Monitoring and Instrumentation Requests for Fortran, Java, C and C++ Programs

APART*Technical Report
Workpackage 1

http://www.fz-juelich.de/apart

Aurora Technical Report

Clovis Seragiotto, Jr., Hong-Linh Truong
Institute for Software Science
University of Vienna
{*clovis, truong*}*@par.univie.ac.at*

Thomas Fahringer
Institute for Computer Science
University of Innsbruck
Thomas.Fahringer@uibk.ac.at

Michael Gerndt, Tianchao Li
Institut für Informatik, LRR
Technische Universität München
{*gerndt, lit*}*@in.tum.de*

Bernd Mohr
Central Institute for Applied Mathematics
Forschungszentrum Jülich GmbH
B.Mohr@fz-juelich.de

Abstract

Most existing program performance tools for parallel systems are based on built-in instrumentation engines that are not based on a standard and therefore lack portability to different programming languages. Commonly instrumentation engines are often bound to specific compilers for specific programming languages, becoming a limiting factor at some stage of a performance tool's lifecycle, restricting its applicability and impeding further developments. As instrumentation engines are often bound to specific compilers for specific programming languages, extending them to additional programming languages comes with a high cost, which in turn prevents portability of higher-level performance analysis tools.

In previous work we proposed a standardized intermediate representation (SIR) for parallel and sequential programs, currently covering procedural and object oriented languages. A SIR decouples a performance tool from specific instrumentation engines under the assumption that the proposed SIR is supported by a large variety of instrumentation engines. In this report we introduce an XML-based language for monitoring and instrumentation requests named MIR, which in combination with the SIR defines a comprehensive, portable interface between higher-level performance analysis tools and instrumentation engines.

1 Introduction

In this document we introduce an XML-based language for monitoring and instrumentation requests named MIR and describe the format of several kinds of requests used to control the instrumentation and monitoring of an application, as well as the format of the responses expected from these requests. There are four types of requests that can be used:

- SIR: a request for the SIR of a set of programs in an application;

*The IST Working Group on *Automatic Performance Analysis: Real Tools* is funded under Contract no. IST-2000-28077

- Snapshot: a request for the current status of an application in execution;
- Instrumentation: a request for instrumenting the application;
- Control: a request for altering the instrumentation of the application or to get the value measured by the instrumentation code.

The syntactic and semantic rules of these requests and their responses are described in the following sections.

2 The SIR Request

A SIR request is used in order to obtain the SIR of one or more programs that make up an application. The SIR generated can be analyzed and its code region identifiers used to instrument the application; a SIR request also specifies where the programs must be written back after the SIR has been instrumented.

SIR requests are simple; besides the root element, *sirreq*, there may be only one other kind of element, *resource*, which names the ”files” (more generally speaking, resources) used to generate the SIR (attribute *in*) and where they should be written back after the instrumentation (attribute *out*).

The following DTD describes the syntax of a SIR request

```
<!ELEMENT sirreq (resource+)>

<!ELEMENT resource EMPTY>
<!ATTLIST resource
    in CDATA #REQUIRED
    out CDATA #IMPLIED>
```

The syntax of the *in* and *out* attributes is defined by *RFC 2396: Uniform Resource Identifiers (URI)* [3]. For instance, the following request could be used to get a program from the Web (along with a file needed to its compilation) and write it on the local disc:

```
<sirreq>
  <resource in="http://www.fictive.com/mmul.c" out="file:///home/clovis/mmul.c">
  <resource in="ftp://anonymous@fictive.com/prototypes.h">
</sirreq>
```

3 The Snapshot Request

A *snapshot request* is used to get information about some entities of an application in execution: sites, nodes, communicators (e.g. for MPI programs [5]), processes, and threads. The request itself is simple and small (it has only the root element, *snapshotreq*), while the response may contain not only the entities enumerated above, but also call stacks of the execution.

The following DTD describes the syntax of a snapshot request:

```
<!ELEMENT snapshotreq EMPTY>
<!ATTLIST snapshotreq
    named (true|false) #IMPLIED
    stack CDATA #IMPLIED>
```

The attribute *named* specifies if the snapshot must also contain the names (if available) of the entities in the snapshot, as the default is the generation of snapshots only with identifiers for these entities. The attribute *stack* specifies if the call stack of the execution is wished, and how deep it must be. The default value for *stack* is zero, that is, no call stack.

