SPASS-meter Manual

version 1.22

University of Hildesheim
Software Systems Engineering
31141 Hildesheim, Germany

Contents

1. Introduction

SPASS[1]-monitor is a resource monitoring framework which is primarily intended for Java programs, but, due to its architecture, will be applicable to other program types as well. It is intended to be small in footprint, simple in architecture, flexible in configuration.

The core concept of SPASS-monitor is the **monitoring group**, a logical grouping of program units such as classes or methods being treated as one unit when accounting resource consumption (memory usage, process time consumption, CPU load, file I/O, network I/O). All consumed resources the members of a monitoring group are accounted for that group. In the simplest case, a monitoring group consists of exactly one element (class or method) and represents the resource allocation of that element. In more complex cases, e.g., a component interface which is specified by several interfaces and realized by multiple classes, a monitoring group may consist of multiple elements.

A monitoring group is user defined and specifies the **monitoring semantics** for its members, i.e., which resources to monitor and whether the resources shall be monitored directly (just for the members themselves) or indirectly (also for dependent classes). This enables the user to focus on relevant information and to gain control over the monitoring overhead, i.e., the additional resource usage incurred by the monitoring activities. The **monitoring scope**, consisting of the monitoring group definitions and their individual monitoring semantics can be configured as part of the source code or in an external configuration file. Furthermore, SPASS-monitor provides **different views** on the accounted monitoring group, i.e. the absolute values accounted for the monitoring group as well as related relative values (fractions, percentages) calculated with respect to the entire program, the virtual machine (based on internal resource consumption of the machine as well as external statistics provided by the operating system for the process) or the entire device.

This document describes the application of SPASS-monitor, particularly its installation, its configuration, additional monitoring modes and its extensions such as the integration into Java Management Extensions (JMX) or the use with Android Apps.

2. Terminology

We will call the program observed by the monitoring framework the **system under monitoring** (SUM). A **source code annotation** is syntactical metadata placed in the source code, typically without influence on the behavior of a program. A Java **instrumentation agent** is a program called by the virtual machine for each individual class loaded into the virtual machine in order to enable possible modifications of the byte code of the class (**instrumentation**). **Dynamic instrumentation** is performed at load time of classes and may be present at startup time of the JVM or started later. Dynamic instrumentation may even happen after load time by modifying a given class (structural restrictions apply as the known interfaces must not change) and replacing it dynamically. **Static instrumentation** is done as part of the development process after compiling the classes and before starting the instrumented program.

3. Monitoring Process

The monitoring process of SPASS-meter is illustrated in Figure 1. Basis for monitoring is the **monitoring scope specification**, i.e., the monitoring groups and their individual monitoring semantics. This configuration is provided by the performance engineer and may differ for the SUM according to varying interests. Then, SPASS-meter **instruments** the SUM according to the monitoring scope specification provided as input. Instrumentation may happen before runtime (static), during runtime (dynamic) or even as a combination of both (mixed). During runtime of the SUM, the **data collection** layer of the SPASS-meter framework receives the observed information, pre-aggregates it due to performance reasons and passes it to the aggregation layer. The aggregation layer maintains

[1] SPASS = Simplifying the development of adaptive software systems, means also "fun" in German

the monitoring groups and performs various detailed aggregation operations depending on the detailed configuration (either given in terms of the monitoring scope specification or the global settings of the SPASS-meter framework provided in terms of command line options). Finally, the aggregated information may be processed online during the runtime of the SUM, e.g., for realizing adaptive systems, or post-mortem as offline analysis.

4. Installation

SPASS-monitor is provided in several main JAR files due to technical reasons. In this Section, we will describe the individual parts in terms of their use during the software lifecycle.
SPASS-meter is prepackaged for three specific purposes. Each SPASS-meter distribution package contains specific libraries such as for Windows (XP and later including 64bit), Linux (32 and 64bit) or Android. Please note that for Linux at least GLIBC 2.7 is required.
Currently, we provide three prepackaged distributions, one plain distribution which provides a post-mortem summary (Windows and Linux), a JMX enabled distribution with specific extensions for the JMX console (Windows and Linux) and one distribution for static instrumentation for Android Apps (no dynamic instrumentation and no JMX functionality as this is currently not supported by Android). Basically, each distribution package is a ZIP archive and can simply be extracted and used, i.e., no specific installation is needed but some considerations as described regarding the specific application must be made.

