the notation):

hlac_create_dfg – create an empty dataflow graph.

```
token
hlac_create_dfg(token* nodelist)
```

On the specified nodes, a new OCM object which stores the pointer to the dataflow graph will be created and a token for this OCM object (i.e., a DFG token) will be returned as a result.

hlac_add_dfg – add a sub-dataflow graph to an existing (empty or non-empty) graph.

```
void
hlac_add_dfg(token dfg, string spec)
```

The DFG token is used to get the pointer to the ‘parent’ dataflow graph, the sub-dataflow graph is added to it, and the root nodes of the added sub-dataflow graph is stored in a list. The sub-dataflow graph is specified as an encoded string, since OMIS does not support arbitrary structures or objects as service parameters.

hlac_delete_dfg – delete a data flow graph.

```
void
hlac_delete_dfg(token* dfglist)
```

The OCM object (and in turn the DFG) will be deleted, when no more token are referring to this object.

DFGactions.C:

This file implements the following OMIS actions:

hlac_dfg_fifo_read – read a FIFO in a DFG.

```
struct dataflowValue {
    floating value;           // see class DataflowValue!
    floating startTime;
    floating stopTime;
    integer  virtualTime;
    integer  requesterID;
    integer  hasVirtualTime;
    integer  hasTime;
};
struct {
    integer       numValues; // number of DataflowValues following
    dataflowValue *vlaues;
} hlac_dfg_fifo_read(token dfg, integer fifo)
```

hlac_dfg_iterator_write – call the ***write*** method on the specified ***IteratorDataflowNode***.

```
void
hlac_dfg_iterator_write(token dfg, integer node, floating value,
                        floating time, integer virtualTime,
                        integer requesterID)
```

hlac_dfg_aggregation_read – call the ***read*** method on the specified ***VTaggregationDataflowNode***.

```
void
hlac_dfg_aggregation_read(token dfg, integer node,
                          integer requesterID,
                          integer virtualTime)
```

hlac_dfg_aggregation_start – call the ***start*** method on the specified ***VTaggregationDataflowNode***.

```
void
hlac_dfg_aggregation_read(token dfg, integer node)
```

hlac_dfg_measurement_read_and_forward – read the results of a measurement and forward them to the specified M***easurementResultDataflowNode***.

```
void
hlac_dfg_measurement_read_and_forward(token dfg, integer node,
                                      string service,
                                      any *parameters,
                                      integer requesterID,
                                      integer virtualTime)
```

FIFOevent.C:

This file contains an implementation for an event which is raised when a dataflow data provider contains values to be read (see the OMIS document for an explanation of the notation).

hlac_dfg_fifo_has_value – a FIFO received a value.

```
void
hlac_dfg_fifo_has_value(token dfg, integer fifo)
--> void          // no service specific event context parameters
```

This event is raised whenever the specified FIFO in the given DFG receives a value which causes the FIFO to make a state transition from empty to non-empty. The event does not provide any specific event context parameters. Rather, the value(s) contained in the FIFO should be read via the ***hlac_dfg_fifo_read service.***

2.2.3.2. Sequence diagrams

The following sequence diagram shows the sequence of actions for measuring a user-defined metrics, whose definition matches the following template:

```
SampleMetrics(…, VirtualTime vt)
{
    …
    return SomePMCmetrics(…,[START,NOW]) AT probe(…,vt);
}
```

Note that the names of the stereotypes are abbreviated in order to show a meaningful interaction. The abbreviations are explained below.

MS	MeasurementSpecification	UDM	UDMetrics
UDS	UDSpecification	N	Node
DFDP	DataflowDataProvider	M	Metrics
AM	ActiveMeasurement	DF N	DataflowNode
UAM	UDActiveMeasurement	OCBI	OMISCallbackInfo
OCBIE	OMISCallbackInfoElement		