The following DTD, which describes the syntax of a snapshot, is more complex, though (the root element is *snapshot*):

```
<!ELEMENT snapshot (site*, node*, communicator*, process*, thread*)>

<!ELEMENT site (node*|(communicator*, process*, thread*))>
```

```
<!ATTLIST site
    id CDATA #REQUIRED
    name CDATA #IMPLIED>

<!ELEMENT node (communicator*, process*, thread*)>
<!ATTLIST node
    id CDATA #REQUIRED
    name CDATA #IMPLIED>

<!ELEMENT communicator (process*)>
<!ATTLIST communicator
    id CDATA #REQUIRED
    name CDATA #IMPLIED>

<!ELEMENT process (thread*|stack*)>
<!ATTLIST process
    id CDATA #REQUIRED
    name CDATA #IMPLIED>

<!ELEMENT thread (stack*)>
<!ATTLIST thread
    id CDATA #REQUIRED
    name CDATA #IMPLIED
    master (true|false) #IMPLIED>

<!ELEMENT stack (#PCDATA)>
```

Each entity in a snapshot has a unique identifier that can be used in an instrumentation request, as shown later. The names of the entities appear if available and if the snapshot request specified the attribute *named* with value *true*. The attribute *master* is used to specify whether a certain thread in the snapshot is the master thread in the process, in which case it appears with the value *true* (the default value is *false*. It makes sense only for applications with multithreaded processes (e.g. using OpenMP [2], hybrid OpenMP/MPI, multithreaded MPI). Finally, each *stack* element describes, in an application dependent format, a stack frame. The maximum number of *stack* elements in a *thread* or *process* element is limited by the value of the attribute *stack* in the snapshot request.

The following example shows a snapshot request for a Java program and the snapshot received as answer:

```
<snapshotrequest named="true" stack="3">

<snapshot>
    <thread id="15" name="AWT-EventQueue-0">
        <stack>java.lang.Object.wait</stack>
        <stack>java.awt.EventQueue.getNextEvent</stack>
        <stack>java.awt.EventDispatchThread.pumpOneEventForHierarchy</stack>
    </thread>
    <thread id="16" name="DestroyJavaVM"/>
    <thread id="1" name="main"/>
        <stack>java.lang.Thread.join</stack>
        <stack>App.main</stack>
    </thread>
</snapshot>
```

4 The Instrumentation Request

Instrumentation requests are issued *before* or *during* the program execution so as to instrument an application. They may

refer to code regions (using identifiers obtained from a SIR document) and entities like processes and threads (using identifiers obtained from a snapshot document).

The code generated through an instrumentation request is called a *probe*. A *probe* has, at every instante, a value associated to it, which corresponds either to the last value measured by the probe or to the aggregation of this value and previously measured values. This value will be called here *probe value*.

Each probe receives also a unique identifier called *probe identifier*, which can be used to retrieve the probe value, as well as to alter the probe or even remove it.

The following DTD describes the syntax of an instrumentation request. The definition of the elements *site*, *node*, *communicator*, *process*, *thread* was omitted, as it is the same as in the snapshot request (Section 3).

```
<!ELEMENT instrreq (
    codeRegion?,
    metric*,
    event*,
    measuring?,
    site*, node*, communicator*, process*, thread*)>
<!ATTLIST instrreq
    activated (true|false) #IMPLIED
    flush (true|false) #IMPLIED>

<!ELEMENT codeRegion EMPTY>
<!ATTLIST codeRegion
    from CDATA #REQUIRED
    to CDATA #IMPLIED>

<!ELEMENT metric EMPTY>
<!ATTLIST metric
    name CDATA #REQUIRED>

<!ELEMENT event EMPTY>

<!ELEMENT measuring (aggregate*)>
<!ATTLIST measuring
    delivery    CDATA #IMPLIED
    destination CDATA #IMPLIED
    interval    CDATA #IMPLIED
    duration    CDATA #IMPLIED>

<!ELEMENT aggregate EMPTY>
<!ATTLIST aggregate
    function (AVERAGE|MAXIMUM|MINIMUM|SUM|VARIANCE) #IMPLIED>
```

The root of an instrumentation request is the element *instrreq*. Inside it the following elements are allowed:

- The *code region* element, with the attributes *from* and *to*. These attributes contain the identifier of a *unit* or *codeRegion* element in a SIR, and delimit the beginning and end of a region to be monitored in the input program. Some metric will be measured at the beginning and at the end of the region, and the *probe value* will be the difference between these two values. If the *to* or the *from* element is omitted, then the metric will be measured only at the beginning or at end of the code region (no difference will be computed).