4.1. Development time

If source code annotations shall be used to configure the monitoring scope, the `SPASS-meter-annotations.jar` file must be included into the classpath of the SUM. At runtime, the contained classes are provided by the library classes described below. Further, `SPASS-meter-ant.jar` realizes the integration with Apache ANT, which is in particular useful for static instrumentation.

4.2. Dynamic instrumentation

For runtime instrumentation, the following files are required.

- `SPASS-meter-ia.jar`: Contains the instrumentation agent. The instrumentation agent dynamically loads the other JAR files described below. Therefore, they must be located in the same directory as the `SPASS-meter-ia.jar` file.
- `SPASS-meter-boot.jar`: Contains internal interfaces as well as annotation definitions and types needed by the instrumentation agent to communicate with of the collection layer.
- `SPASS-meter-rt.jar`: The runtime library of SPASS-meter includes the remaining parts of the monitoring framework as well as the native library for accessing operating system information sources.

4.3. Static instrumentation

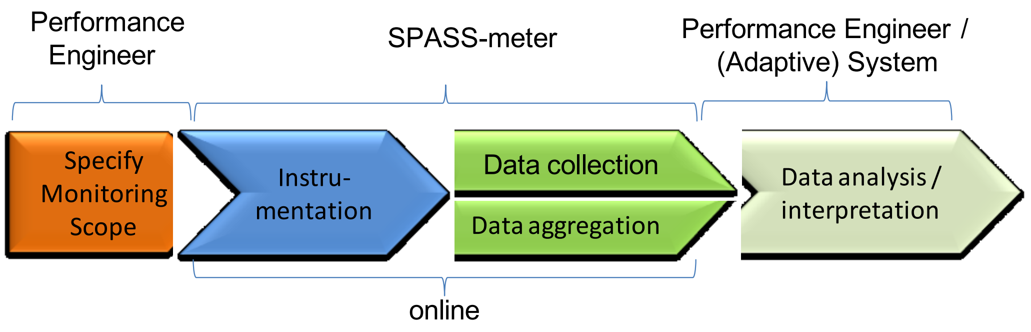


Figure 1: Spass-meter monitoring process

For static instrumentation, basically `SPASS-meter-static.jar` provides the plain tooling part while `SPASS-meter-ant.jar` contains an integration of the tooling with ANT. For static instrumentation, `SPASS-meter-static.jar` must be part of the SUM classpath.
Please ensure that the JAR files are readable, i.e., file and folder permissions are set accordingly, in particular if JVMs shall be monitored that are spawned in the context of different users.

5. Configuration

Before applying SPASS-monitor to a SUM, the framework needs to be configured, in particular to define the monitoring scope. Furthermore, the user may specify general options, e.g., where to store the post-mortem summary. General options can be specified using the command line of the SPASS-meter instrumentation agent (Section 5.1). The SUM specific configuration can be given in terms of source code annotations (Section 5.2) or as external configuration in an XML file (Section 5.3). In an XML file, also some general options specific regarding the measurement of the configured SUM can be stated.

5.1. General configuration

Let us assume that `program.jar` contains the SUM which is started via the class `Main`. Then the following command[2]

```
java
  -javaagent:spass-meter-ia.jar=out=program.log
  -classpath .;program.jar
  Main
```

activates the SPASS-meter instrumentation agent, starts the program, instruments the classes loaded by the program and collects the results in program.log. The option `out` is a general option which is directly presented to the instrumenter (the = character before the option is the parameter separator character defined by Java). In this case we assume that monitoring groups and further details are given as source code annotations, i.e., no external XML configuration file is needed. SPASS-meter offers the following global options. Default values are underlined. Multiple configuration options can be appended and are separated by a comma.
Below we list the basic set of configuration options. Please refer to Section 11 for advanced configuration options.

name	description and possible values	type and default
`xmlconfig`	Read the monitoring configuration from the given XML file (c.f. Section 5.3).	File location
`out`	Write the monitoring results as a summary to the given file using a default textual data schema. If a directory is given, SPASS-meter will create log files with names according to the JVM identifier (minimal support for distributed monitoring).	File location
`printStatistics`	Printout statistics at the end of monitoring.	<u>`TRUE`</u>, `FALSE`

[2] Versions prior to 1.1 required that spass-meter-rt.jar was listed in the classpath.