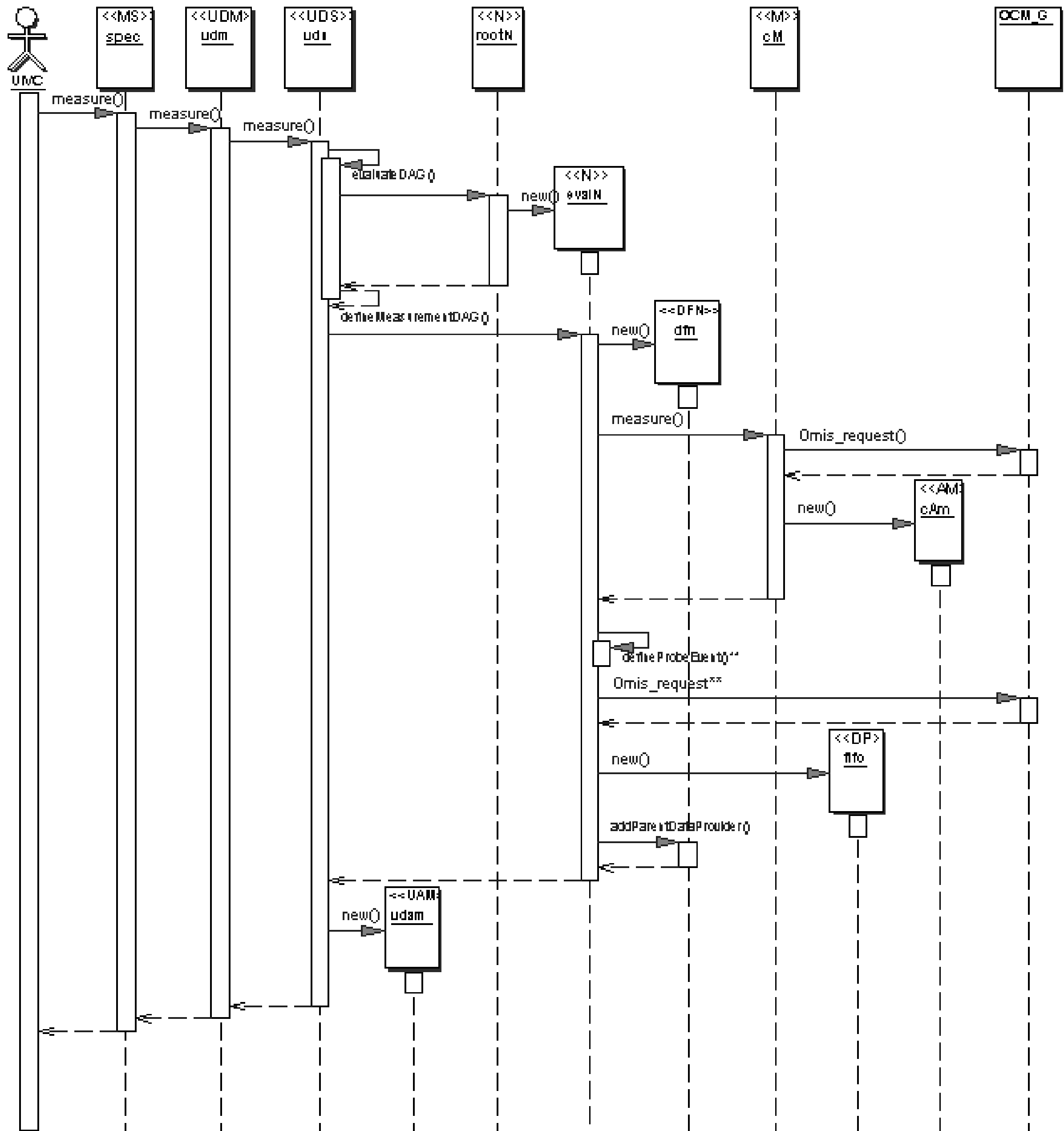


Figure 2-8: Sequence diagram for measuring a user defined metrics

The interaction which takes place when a probe event is detected by OCM-G is shown in the following sequence diagram. For each detected probe event, one measurement value is pushed into the FIFO. Once the measurement is read, the FIFO will be emptied and its content is passed to the read callback. The diagram clearly shows the asynchronous character of the OCM-G interface. Note that this diagram presents the case where the measurement is **not** evaluated in a distributed way (e.g., because child measurement ***childAM*** can only be evaluated in G-PM itself).

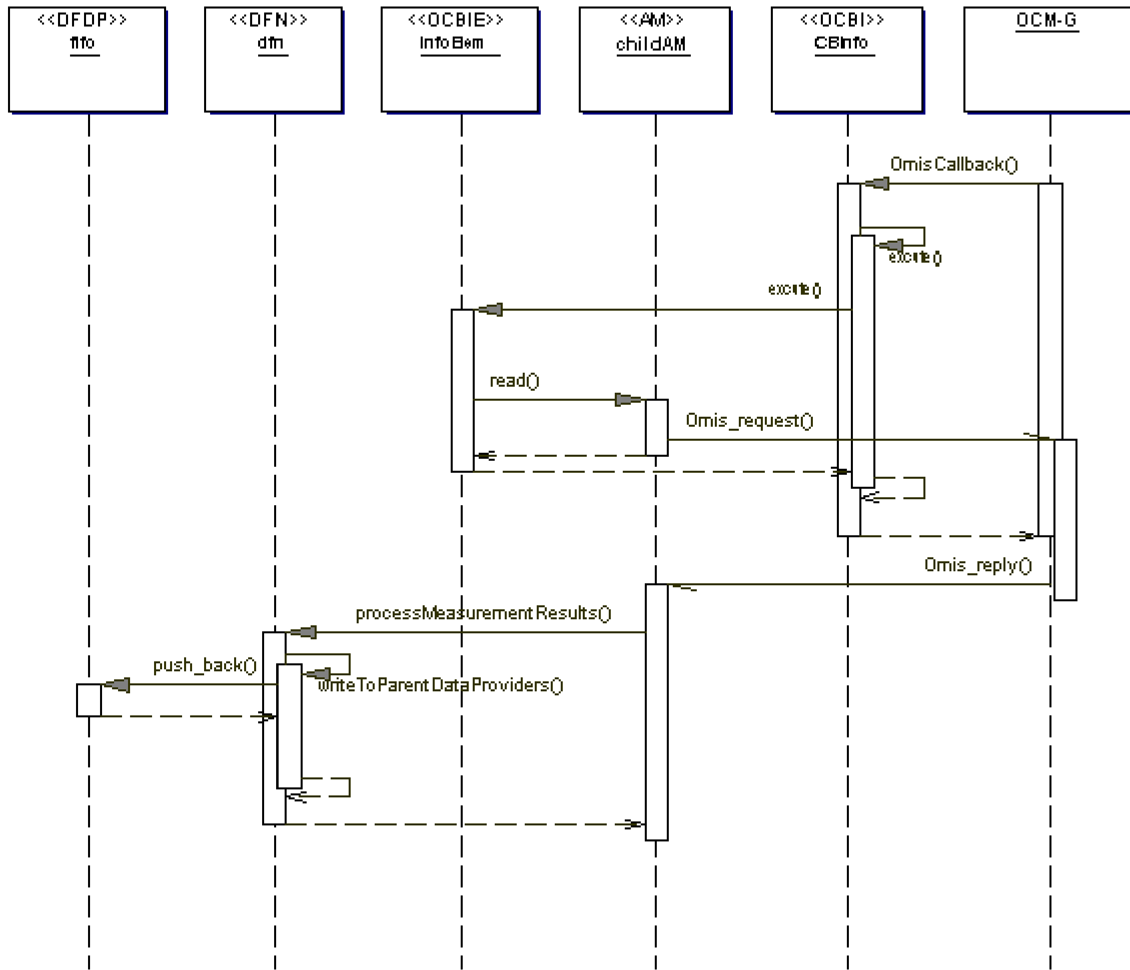


Figure 2-9: Sequence diagram during measuring a user defined measurement

The sequence diagram in Figure 2-10 shows the interactions which happen when a measurement of a user-defined metrics matching the following template is being read:

```
SampleMetrics(…, TimeInterval ti)
{
    return SomePMCmetrics(…,ti);
}
```

Note that this example is only intended to show the interactions in a clear way; it is not a particularly good example for a user-defined metrics.