  The concept of ”valid” region, however, is application dependent - one instrumentation tool may allow to define a region that begins in a function and ends in other, while another tool not.

- The *metric* element, defining which metric should be measured for the region of interest. Several *metric* elements may be present in a single instrumentation request. Possible values for metrics depend on specific implementations (see Section 8).

- The *event* element, indicating that event traces must be generated for the region of interest. The format of the event traces remains to be defined and is not covered in this specification.
- The *measuring* element, which defines:
  - how often, in milliseconds, measurements must be done (attribute *interval*). The default value is zero, which means that the measurements are done only when the code region defined with the *codeRegion* element is executed.
  - how often, in milliseconds, values measured are automatically delivered (attribute *delivery*). The default value is zero, which means that the values are not automatically delivered; they must be retrieved through a *control request* (see Section 5). The value -1 has a special meaning: the values are delivered every time they are measured.
  - where values measured must be delivered (attribute *destination*). Values are always delivered as *measurement* documents (see Section 6), but the default value for the *destination* attribute is application dependent. The format, though also application dependent, must follow the URI syntax [3].
  - how long, in milliseconds, a measurement takes (attribute *duration*). The default is zero; a non-zero value T means that the measurement must be done at instant K, then at instant K + T, and that the difference between the two values measured must become the probe value.

  A measuring element may contain *aggregate* elements, to indicate that measurements must be grouped according to certain statistic functions (*AVERAGE*, *MINIMUM*, *MAXIMUM*, *SUM*, and *VARIANCE*). If no *aggregate* element is specified, or if it is specified with no function, then the probe value will always be the last value measured; otherwise, it will be the value returned by one of these functions (or values, if more than one function is specified).

  A *measuring* element without attributes and without an *aggregate* element is invalid, while an instrumentation request containing only the *measuring* element defines new default values for interval, duration, delivery, destination, and aggregate. To such a request no probe identifier is assigned.
- The *thread*, *process*, *communicator*, *node*, and *site* elements, specifying the identifiers of threads, processes, communicators, nodes, and sites for which measurements must be taken. The format of an identifier for any of these elements must be obtained from a snapshot document (see Section 3). Two symbols, however, have a special meaning for an identifier: the asterisk, which means ”all”, and the question mark, which means ”the current entity” (or ”the entity doing the measurement”). For instance, `<process id="*">` means that the measurement must be taken for all processes related to the application, while `<process id="?">` means that the metric must be measured only in one process, namely the one that is doing the measurement itself. Question marks are useful, for instance, when taking measurements for a code region. In fact, if an instrumentation request specifies a code region but no entity, the question mark is assumed as ”identifier” for the elements *thread*, *process*, *communicator*, *node*, and *site*.

  Note that there is a difference between a request that specifies
  ```
  <process id="P1"/>
  ```
  and one that specifies
  ```
  <process id="P1">
    <thread="*">
  </process>.
  ```
  The first request asks for one single value, measured for the process *P1*, while the second request asks for several values, one for each thread of process *P1*.

An instrumentation request may also have two attributes: *flush* and *activate*. When the *flush* attribute has the value *true*, the current instrumentation request is flushed, together with all the previous instrumentation requests where the attribute *flush* was *false* or absent. One particular consequence is that only now SIRs may be parsed back to source code (now also with instrumentation code). The attribute *activate*, if specified with the value *true*, indicates that the measurements must start as soon as the instrumentation is flushed. When the value is *false* (or the attribute is absent), the probe starts inactive, and a control request (see Section 5) will be needed to activate it.