`tcp`	Process instrumentation and resource observation in this JVM and send the monitored results to the specified remote monitoring server. For details see Section 6.	structured string
`instrumentJavaLib`	If enabled, the Java library is instrumented in order to gain information (default). If disabled, only the program is instrumented with impact on the collected data, e.g. memory cannot be accounted in all cases (needed for static instrumentation).	`TRUE, FALSE`
`groupAccounting`	Defines the default strategy for group accounting.	o `DIRECT` account the probed data for the immediate group on the thread call stack o `INDIRECT` account the probed data for all (nested) groups on the thread call stack o `LOCAL` on global configuration level, overrides all local settings, ignored else
`accountableResources`	Defines the default accountable resources.	`CPU_TIME`, `FILE_IO`, `MEMORY`, `NET_IO`, default is empty, i.e., all
`outInterval`	Defines the output interval for runtime refresh of monitoring groups. If less than 500 the value is interpreted as 500ms units, if less or equal to 0 this option will be disabled.	integer, default is 0 (disabled)
`mainDefault`	The default instrumentation behavior in case that explicit start and end configuration is missing. Applies to the first main method found by the instrumenter, i.e. the program start.	`NONE, START, END, SHUTDOWN, START_END, START_SHUTDOWN`
`registerThreads`	Do threads need explicit registration with their native counterparts, only needed in case of JVMs without JMX support, e.g., on Android.	`TRUE, FALSE`
`exclude`	Classes to be excluded from instrumentation given as a comma separated list in terms of their (prefixed) JVM names (packages separated by /). This may affect measurements on SUM level, e.g., thread information may be missing!	empty
`allClassMembers`	Consider all class members for	`true`

	instrumentation or apply configuration-dependent filters such as "plain time" (this may affect measurements on SUM level), e.g., thread information may be missing!	
`instanceIdentifierKind`	How shall consumptions of individual instances within a thread be recorded (not intended for frequent object creation).	`NONE`, `THREAD_ID`, `IDENTITY_HASHCODE`

Further options provided by SPASS-meter are either deprecated or experimental and, thus, not listed here.

5.2. Annotation-based Configuration

In this section we describe the annotations for configuring the monitoring operations in source code. Annotations can be applied in case of accessible source code and particularly support development activities as they are considered by refactoring operations. Annotation-based configuration is of particular interest for systems which are intended to be bundled with SPASS-meter.

Note: Each monitoring configuration needs explicit information where to start and where to stop monitoring. This information is particularly necessary in case when initialization actions should not be monitored or the end of monitoring is not the same as the start as it is usual in event based systems such as applications with user interface. The main method may be flagged with both, the start and the end annotation. For annotation-based configuration you need to include `spass-meter-annotations.jar` into SUM classpath.

Annotation	Description	Applicable to	Parameters
`Monitor`	Enables monitoring for a specific program element and assigns it to the specified monitoring group.	Class, Method	• `id`: the symbolic name (s) of the monitoring group; creates a multi-group if used multiple times • `debug`: enable specific debugging output, default off (combination of `CONFIGURATION`, `MEMORY_FREE`, `METHOD_ENTER`, `METHOD_EXIT`, `MEMORY_ALLOCATION`, `NET_IN`, `NET_OUT`, `FILE_IN`, `FILE_OUT`) • `groupAccounting`: specific accounting for this monitoring group, may be `LOCAL`, `DIRECT`, `INDIRECT` or `DEFAULT` (default, take the value from the global configuration) • `resources`: specific resources to be accounted, may be `CPU_TIME`, `MEMORY`, `FILE_IO`, `NET_IO` (default see `defaultGroupResources` in global Configuration) • `considerContained`: Should accountable resources of related groups be considered: `TRUE`, `FALSE`, `DEFAULT` (see global configuration `multiConsiderContained`)