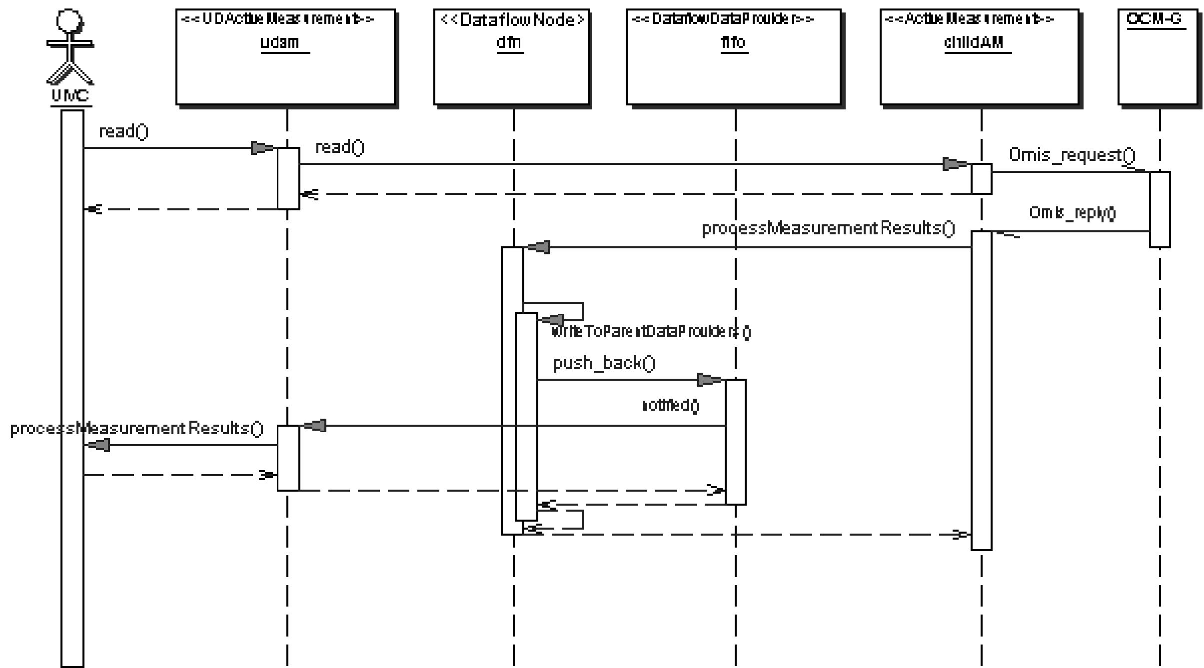


Figure 2-10: Sequence diagram during measuring a user defined measurement

2.2.3.3. Translation of PMSL and implementation of measurements

For the sake of illustrating the process of translating PMSL to DAGs and DAGs to DFNs, we will use two example metrics definitions as shown in Figure 2-11 below. The purpose of these metrics is to measure the amount of I/O caused by a specify type of user interaction in an interactive applications. We assume that the programmer inserted two probes into the application (called `begin` and `end`), marking the beginning and end of the processing of a user interactions. Probe calls belonging to the same interaction are identified by receiving the same value for the virtual time.

```
 1  IO_volume_for_interaction_vt(Process[] processes, File[] par
                                 VirtualTime vt)
 2  {
 3     PROBE end(Process p, VirtualTime vt);
 4     PROBE begin(Process p, VirtualTime vt);
 5     Process p;
 6     Value[] volume;
 7     volume[p] = IO_volume(p, partners, [START, NOW]) AT end(p
                 - IO_volume(p, partners, [START, NOW]) AT begin
 8     return SUM(volume[p] WHERE p IN processes);
 9  }
10  IO_volume_for_interaction(Process[] processes, File[] partne
                              TimeInterval time)
11  {
12     VirtualTime vt;
13     Value[] volume;
14     volume[vt] = IO_volume_for_interaction_vt(processes, part
15     return SUM(volume[vt] WHERE volume[vt].time IN time);
16  }
```

Figure 2-11: Two example metrics specified with PMSL.

In the example above two metrics are defined: `IO_volume_for_interaction_vt` specifies a metrics for the amount of disk I/O for a point in virtual time, i.e. (in our example) for a single user interaction. `IO_volume_for_interaction` is a metrics that refers to a given interval of (real) time. Its result is the I/O volume caused by all user interactions in the specified measurement time interval. See the G-PM user's manual for a more detailed description of PMSL and its features.

The translation of PMSL specifications is designed in such a way that a distributed implementation of the corresponding measurements can be realized. There are two reasons why the measurements must be implemented in a distributed fashion: First, the `AT` construct in PMSL requires a measurement value to be determined when an event occurs. In order to avoid communication delays which would adulterate the results, determining the measurement value must happen at the same place where the event occurs. Second, distributed processing of measurement data often significantly reduces the amount of data that must be sent to the G-PM tool via the network, thus reducing the perturbation of the measured system and increasing scalability. In the following, we document the process of converting PMSL specifications to on-line measurements, using the metrics from Figure 2-11 as an example. In Section 2.2.3.4, we present more details on the distributed evaluation.

The way from a PMSL specification to a distributed on-line measurement consists of five phases:

1. When the user defines the new metrics, it is parsed into an intermediate representation (IR).
2. Later, when the user defines a measurement of this metrics, the IR is partially evaluated, using the now known parameter values (i.e. objects to measure, measurement restrictions).
3. The partially evaluated IR is optimised in order to reduce the measurement's perturbation.
4. The necessary inferior measurements and the monitoring requests for all probe events are defined. At the same time, a dataflow graph is created, which controls the computation of the final measurement results.
5. The dataflow graph is distributed to the components of the OCM-G monitoring system and the measurement is started.