The response to an instrumentation request is a *probe* document, the syntax of which is defined as follows:

```
<!ELEMENT probe EMPTY>
<!ATTLIST probe
    id CDATA #REQUIRED>
```

The attribute *id* of a probe is the identifier of the instrumentation request; it may be used later in a control request and also to identify a value in a measure document (see Section 6).

The following example shows how to measure the number of times the threads *1045* and *1032* blocked to enter in a critical section. This value is measured every time any thread executes the code region with identifier *c1*, and the maximum of the values measured is sent every second to the file "/tmp/foo.txt".

```
<instrreq>
  <codeRegion from="c1"/>
  <metric name="BLOCKED_COUNT"/>
  <measuring
    delivery="1000"
    destination="file:///tmp/foo.txt"
      <aggregate function="MAXIMUM"/>
  </measuring>
  <thread="1045"/><thread="1032"/>
</instrreq>
```

If we used *<thread id="?">* instead, the metric BLOCKED_COUNT would be measured for each thread, but the measurements for any thread T would be taken only when T executed the code region *c1*. By using *<thread id="*">*, however, the measurements would be taken for all threads every time *any* of them executed the code region *c1*.

5 The Control Request

A *control request* is used to access the probe created through an *instrumentation request*. With control requests and the probe identifiers returned by the *instrumentation requests* (see Section 4) it is possible to change the instrumentation or retrieve the values it measures.

The root of a control request is the *ctrlreq* element. The DTD giving the syntax of control requests is given below:

```
<!ELEMENT ctrlreq (
    probe+,
    metric*,
    measuring?,
    site*, node*, communicator*, process*, thread*)>
<!ATTLIST ctrlreq
    flush (true|false) #IMPLIED
    action (VALUE|ACTIVATE|DEACTIVATE|RESET|REMOVE) #REQUIRED>
```

- The *probe* element specifies, with the attribute *id*, the identifier returned by a previous instrumentation request. Several *probe* elements may be specified in the same request.
- The *action* attribute defines the effect of this control request on the probe(s). Possible actions are:
  - *VALUE*: The last value measured (or the last aggregation) is returned as a *measurement* document (see Section 6); the instrumentation does not change.
  - *ACTIVATE*: The measurements start to be taken for the specified probe(s). If they already were active, nothing happens.
  - *DEACTIVATE*: The measurements stop to be taken for the specified probe(s). The perturbation generated by the probe(s) should be minimal.
  - *RESET*: Resets the aggregations associated with the specified probe(s). All the measurements taken for that probe(s) until the moment of this request will be forgotten, as if the probe had just been inserted.
  - *REMOVE*: Invalidates the specified probe, possibly removing the instrumentation.
- The attribute *flush*, as well as the elements *metric*, *measuring*, *site*, *node*, *communicator*, *process*, and *thread* are equivalent to their counterparts in a instrumentation request. If left unspecified, the previous settings associated with the specified probe(s) remain unaltered.

The following example removes the probes *p1* and *p2*:

```
<ctrlreq>
   <probe id="p1"/> <probe id="p2"/> <action type="REMOVE"/>
</ctrlreq>
```

This example returns the last value measured for probe *p2*:

```
<ctrlreq>
    <probe id="p2"/><action type="VALUE">
</ctrlreq>
```

Finally this example changes the probe *p3* to measure every 4 seconds the time that communicator *cb* spent sending messages:

```
<ctrlreq>
    <probe id="p3"/>
    <action type="RESET"/>
    <metric name="NET_SEND"/> <!-- NET_SEND means time spent sending messages -->
    <measuring interval="4000"/>
    <communicator id="cb"/>
</ctrlreq>
```

6 The Measurement Document

The response to a control request whose action has the type *VALUE*, as well as the document sent automatically if the probe is associated to a delivery interval different from zero, is a *measurement* document, the syntax of which is defined as follows:

```
<!ELEMENT measurement (measurement)*>
<!ATTLIST measurement
    probeId        CDATA #IMPLIED
    siteId         CDATA #IMPLIED
    nodeId         CDATA #IMPLIED
    communicatorId CDATA #IMPLIED
    processId      CDATA #IMPLIED
    threadId       CDATA #IMPLIED
    value          CDATA #IMPLIED>
```

A *measurement* document is generated from a set of tuples (probeId, siteId, nodeId, communicatorId, processId, threadId, metric, value), where *null* is also a possible value for siteId, nodeId, communicatorId, processId, and threadId (in a pure sequential program, for instance, all of them can be *null*). Each tuple corresponds either to the value measured for some metric or to an aggregation of values (average, maximum, minimum, sum, variance); for the aggregations average, sum, and variance, siteId, nodeId, communicatorId, processId and threadId will always be null.