			• `distributeValues`: Should resource consumption be evenly distributed: `TRUE`, `FALSE`, `DEFAULT` (see `multiConsiderContained` in global configuration) • `instanceIdentifierKind`: How shall consumptions of individual instances within a thread be recorder (not intended for frequent object creation): `NONE`, `THREAD_ID`, `INDENTITY_HASHCODE`
`ExcludeFromMonitoring`	Exclude the related element from monitoring. Containing class should be flagged with `Monitor`.	Class, Method	
`StartSystem`	Mark this method as the start of the system, i.e. where to start recording.	Method	• `shutdownHook`: if `TRUE`, the code for ending the monitoring is attached to the JVM as a shutdown hook (`FALSE`) • `invoke`: (Test-) Method to be invoked at the end of shutdown hook (default is empty)
`EndSystem`	Mark this method as end of the system, i.e. where to stop rercording.	Method	• `invoke`: (Test-) Method to be invoked at the end of monitoring (default is empty)
`Timer`	Obtain response time of one or several methods. Not recorded in summary. Sends an event upon completion.	Method	• `id`: the unique identification of the timer • `considerThreads`: add the thread identification to the id. • `affectAt`: where to add the code to the method (`DEFAULT`, `BEGINNING`, `END`, `BOTH`) • `state`: the next state of the timer (`START`, `SUSPEND`, `RESUME`, `FINISH`, …)
`ValueChange`	Notify about value changes of attributes. Not recorded in summary.	Attribute	• `id`: the unique identification of the attribute / event for notification (not recorded in post-mortem summary)
`ValueContext`	Define the (reusable) id of a value change for an instance of an object holding	Attribute of object variable	• `id`: the unique identification of the attribute / event

	attributes marked with a value change annotation and id "*" whereby at a concrete change "*" is replaced by the surrounding context id.		
`NotifyValue`	Context specific correction of automatically determined resource values, e.g. in order to properly reflect commonly used or shared resources. The value determined by expression is added to the current value of the respective monitoring group.	Method	• `id`: the unique identification of the monitoring group to be informed (optional, if not given the current monitoring group is used) • `value`: the kind of value to be modified (ALL, FILE_IN, FILE_OUT, NET_IN, NET_OUT, VALUE) • `notifyDifference`: notify the absolute value (`true`) or the difference between start and end of method (`false`) • `tagExpression`: java expression relying on parameters of the method or attributes in order to determine the correcting value (default is empty)

Further details regarding the options of the individual annotations can be found in the JavaDoc source code documentation of the annotations. Some hints are stated below

- The typical annotation is `Monitor` in combination with global settings for resources. Although, there are plenty combinations possible, the configuration is typically rather simple (see example below).
- The annotation `Monitor` takes precedence over `ExcludeFromMonitoring`. Monitor is not automatically applied to super- or subclasses – instead each individual class needs to be flagged appropriately (see `annotationSearch` in the global configuration). For inner classes a lookup for the annotations of the (outer) declaring class is performed by default. Here, `ExcludeFromMonitoring` also makes sense for classes.
- Defining additional information to the `Monitor` annotation may imply some drawbacks. In case that `id` is used only once, the configuration is consistent as all information is available when internally registering the group for monitoring. In case of multiple classes or methods which form a monitoring group, each annotation may define its own set of additional information but only the annotation which is loaded first counts. In that case, we (currently) suggest additionally using the XML-based configuration and specifying the additional information per `id` once in a consistent form.
- Denoting the end of the SUM is important to properly terminate the monitoring activities and, thus, the SUM itself. Dependent on the actual configuration, SPASS-meter may start various background threads, e.g., for periodically collecting system information. These threads should be terminated when the SUM terminates in order to prevent a hanging VM. This can either be achieved by the `shutdownHook` flag in `StartSytem` or by properly

marking each possible end of the SUM by `EndSystem` (e.g., in graphical systems multiple exists may exist depending on the programming style). For future compatibility, we suggest to use `EndSystem`.

- All recorder ids are trimmed, i.e. leading and trailing spaces are removed!

The example below[3] defines to two monitoring group, the first one called according to the qualified class name of CpuTimeTest, the second one called “exec”. Both operate with the default monitoring semantics of the global configuration. The program starts and ends at main (the global configuration `mainDefault=START_END` would have the same effect but without explicit annotations).

```
@Monitor
public class CpuTimeTest {

    @Monitor(id = "exec")
    private void execute() {
        // perform consumptive action
    }

    @StartSystem
    @EndSystem()
    public static void main(String[] args) {
        new CpuTimeTest().execute();
    }

}
```

5.3. XML-based configuration

Configuration in source code using annotations is not possible for all application cases, e.g., when there is no access to source code, source code should not be modified or annotations are not available.