2.2.3.3.1. Parsing PMSL into an IR

As an intermediate representation for PMSL specifications we use simple expression-DAGs (directed acyclic graphs), where inner nodes are operations, while leaf nodes are variables or constants. **Error! Reference source not found.** shows the IRs for the example metrics of Figure 2-11. Note that using an expression-DAG as IR is only possible since PMSL is a functional language. Most of the translation process is relatively straightforward, with three notable issues:

- Assignments are handled by subsequently replacing all uses of the assigned variable with the DAG of the assignment's right hand side. For indexed variables, this replacement includes an index substitution. Thus, local variables like the volume arrays in Figure 2-11 are completely eliminated.
- Variables used as iterators in aggregation functions (e.g. `p` in line 8 of Figure 2-11) are substituted by anonymous, private copies. This is necessary to avoid conflicts when the same variable is used in different aggregations.
- After creating the IR, we perform a common subexpression elimination. This avoids multiple definitions of the same inferior measurements later. In our example, the two references to the `IO_volume` metrics are unified in Figure 2-12a.

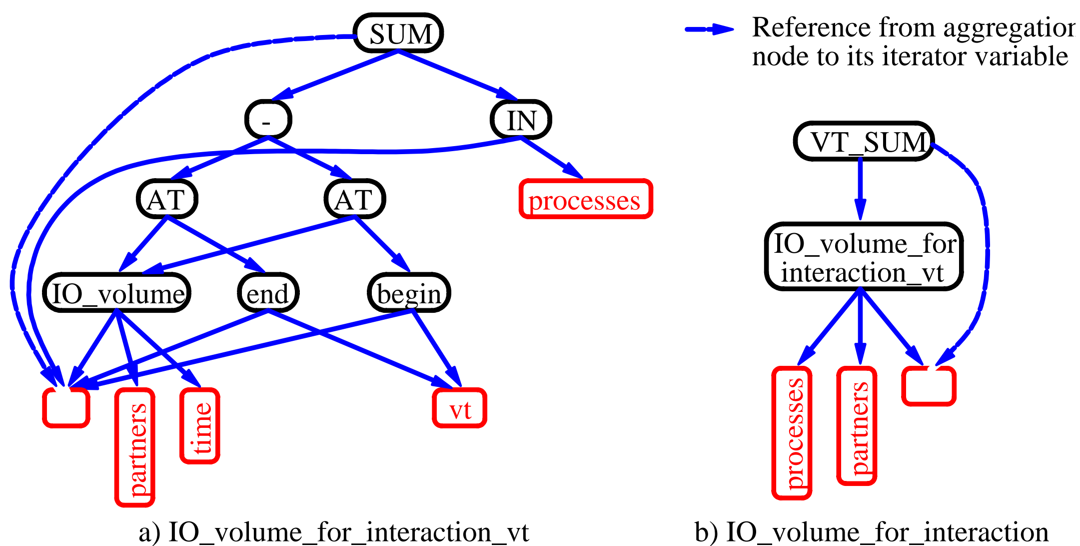


Figure 2-12: Intermediate representation of the example metrics.

The main problem at this stage, however, is the handling of the time parameters. While they are explicit input parameters in the specification, this is not the case in the implementation. An on-line performance analysis tool just requests a measurement value from time to time. Thus, the actual time of a measurement is the time when the underlying monitoring system processes this request. Fortunately, the use of time is rather limited in PMSL:

A virtual time parameter can not be modified, but just passed to inferior metrics and probes. Thus, in the implementation we can convert it from an input to an output parameter. However, we must then ensure that all "uses" of the parameter provide the same value (e.g. in line 7 of Figure 2-11 both probes must provide the same value for `vt`). This is done by taking the virtual time into account in the firing rules of the dataflow nodes created in phase 4.

Real time (and time interval) parameters can not be modified, too. Like virtual time, they can be passed to inferior metrics. This case is simply handled by requesting the values for these metrics at (approximately) the same time.

But a time interval can also be used as an operand to an `IN` operator inside an aggregation function, like in line 15 of Figure 2-11. This case is detected via pattern matching in the IR, which checks for constructs of the form

aggregation (*expr* WHERE *expr*`.time IN` *time*)

We then introduce special aggregation nodes (e.g. "VT_SUM" in Figure 2-12b) which later handle this case.

Partial Evaluation of the IR

Once the user defines a measurement based on the specified metrics, all the parameters of this metrics are fixed (with the exception of the time parameters, which are handled as described before). Thus, we can assign values to some of the leaf nodes in the DAG and evaluate its inner nodes. For each node, the evaluation either results in a constant or - if some of the node's operands still have unknown values - in a new DAG. The result of this phase is shown in Figure 2-13a, where we assumed that a measurement of `IO_volume_for_interaction` should be performed for two processes (`p1` and `p2`), restricted to consider only I/O to a file named `file1`.

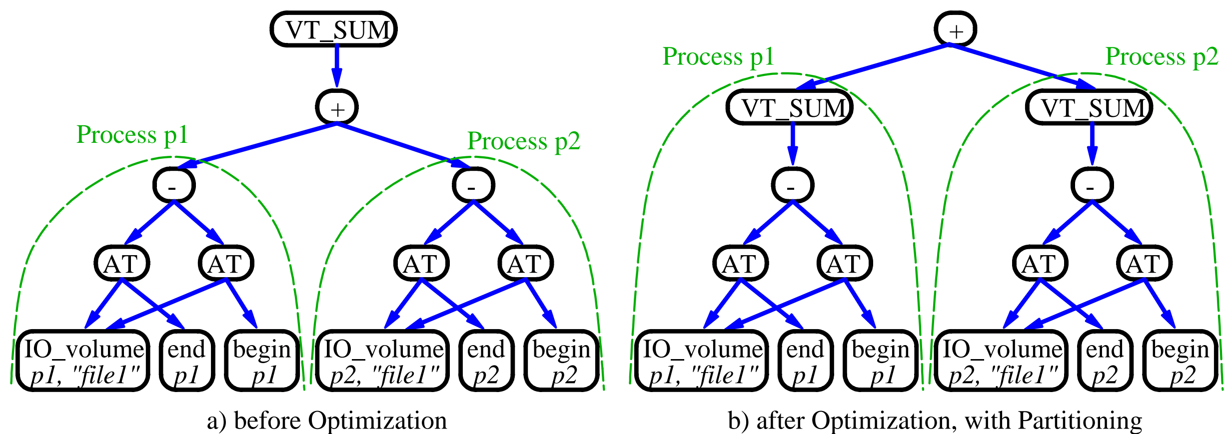


Figure 2-13: Partially evaluated intermediate representation.