In order to generate the document in a compact form, the following algorithm is applied (where we call *last defining request* the instrumentation request that created a probe or the control request that last modified it):

1. *Generation:* For each tuple, a *measurement* element is generated using the values in the tuple as the values of the respective attributes in the element (note that there is no attribute for *metric*).
2. *Sorting:*
    - If two *measurement* elements m_1 and m_2 have the same value for the attribute *probeId* but different values for the attribute *siteId*, then m_1 must appear in the document before m_2 if the site in m_1 was neither an asterisk nor a question mark and it has been specified before the site in m_2 in the last defining request of the corresponding probe. A similar rule is applied if two *measurement* elements have:

  - the same value for the attributes *probeId* and *siteId* but different values for the attribute *nodeId*;
  - the same value for the attributes *probeId*, *siteId*, and *nodeId* but different values for the attribute *communicatorId*;
  - the same value for the attributes *probeId*, *siteId*, *nodeId*, and *communicatorId* but different values for the attribute *processId*;
  - the same value for the attributes *probeId*, *siteId*, *nodeId*, *communicatorId*, and *processId* but different values for the attribute *threadId*.
- If two *measurement* elements m_1, generated from tuple t_1, and m_2, generated from tuple t_2, have the same values for the attributes *probeId*, *siteId*, *nodeId*, *communicatorId*, *processId*, and *threadId*, then m_1 must appear in the document before m_2 if the metric in t_1 was specified before the metric in t_2 in the last defining request of the corresponding probe. This rule guarantees that the metric a *measurement element* refers to can always be inferred from the last defining request.

3. *Compression:*

- Remove what can be inferred from the last defining request:
  - The attribute *probeId* is removed from all *measurement* elements if the document contains values for only one probe.
  - An attribute *siteId*, *nodeId*, *communicatorId*, *processId* and *threadId* is removed if its value is *null* or if the last defining request for the probe did not specify an aggregation and used neither an asterisk nor a question mark as identifier of the corresponding site, node, communicator, process or thread.
- Merge elements that have attributes in common:
  For each attribute k in the list (probeId, siteId, nodeId, communicatorId, processId, threadId) in this order, if n *measurement* elements ($n > 1$) have the same value for the attribute k, then they will be replaced by a single *measurement* element with only the attribute k and its value and, nested within this new element, the previous n *measurement* elements without the attribute k.
- If there is still more than one *measurement* element, a new "root" *measurement* element, without any attribute, is generated to nest the other *measurement* elements.

The following example shows an instrumentation request that measures the number of threads for the communicators *c1* (containing the processes *p1* and *p2*) and *c2* (containing the process *p3*), as well as a possible *measurement* document generated for the values measured (comments between <!– and –> are not generated):

```
<instrreq>
    <metric name="THREAD_COUNT"/>
    <communicator id="c1"><process id="*"/></communicator>
    <communicator id="c2"><process id="*"/></communicator>
</instrreq>

<measurement>
    <measurement>      <!-- communicator c1 -->
        <measurement processId="p2" value="3"/>
        <measurement processId="p1" value="4"/>
    </measurement>
    <measurement>      <!-- communicator c2 -->
        <measurement processId="p3" value="5"/>
    </measurement>
</measurement>
```

If the instrumentation request had specified the *aggregate* element with the attribute *function = MAXIMUM*, the *measurement* document would be simply:

```
<measurement communicatorId="c2" processId="p3" value="5"/>
```

7 Errors

Responses to requests may also return *errors* instead of a normal answer according to the following syntax:

```
<!ELEMENT errors (error)+>
<!ELEMENT error (#PCDATA)>
```

where each *error* element contains an application-dependent error message. For example,

```
<errors> <error>File not found: mm.c</error> </errors>
```

8 Metrics

Each metric has a unique name. The attribute *name* in the element *metric* specifies the unique name of the metric. Metric can be timing (e.g. wallclock time), counter (e.g. number of function calls), and hardware counter (e.g. L2 cache misses provided by PAPI [1]). Currently, the name and the number of metrics supported are dependent on specific implementations of the instrumentation and monitoring tool. Each implementation should provide a *metric catalog* that documents its supported metrics. As a performance tool may work with different instrumentation and monitoring tools, a metric may need to be associated with a name space.