5.3.1. Structure

Configuration files are stated in XML. In detail, the structure follows the source code annotations and their provided data, i.e., the table shown in Section 5.2 applies. Packages (`namespace`) and classes (`module`) can be specified in two forms, with fully qualified names as well as using the nested notation similarly to the package-class nesting in Java. If needed, attributes (`data`) or methods (`behavior`) may be given, the latter ones with Java source code method signatures.
In this section we discuss the additions to Section 5.2. The fragment below highlights the specific parts of the XML configuration language. These parts will be discussed in detail in the remainder of this document.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://sse.uni-hildesheim.de/instrumentation">
  <namespace name="java">
    <namespace name="lang">
      <module name="Object">
        <monitor id="TheObject" debug="NET_IN, NET_OUT"/>
        <data name="hashCode"/>
          <valueChange id="hashCodeValue"/>
```

[3] The SPASS-meter examples project provides a complete implementation of the example shown here.

```
          <valueContext id="hashCodeValueContext"/>
        </data>
        <behavior signature="equals(java.lang.Object)">
          <excludeFromMonitoring/>
        </behavior>
        <behavior signature="main(java.lang.String[])">
          <endSystem/>Error! Bookmark not defined.
          <startSystem shutdownHook="false"/>
        </behavior>
        <behavior signature="hashCode()">
          <Timer id="hashCode" state="START_FINISH"
            affectAt="DEFAULT" considerThreads="false"/>
          <VariabilityHandler/>
          <NotifyValue id="hashCode" expression="10"
            value=" VALUE" notifyDifference="true"/>
          <ConfigurationChange id="hashCode"
            valueExpression="configurationToString()"/>
        </behavior>
      </module>
    </namespace>
  </namespace>
</configuration>
```

- The root element is `configuration`. It links to the XMLSchema[4] and may optionally define if this configuration is `exclusive`, i.e. whether annotations given in source code should be ignored or not exclusive (`exclusive="false"`).
- A `namespace` corresponds to a Java package. The name may be given as a fully qualified name or as a simple package name. The content within a namespace is interpreted as specification on the contents of the namespace. Two specific attributes may be given here:
  - `pattern`: States a (Java) regular expression appended to the fully qualified name of the namespace in order to select contained elements.
  - `typeOf`: Constraints based on the programming language type in terms of fully qualified names. If given, the contained annotations are applied only if the concrete type is equal or a subtype of the type given in `typeOf`.

  In combination with `pattern` and `typeOf` also a specific `instanceIdentifierKind` may be given. Otherwise the one of the global configuration or its default are used.
- A `module` corresponds to a Java class. Similarly to namespaces, modules may be specified using relative or fully qualified names. Modules may appear on top-level, i.e. directly below `configuration`. Modules may be nested in order to refer to inner classes. The attributes `pattern` and `typeOf` apply as described for `namespace`.
- A `data` element corresponds to a Java attribute and may be stated only within a module. The `name` attribute states the name of the attribute.
- A `behavior` element corresponds to a Java method and is specified by its signature consisting of the method name, the parameter list (in braces) and the fully qualified parameter types as a comma separated list. Whitespaces are not allowed in a signature. As usual in Java, the return type is not part of a signature. Also thrown exceptions are not stated here.
- Source code annotations can be stated as contained elements whereby the data supported for an annotation is given in terms of attributes.

[4] In eclipse add the SSE-URL as key linked to the XMLSchema file to the XMLCatalogue and enable the validation of XML files.

5.3.2. Additional Information

This section details some additional information regarding the current implementation.

- At the moment we refrain from adding `pattern` and `typeOf` to the monitor annotation, because we expect it to be particularly beneficial for the external XML-based configuration. This could be an alternative to introduce these capabilities also for the annotation-based configuration style.
- We named the elements in a neutral style independent from a particular programming language in order to facilitate the application of this configuration language also for native programs etc. in future.
- We followed the nesting of elements as it is typical for object-oriented programming languages in order to avoid repeated names. Instead of this nested style also individual elements may be specified using their fully qualified name.
- Global configuration options such as `annotationSearch` may be given as XML elements in the top-level XML element.
- The XML element `groupConfiguration` directly nested in configuration may be used to consistently define a group specific configuration for multiple monitoring groups. A definition for the example would look like

  ```
  <groupConfiguration id="TheObject" debug="NET_IN, NET_OUT"/>
  ```

  while the application of the related monitoring group could then be reduced to

  ```
  <monitor id="TheObject"/>
  ```

  In particular, this applies to multiple applications of the same monitoring group, e.g., a monitoring group over multiple classes) and allows consistent definition of additional information for monitoring groups. The group configuration may also receive the other attributes `groupAccounting`, `resources` and `instanceIdentifierKind`. Attributes not stated explicitly will receive the same information as specified in the general configuration as described in Section 5.1 (except of debug which is empty by default). In case that the same configuration should be applied to multiple monitoring groups, it is possible to refer to one group configuration providing the information, e.g.