The figure shows that besides the actual evaluation, a couple of other transformations is applied in this phase:

- Uses of metrics, which have been specified via PMSL, are ”inlined”, i.e. the node has been replaced with the partially evaluated DAG of this metrics. In our example, the node `IO_volume_for_interaction_vt` of Figure 2-12b has been replaced in this way.
- Since the parameters of all used metrics must evaluate to constants in this phase, there is no need anymore to represent them as explicit child nodes. Instead, the metrics nodes for built-in metrics directly contain the full measurement specification.
- Similarly, probe nodes now directly contain the information needed to request the monitoring of this probe.
- Aggregation nodes are replaced by simple operations when the iteration set is known. E.g. the “SUM” node of Figure 2-12a is replaced by a “+” node with two operands, one for each process considered.

Optimisation of the Partially Evaluated IR (PEIR)

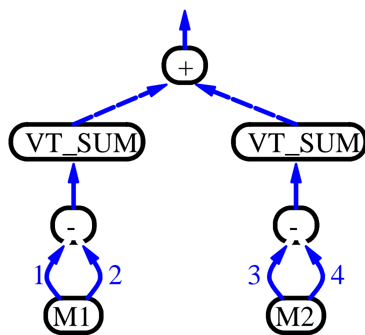
This phase is not yet implemented in the current prototype. It will be realized in the final prototype.

The idea of this phase is that the PEIR can be optimised in several ways to improve the quality of the resulting measurement. The main optimisation goal is to reduce the frequency and volume of data which needs to be sent over the network. This can be achieved by moving nodes aggregating over virtual time down towards the leaves. E.g. in Figure 2-13a, the labelled sub-DAGs can be evaluated locally in the context of the monitored processes. However, the “+” node combines data from different processes. This means that whenever an end event occurs, some communication is necessary. By interchanging the “+” and ”VT_SUM” nodes, as shown in Figure 2-13b, the aggregation over time can now also be done locally. Communication is only required when the aggregated values are needed because the result of the measurement has been requested.

Definition of Measurements and Monitoring of Probes

In this phase, a measurement is defined for each metrics node in the PEIR. Likewise, the monitoring of all probes used in the PEIR is requested. The latter results in a callback function being invoked when a probe is executed. This function usually will read some of the inferior measurements. Their results are again delivered via a callback. Since we will get these results at different times, because they are triggered by different probes, the best way to combine them into the final result is by using a dataflow approach. Thus, we also create a proper dataflow graph in this phase.

Dataflow graph:



Callback	Actions performed
p_i : begin (i = 1,2)	get vt from event notification request a result of measurement i with callback data = (vt, 2+2*(i-1))
p_i : end (i = 1,2)	get vt from event notification request a result of measurement i with callback data = (vt, 1+2*(i-1))
$measurement_i$ (i = 1,2)	get (value, time) from measurement result get (virtualTime, requesterID) from callback data pass (value,time,virtualTime,requesterID) to node M_i

data flow is triggered by execution of probes
data flow is triggered by requesting the measurement's result

$measurement_i$: measurement of IO_volume for process p_i

Figure 2-14: Implementation of the measurement.

The result for our example is shown in Figure 2-14. It shows both the dataflow graph and the activities performed by the different callbacks. The tokens flowing through the dataflow graph consist of a measurement value, its time stamp, and optionally a virtual time stamp and a requester identification. The latter is needed when the same measurement is read by different probe callbacks. In these cases, the requester identification is used to determine the dataflow arc to which the result is sent. E.g. node “M1” will forward its result token to the arc labelled, when the result was requested by the `begin` callback, and to arc labelled when requested by the `end` callback.

While in the current prototype, the whole measurement is evaluated centrally inside the HLAC, the advantage of this dataflow scheme is that the same scheme can be used in the next prototype to support distributed processing. An arc between dataflow nodes processed on different hosts just translates to a communication link.

Measurement

During the measurement, tokens are created in the dataflow graph either when a probe is executed or when the measurement result is requested. Usually, a dataflow node fires, i.e. performs its operation and produces a result token, when a token is available on each of its input arcs. In our implementation, there are two extensions to this general rule:

- If an input token contains a virtual time stamp, the operation waits until it has a complete set of operands with identical virtual time stamps (e.g. the “-” nodes in Figure 2-14).
- When a dataflow node aggregating over virtual time (e.g. the “VT_SUM” node) triggers, it does not produce a result token but rather updates an internal summary value. The node produces a result token only upon a special request. In our example, this request is issued when the result of the complete measurement is requested.

2.2.3.4. Distributed evaluation

As it shown in the sequence diagram in Figure 2-9, whenever a probe event is detected, G-PM will be informed and tries to get the result of the corresponding measurement. This will increase the frequency and volume of data which needs to be sent from the local monitors to the G-PM tool over the network during the measurement of such user defined metrics. To get rid of this deceleration, we need a solution which is based on distributed evaluation.

In addition, some constructs like the "AT" construct in PMSL require a measurement value to be determined when the occurrence of a specified event is detected. In order to avoid communication delays which would adulterate those results, a measurement value must be determined - if it is possible - on the compute nodes where the event occurs.

The Data Flow (DF) model as described above is used not only to distribute the measurement evaluation but also to reassemble the measurement results. Since these results are derived at different times (as they are triggered by different probes), the dataflow graph supports combining them into the final result. Since this model does not impose any complete ordering on the elementary function of a task, we can determine parts of independent tasks easily. In turn this will simplify the determination process for the evaluation location of sub-tasks. This model is also favoured when collecting the results, since the asynchronous nature of this model provide us support for data aggregation.