Table 1 displays an example of metrics that can be supported by a monitoring tool for Java programs (time is always given in milliseconds).

Metric Name	**Description**
WC_TIME	Wallclock time.
CPU_TIME	CPU time.
LOADED_CL_TOTAL	Number of classes loaded since the application started.
LOADED_CL_CURR	Number of classes currently loaded.
UNLOADED_CL	Number of classes that have been unloaded.
COMP_TIME	Time spent with just-in-time compilation.
GC_COUNT	The total number of garbage collections that have occurred.
GC_TIME	The accumulated garbage collection time in milliseconds.
HEAP_MEM_USAGE	Amount of used heap memory, in bytes.
NON_HEAP_MEM_USAGE	Amount of used non-heap memory, in bytes.
NON_FINALIZED_OBJECTS	Number of objects for which finalization is pending.
THREAD_COUNT	The current number of live threads.
DAEMON_THREAD_COUNT	The current number of live daemon threads.
THREAD_WAITED_COUNT	Total number of times a thread has waited for notification (that is, number of invocations to the method wait).
THREAD_WAITED_TIME	The accumulated time a thread has waited for notification.
BLOCKED_COUNT	Total number of times a thread blocked to enter or reenter a critical section.
BLOCKED_TIME	The accumulated time a thread has blocked to enter or reenter a critical section.
NET_SEND	Time spent sending messages through the network.
NET_BYTES_SEND	Number of bytes sent through the network.
NET_BLOCKED_SEND	Time spent waiting to send the first byte when writing to the network.
NET_RECV	Time spent receiving messages without blocking.
NET_BYTES_RECV	Number of bytes received through the network.
NET_BLOCKED_RECV	Time spent waiting for the first byte when reading from the network.
NET_INIT	Time spent initializing connections.
NET_END	Time spent finalizing connections.

Table 1. Example of metrics measured for Java programs.

9 DTD to XML Schema Translation

Generic rules for translating DTDs to XML schemas can be found in [4]; in addition, the type restrictions shown in Table 2 should also be used.

Element	Attribute	Type in the XML Schema
thread	omp-master	boolean
resource	in	anyURI
	out	anyURI
snapshot	named	boolean
measuring	delivery	nonNegativeInteger
	destination	anyURI
	interval	nonNegativeInteger
	duration	nonNegativeInteger
measurement	value	decimal
instrreq, ctrlreq	flush	boolean

Table 2. Types to be used when converting to XML Schema the elements and attributes of the DTD describing the MIR grammar

.

10 Conclusion

This document has defined a set of requests and responses to be used in the communication with instrumentation engines. The use of these requests can greatly reduce the dependency of a performance tool on a specific instrumentation engine without restricting the capabilities of neither of them.

It must be noted that an instrumentation engine does not need to fully support all of the possible requests in this document to be ”MIR compatible”. For instance, it may be impossible to remove instrumentation that has been statically inserted; therefore, a tool for static instrumentation may still support the MIR format, just without supporting all its features.

As XML documents may be lengthy, we tried to reduce the average size of requests and responses, although it has always been clear that they will never be so small as a binary format tuned for a specific application, language, and platform. This overhead can however be not so important when compared to the portability achieved through the use of MIR.

References

[1] PAPI - Performance Application Programming Interface. http://icl.cs.utk.edu/papi/.

[2] OpenMP Fortran Application Program Interface Version 2.0. http://www.openmp.org/specs/mp-documents/fspec20.pdf.

[3] RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax. http://www.ietf.org/rfc/rfc2396.txt.

[4] A Conversion Tool from DTD to XML Schema. http://www.w3.org/2000/04/schema_hack.

[5] W. Gropp, E. Lusk, N. Doss, A. Skjellum, A high-performance, portable implementation of the MPI message passing interface standard, Parallel Computing, 22(6):789-828, Sept 1996