  ```
  <groupConfiguration id="TheObject1" refId="TheObject"/>
  ```

  provides the monitoring group `TheObject1` with the same configuration defined for the monitoring group `TheObject`. Referenced configuration may occur in any order, i.e., they need not to be specified before the referring configuration.
  Group configurations in XML files may also be applied in combination with source code annotation based configuration in order to improve consistency. In this case, the XML configuration file must not be authoritative (`exlusive="false"`).

Hint: Please note that due to performance reasons processing the `debug` information in `Monitor` may be disabled in framework code.

6. Remote monitoring

SPASS-monitor allows separating the data aggregation from physical resource monitoring and collection. Therefore, upon each preaggregated value from the collection layer an event is generated and send (currently only) via TCP to a dedicated Server which performs the data aggregation and recording.
When starting the instrumentation on the SUM, the additional parameter

```
tcp=<host>:<port>
```

activates the generation and transmission of events to the specified host on the given TCP port. The `tcp` option can alternatively be given as part of the XML monitoring scope. As the global `tcp` option is optional, all other parameter described in Section 5.1 can be applied and (except for `bootJar`, `debuglog`, `xmlconfig`) are transferred upon initialization of the `tcp` connection to the server. The server accepts the following command line parameters:

name	description and possible values	type and default
baseDir	base path for relocating the file specified by the `out` parameter	FileLocation (no default)
port	TCP port for listening to monitoring events	int (no default)

To start the server, call

```
java
  -classpath spass-meter-rt.jar
   de.uni_hildesheim.sse.monitoring.runtime.recordingServer.
     TCPRecordingServer
   baseDir=. port=6002
```

In this case the current directory is specified as base directory and monitoring events are received on port 6002.

7. Static instrumentation

For some applications, dynamic instrumentation at startup time is not adequate. This is particularly the case for Android Apps or when a large (base) system with a stable codebase is used and dynamic instrumentation affects startup time significantly, e.g. for a J2EE container server. As the codebase is stable, an alternative is to instrument the stable parts of the system once (before runtime, i.e. in a static fashion) and to instrument changing parts at runtime (if needed at all).

7.1. Java Programs

For static instrumentation, SPASS-meter is used as a development tool. The direct call on the command line is[5]

```
java
  -classpath spass-meter-ant.jar;<required libs>
   <input jars> <output directory> <configuration>
```

Whereby

- *<input jars>* denotes the JAR files to be instrumented (separated by comma, in quotes if spaces are used in the file paths)
- *<required libs>* is the list of libraries required for building the JARs in *<input jars>*
- *<output directory>* points to an existing directory where to put the instrumented JARs to (each input JAR is transformed to a JAR with the same name in the output directory>
- *<configuration>* is a command line configuration of SPASS-meter as described in Section 5.1.

[5] use the distribution jar for your respective operating system. `spass-meter-ant.jar` also includes the ANT task for SPASS-meter. Alternatively you can use `spass-meter-static.jar` for a direct command line call / integration. The respective class is `de.uni_hildesheim.sse.monitoring.runtime.preprocess.Preprocess`

Alternatively, the static instrumenter may also be used within an ANT file. The advantage of using ANT is that you can use full ANT paths including wildcards, which are passed on for instrumentation to the static Therefore, define the SPASS-meter task by

```
<taskdef resource="spass-meter.properties" onerror="ignore">
     <classpath>
          <pathelement location="spass-meter-ant.jar"/>
     </classpath>
</taskdef>
```

And execute the instrumentation within an ANT target using[6]

```
<spassInstrumenter
     classpathref="<required libs>"
     in="<input jars>"
     out="<output directory>"
     params="<configuration>" />
```

For executing the statically instrumented program, please add `spass-meter-boot.jar` and `spass-meter-runtime.jar` to the classpath. In case of mixed instrumentation (static and dynamic for one program) just use the instrumented JARs with the dynamic instrumentation commands discussed above.

7.2. Android Apps

In this Section we briefly describe the integration into the Android build process. Currently, Android Apps need to be instrumented statically as (at least until Android version 2.3) does not support dynamic instrumentation. However, we started our work on the Android build process before ANT and the Android build process supported task extension, so we rebuild the Android compile steps. It might be that this is easier in your specific environment. The library directory of the application shall contain the contents of the `libs` directory of the SPASS-meter Android distribution (including the two version of the native library).
Basically, we recommend packaging all Java classes (before calling the `dex` tool) into one JAR and statically instrument that JAR before turning it into an APK file.
Using the SPASS-meter ANT integration this can be achieved as shown below:

```
<spassInstrumenter classpathref="${classpath.instrumentation}"
    in="${app.jar}"
    out="${instrumented.dir}"
    params="${spass.args}" />
```

Here, `classpath.instrumentation` shall contain the other JAR files in the SPASS-meter distribution for Android and `spass.args` contains the global configuration options. Please note that the globalOption `registerThreads` shall be set to true due to technical reasons.

8. Monitoring results

Dependent on the actual configuration, SPASS-meter provides its results on different levels of granularity. In this Section, we will focus on the default post-mortem summary, i.e., a specific summary after the SUM is terminated. This mechanism can be customized using an internal plugin-

[6] Currently no dirsets or filesets are considered.

interface so that more specific data can be emitted. Further interfaces allow obtaining specific data at runtime (a JMX-based and a WildCat-based mechanism will be described in Section 9).

Let us assume that we perform monitoring of the simple example shown in Section 5.2 with the following parameters:

```
java –javaagent: spass-meter-ia.jar=logLevel=OFF,out=test.out
example.cpuTime.CpuTimeAnnotation
```

A post-mortem textual summary output file will be written to `test.out` in the current directory. A tabular formatting of such a file is shown below.

description	mem use	mem allo	sys time	agg sys ti	cpu time	in	netin	filein	out	netout	filout	jvm load	sys load
(threaded)													
1					31200200								
Program	2,46E+08	160112	4,44E+09	0	46800300	367	0	367	0	0	0	100,00%	12,10%
CpuTimeXml	112	112	2E+09	0	15600100	367	0	367	0	0	0	33,33%	4,03%
exec	160000	160000	2,42E+09	0	15600100	0	0	0	0	0	0	33,33%	4,03%
BREAKDOWN													
System min	3,2E+09												0,00%
System avg	3,2E+09					1,67E+11			1,45E+09				17,92%
System max	3,21E+09												26,64%
JVM min	24162304												0,00%
JVM avg	25063424					4934187			408064				12,10%
JVM max	25288704												25,48%
JVM vs. System	25063424					4934187			408064			12,10%	17,92%
recorder vs. sys	0	0	0	0	0	0	0	0	0	0	0		0,00%
recorder vs. jvm	0	0	0	0	0	0	0	0	0	0	0	0,00%	
CpuTimeXml vs. sys	112	112	2E+09	0	15600100	367	0	367	0	0	0		4,03%
CpuTimeXml vs. jvm	112	112	2E+09	0	15600100	367	0	367	0	0	0	33,33%	
exec vs. sys	160000	160000	2,42E+09	0	15600100	0	0	0	0	0	0		4,03%
exec vs. jvm	160000	160000	2,42E+09	0	15600100	0	0	0	0	0	0	33,33%	
CONFIGURATIONS													
1| exec,	160112	160112	2,42E+09	0	31200200	367	0	367	0	0	0	66,67%	8,07%
1| exec, vs. sys	160112	160112	2,42E+09	0	31200200	367	0	367	0	0	0		8,07%
1| exec, vs. jvm	160112	160112	2,42E+09	0	31200200	367	0	367	0	0	0	66,67%	

A SPASS-meter post-mortem summary consists of three parts, the first one summarizing the monitoring groups, the second one showing a breakdown according to the monitoring levels and a third one regarding “configurations”. However, the last section is currently experimental and we do not discuss that in detail at the moment.

- **Monitoring groups:** The data recorded for all specified monitoring groups as well as for the entire SUM are given. Further, a breakdown of the time consumption of all threads is shown. Please note that `CpuTimeXml`[7] (the main class of the SUM) and the monitoring group `exec` are non-overlapping, i.e., `exec` consumes most resources at runtime while `CpuTimeXml` the remainder of the resources of the main class. Dependent on the configuration, Program shows the resource consumption of the entire SUM within the JVM.
- **Breakdown:** The first three lines show the minimum, average and maximum (if available) resource consumption of the entire System while runtime. The next three lines indicate similar values for the resource usage of the JVM process from outside. Then the JVM, the SUM (called *recorder* here and the monitoring groups are broken down with respect to their factional resource consumption compared with the System and the JVM.

[7] In a real summary file this would be the qualified name of the class but we omitted this in the table due to formatting reasons.

9. SPASS-meter Extensions

SPASS-meter can be extended via a set of interfaces in `de.uni_hildesheim.sse.monitoring.runtime.plugins`, such as

- **Plugin**: Classes to be instantiated and initialized during startup phase of SPASS-meter. Comma-separated list in the system property `spassmeter.plugins`.
- **MonitoringGroupChangeListener**: A change to a monitoring group happened.
- **TimerChangeListener**: A change to a timer (see section 5.2) happened.
- **ValueChangeListener**: A change to an attribute value (see section 5.2) happened.

Currently, default listeners such as JMX (see Section 9.1) or WildCat (see Section 9.3) are registered in the `PluginRegistry` class while other listeners can be packaged with the SPASS-meter JARs and specified in a file named `plugin.lst` given in the root directory of the respective JAR (each plugin class in a separate line). Currently, we provide prepackaged versions in particular for JMX. Please note that the actual versions of both extensions are intended as demonstrations (realized as part of the MSc thesis of Stephan Dederichs).

9.1. JMX support for SPASS-meter

The JMX (Java Management Extensions) support for SPASS-meter is external to the SPASS-meter project, as not all Java platforms really support JMX, e.g. the Android platform. The JMX support JARs are build in an own project which relies on SPASS-meter.

Please note that for enabling JMX functionality the JVM parameter

`-Dcom.sun.management.jmxremote`

must be set for the JVM executing the instrumented program.

The following additional SPASS-meter global parameters are recognized by the JMX extension

name	description and possible values	type and default
jmxConfig	path to the XML-based configuration file for the JMX extension	FileLocation (default)

The JMX support can be configured by a XML file. Data collected by SPASS-meter from a SUM will be added by default. However, system-level data needs to be specified individually in order to control the overhead. Doing so requires additional knowledge about the provided classes and their individual data attributes. In detail, the structure of the XML configuration file is discussed below.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <functions>
    <addedfunction name="<classname>"
      displayname="<displayname>"/>
    ...
  </functions>
  <classes>
    <class name="<classname>" displayname="<displayname>">
      <attribute name="<attributename>" type="<type>">
        <function name="<functionname>"
          description="<description>"/>
      </attribute>
      ...
    </class>
    ...
  </classes>
</configuration>
```