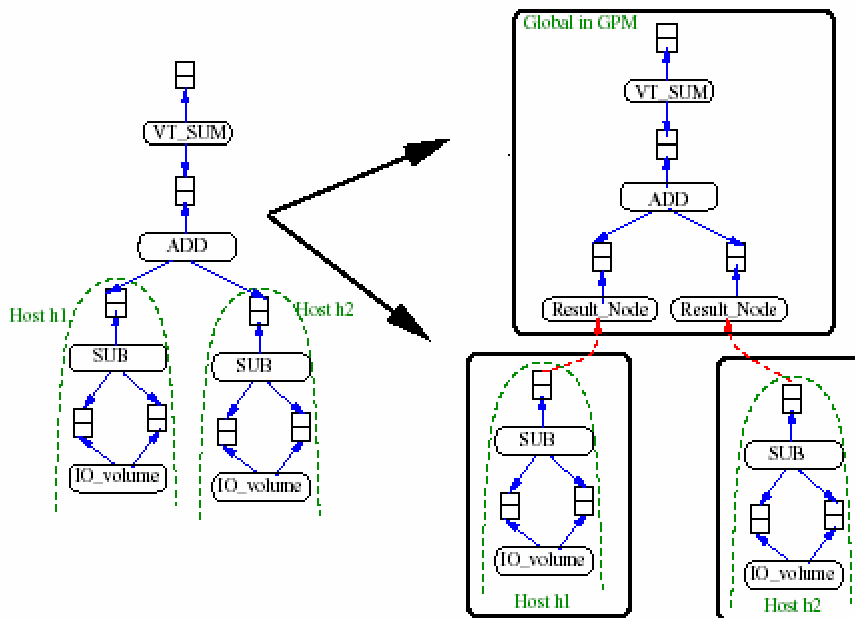


Figure 2-15: distributing a DFG into parts which can be evaluated locally.

2.2.3.4.1. Using dataflow graphs to distribute the evaluation

The DFG used in this context has two major types of nodes: dataflow nodes which aggregate or create packets of result values, and dataflow nodes which are used as First In - First Out (FIFO) as shown in

Figure 2-15.The packets contain measurement result values, which flow through the DFG as tokens consisting of measurement values, a time stamp, and optionally a virtual time stamp and a requester identification. The latter is needed when the same measurement is read by different probe callbacks. In those cases, the requester identification is used to route the packets properly. The arcs between these two types of dataflow nodes are used simply as communication link.

The dataflow nodes are created directly from the respective nodes of the DAG, except for the sub-DFGs having dataflow nodes with AT construct as roots. In such cases, the whole sub-DFG will be replaced with a single dataflow node. This dataflow node is used to collect the result values, whenever the occurrence of the underlying event is detected. For every dataflow node created in this way, a data provider node will be added at the top of it, to reassemble the result packets which are to be synchronized.

Whenever the state of a specific data provider turns from empty to non-empty, the parent dataflow node of this data provider will be notified to read the result packets from all the children nodes. This process continues until one of the children is empty. While reading the result packets, results synchronization will be performed depending on the value of the virtual time. This means that the whole result should have either the same virtual time or no virtual time at all. Otherwise, a new packet representing an error will be created and forwarded to the parent data provider.

After a successful synchronization, in general, a final result and the new time stamp will be computed using the functionality of the node and the new packet will be routed according to its requester identification of the result value. This procedure is applied only for internal dataflow nodes. Dataflow nodes, which are end-points and thus have no children data providers, forward the result packets computed at this node to the parent data provider according to the requester identification of the result values.

2.2.3.4.2. Using OMIS and OCM-G for distributed evaluation

One of the notable features of OMIS is the use of the event/action-paradigm. This feature is used here to execute actions when the occurrence of a specified event is detected. For that, we have to collect the requests that describe the event to be detected and the corresponding actions to be executed. Thus, before distributing the sub-DFGs, which can be evaluated on another component of the OCM-G, the OMIS requests for all probe events and for all data providers will be collected. When the occurrence of an event within a data provider on a remote host is detected, the corresponding parent dataflow node in GPM will be notified.

When a single probe event occurs, there could be multiple actions performed, even at different components of the OCM-G, thus the information to be gathered for every probe event should contain the access location of the actions to be performed. As a result, two types of actions are to differentiate: actions that are going to be performed at the components where the events occur and actions that can be executed only on G-PM. The latter occurs whenever we have the probe and the metrics nodes on different components of OCM-G or when the metrics node cannot be evaluated on one specified node, as it needs different processes on different components of the OCM-G. Unfortunately, it is impossible to define the probe events at this stage in the former case, since the DFG token identifying the remote host component is not known yet. Thus, we have to gather all the information and define the probe event request after the whole sub-DFG has been distributed. To define requests to such probe events, we have to separate the actions to perform into two categories: actions to be performed on the node where the occurrence of the event is detected and the actions that should be executed on another component of OCM-G.

In addition to those conditional requests stated above, there are unconditional requests, which merely execute actions without detecting the occurrence of any event. Those unconditional requests are used to create and add sub-DFGs on the appropriate host and to read active measurements that are not in G-PM. All those requests can only be defined after the distribution of sub-DFGs.

2.2.3.4.3. Sending the Sub-DFGs to their remote hosts

After the construction of the main-DFG in G-PM, the access location of the sub-DFG will be determined. The determination depends on the access location of probes and metrics. Built-in metrics, which need processes located on different nodes, can obviously be evaluated only in G-PM. A sub-DFG, which has metrics nodes and the corresponding probe nodes on a different host will also be evaluated globally.

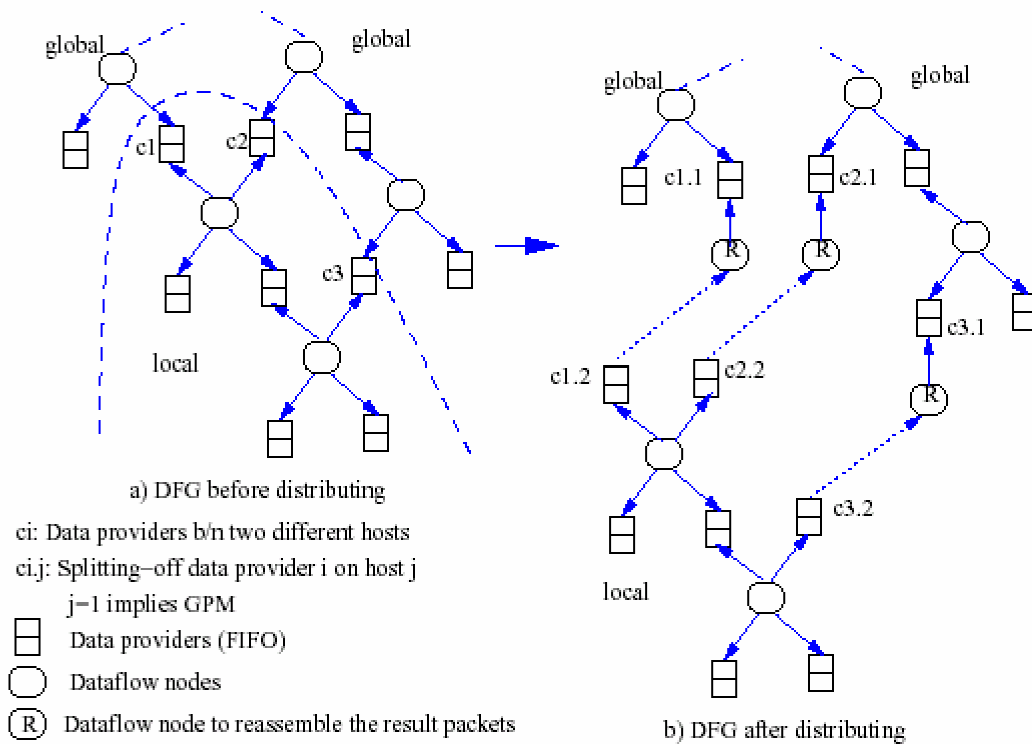


Figure 2-16: Creating a dataflow node in GPM to reassemble the result values from a remote host.

The sub-DFG that can be evaluated locally will be serialized to a string and sent to these components with the help of the following OMIS request.

```
hlac_create_dfg ([<string representing the remote host>])
```

This unconditional request service creates an empty DFG on a remote host when this remote host is requested for the first time. The parameter of this OMIS service is a list of tokens which identify the

remote hosts, where a new empty DFG will be created. This service returns a token, which will be used to add sub-DFG to the a-priori created DFG instead of creating a new one on the same remote host. To add a sub-DFG, we use the following request.

```
hlac_add_dfg (<DFG token>, <string form of the subDFG>)
```

When two sub-DFGs belonging to the same components of the OCM-G share some DFG components, these components will be identified on the remote host and will be shared there further, in order to avoid creating the same kind of DFG components on the same components of the OCM-G. After sending a sub-DFG to its appropriate host, a new dataflow node will be created to replace/represent the sub-DFG in GPM as shown in the following Figure:

This dataflow node will be notified when a result packet is detected at the corresponding data provider on the remote host. The detection of this event will be handled with the OMIS service request shown as follows:

```
hlac_dfg_fifo_read(<DFG token>, <index of the data provider>);
```

This notification is possible, since these OMIS requests are attached to the data providers before the sub-DFGs are sent to their appropriate hosts as described above . The first parameter of this request is again the DFG token, which is used to identify the remote host of the sub-DFG. These events will be triggered when the corresponding data provider with specified number (index) as the second parameter of the request receives a result measurement on the specified host

When this event is detected, the OMIS request service

```
hlac_dfg_fifo_has_value(<DFG token>, <index of the data provider>);
```

will be used to call the callback function of the dataflow node representing this sub-DFG in GPM. This data flow node in GPM will forward the result packet to its corresponding data provider. Thus, the last two OMIS service requests represent the use of event/action-paradigm of the OMIS as mentioned above.

Through distributing the sub-DFG to their appropriate hosts and connecting their data providers with the corresponding data flow nodes in GPM, we build a virtual network. With this efficient distributed implementation applications can be monitored with minimal additional overhead.

2.2.4. The class diagram of the UIVC component

The class diagram of the UIVC component is composed of the classes representing the main elements of the G-PM user interface. The public members of the classes illustrated in the diagram are in principal associated with the commands of the main menu of the application, while the members associated with the tasks of the widget library, such as drawing a window or passing a control flow to a window are not in scope of the document.

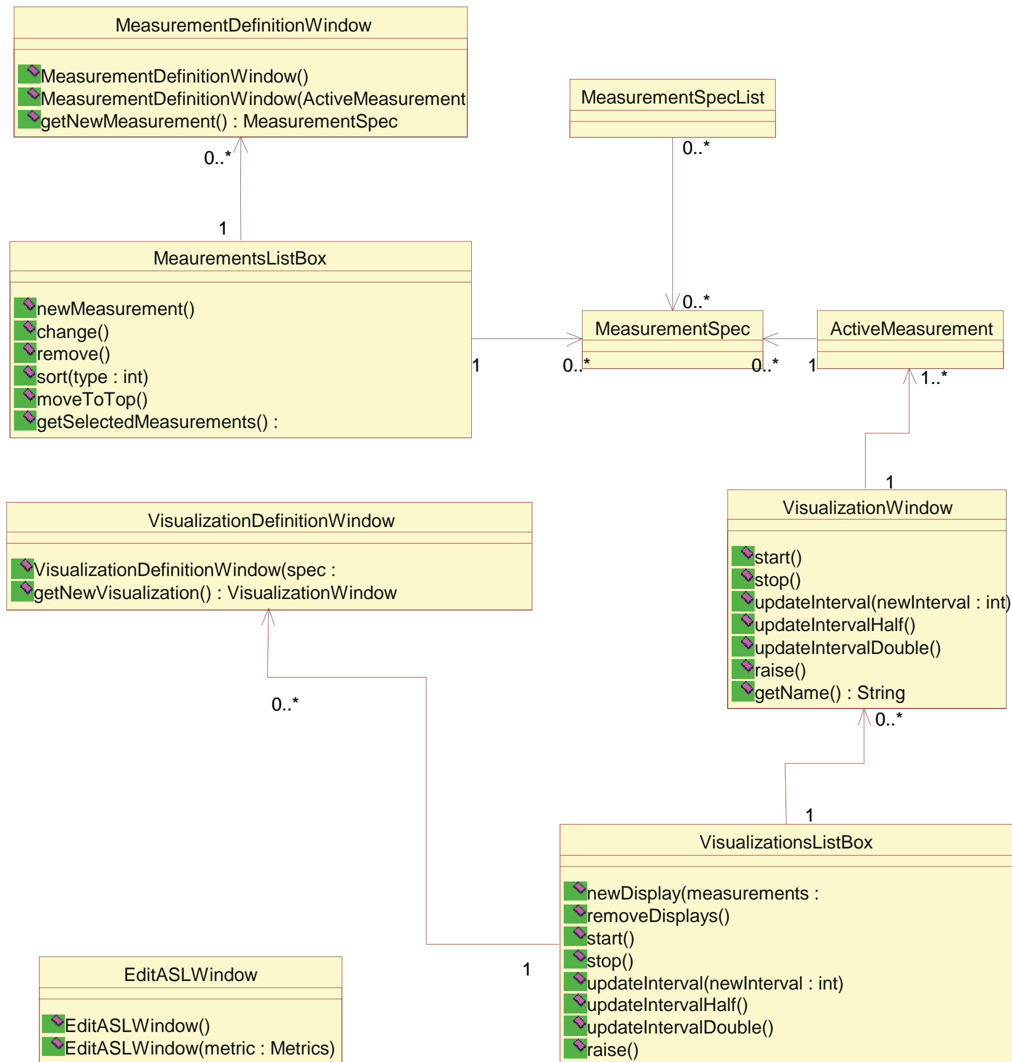


Figure 2-17: Class diagram of the User Interface and Visualization Component (UIVC)

The classes of the UIVC are illustrated in Figure 2-17. The classes *MeasurementsListBox* and *VisualizationsListBox* correspond to the elements of the main program window. These classes are responsible for handling the commands from the submenus “Measurements” and “Displays” respectively. The *MeasurementsListBox* class contains objects of the *MeasurementSpec* class as list entries, while the *VisualizationsListBox* class contains objects of the *VisualizationWindow* class as list entries.

The classes *MeasurementDefinitionWindow* and *VisualizationDefinitionWindow* are associated with the "Measurement Definition" and "Visualisation Definition" windows respectively. These classes are responsible for handling the process of defining the properties of the new measurements and displays. These properties are reflected in the variables of the *MeasurementSpec* and the *VisualizationWindow* classes respectively.

The *VisualizationWindow* class is responsible for handling the showing and manipulation of the "Visualization Window". Via the list of objects of *ActiveMeasurement* class, it also controls the running measurements associated with a display. This class is an abstract class, having a separate class inherited for each particular visualization type, such as e.g. "bar-graph".

The *EditASLWindow* class is associated with the "Edit custom metric" window. The class is responsible for enabling the editing of the custom metric text. It also handles, with a help of HLAC component classes, the process of activating and saving the edited metric.

3. PRODUCT TESTING

3.1. PMC

The PMC testing protocol was designed to evaluate the behaviour of the G-PM tool under measuring values based on different metrics. For each of the metrics a dedicated application, that shows a predetermined performance behaviour in the context of the measured metric, has been created. This applications has been used to conduct the black-box testing of the PMC subsystem.

During the black-box testing the following issues have been taken into account:

- the correlation between real (known a priori) performance characteristics of the application and the characteristics observed with the G-PM tool,
- the stability of the G-PM tool regarding interactions with the user interface and during the performance measurement process,
- the CPU load of the machine the G-PM tool is running under measurement involving processing of large amount of measured data.

Additionally during the tests, all the communication between the OCM-G and the G-PM was logged and offline analysis was done. The following issue has been taken into account:

- the use of correct OCM-G objects (e.g. counters or integrators) and commands for a specified metric,
- the interoperability between the G-PM and the OCM-G with respect of possible deadlocks, and/or inefficiency within the implemented communication protocol.

3.2. HLAC

The HLAC has been tested both via black box and grey box testing. A list of PMSL specifications has been assembled as test cases. Listing these test cases here is beyond the scope of this manual, however, the file is a part of the G-PM source code: it is located in `src/hlac/example_specs.`

During testing, this file is loaded into G-PM and measurements based on the specified metrics are defined. The tests are mainly based on debug output which is not normally switched on. In particular, when a PMSL specification is parsed, the HLAC prints the resulting intermediate representation (i.e., the DAG). When a measurement based on a user-defined metrics is defined, the HLAC prints

- the partially evaluated intermediate representation (which is a DAG, too),
- the data flow graph (DFG) created from the partially evaluated intermediate representation,
- the sub-DFGs, which result when the evaluation has been distributed.

In addition, all the requests send to the OCM-G are logged.

This information is currently inspected and verified manually. In order to support this task, a graphical presentation of the DAGs and DFGs can be created.

Due to the complexity of the logged data, a fully automatic grey box testing is not yet feasible. It is, however, possible to run automated black-box tests of G-PM (including the HLAC). This is supported by a command line version of G-PM, which reads user-defined metrics and measurement specifications from files and stores the measurement results in a file. Together with a simulation of the OCM-G monitoring system (directory `src/omis-dummy`), reproducible results can be obtained. They are compared to the expected results using a simple PERL script. This test environment is available in the `test` directory. An automatic test can be started using '`make test`'.

4. CONTACT INFORMATION AND CREDITS

The Task 2.4 Leader is:

Wlodzimierz Funika funika@uci.agh.edu.pl

The Task 2.4 team consists of:

- Roland Wismueller roland.wismueller@uni-siegen.de
- Hamza Mehammed mehammed@in.tum.de
- Tomasz Arodz tomaro@student.uci.agh.edu.pl
- Marcin Kurdziel marcink@pds.net.pl

5. GNU GENERAL PUBLIC LICENSE, VERSION 2, JUNE 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
675 Mass Ave, Cambridge, MA 02139, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, does not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this

License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to

do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>

Copyright (C) 19yy <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) 19yy name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'. This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program `Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> <signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.