The root element is `configuration`. It contains two elements:

- `functions`: The element `functions` represents user defined functions realized in terms of individual classes complying to a specific interface for aggregating system values. It contains arbitrary elements of type
    - `addedfunction`: This element specifies an user defined function with the two attributes `name` and `displayname`. The attribute `name` specifies the class name of the additional function and the attribute `displayname` specifies the name for accessing the function.
- `classes`: This element represents all (data) classes which will be available in the JMX support. Therefore it contains any quantity of the element
    - `class`: This element represents a data class which will be available in the JMX support. The attribute `name` specifies the class name of the class and the attribute `displayname` specifies the name for displaying the class in JMX. If functions should be used on attributes of the class, the element `attribute` must be defined. Such element can be defined for each numerous attribute of the class.
        - `attribute`: Specifies an attribute on which one or more functions will be executed. An attribute is defined by the attributes `name` and `type`. The attribute `name` specifies the name and the attribute `type` the java type of the attribute. It can contain any quantity of the element function, but at least one function must be defined.
            - `function`: Specifies a function which aggregates the values of an attribute. Additional to the user defined functions, there are three pre defined functions (min, max and avg) which can be used. For every function the attributes `name` and `description` must be specified. The `name` specifies the function which will be used and the `description` stores a description of the value which is calculated through the function.
            -

An example configuration is part of the SPASS-meter JMX distribution. An excerpt from the example is shown below. It enables the display of system-level memory information through the class `JMXMemoryData` shown on the display under the category `Memory` in terms of the `CurrentMemoryUse` aggregated to minium, maximum and average values.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <functions />
  <classes>
    <class
     name="de.uni_hildesheim.sse.jmx.services.dynamic.JMXMemoryData"
     displayname="Memory">
     <attribute name="CurrentMemoryUse" type="long">
       <function name="min"
         description="The minimal current memory use."/>
       <function name="max"
         description="The maximal current memory use."/>
       <function name="avg"
         description="The avg current memory use."/>
       </attribute>
     </class>
  </classes>
</configuration>
```

9.2. JMX console extension for SPASS-meter

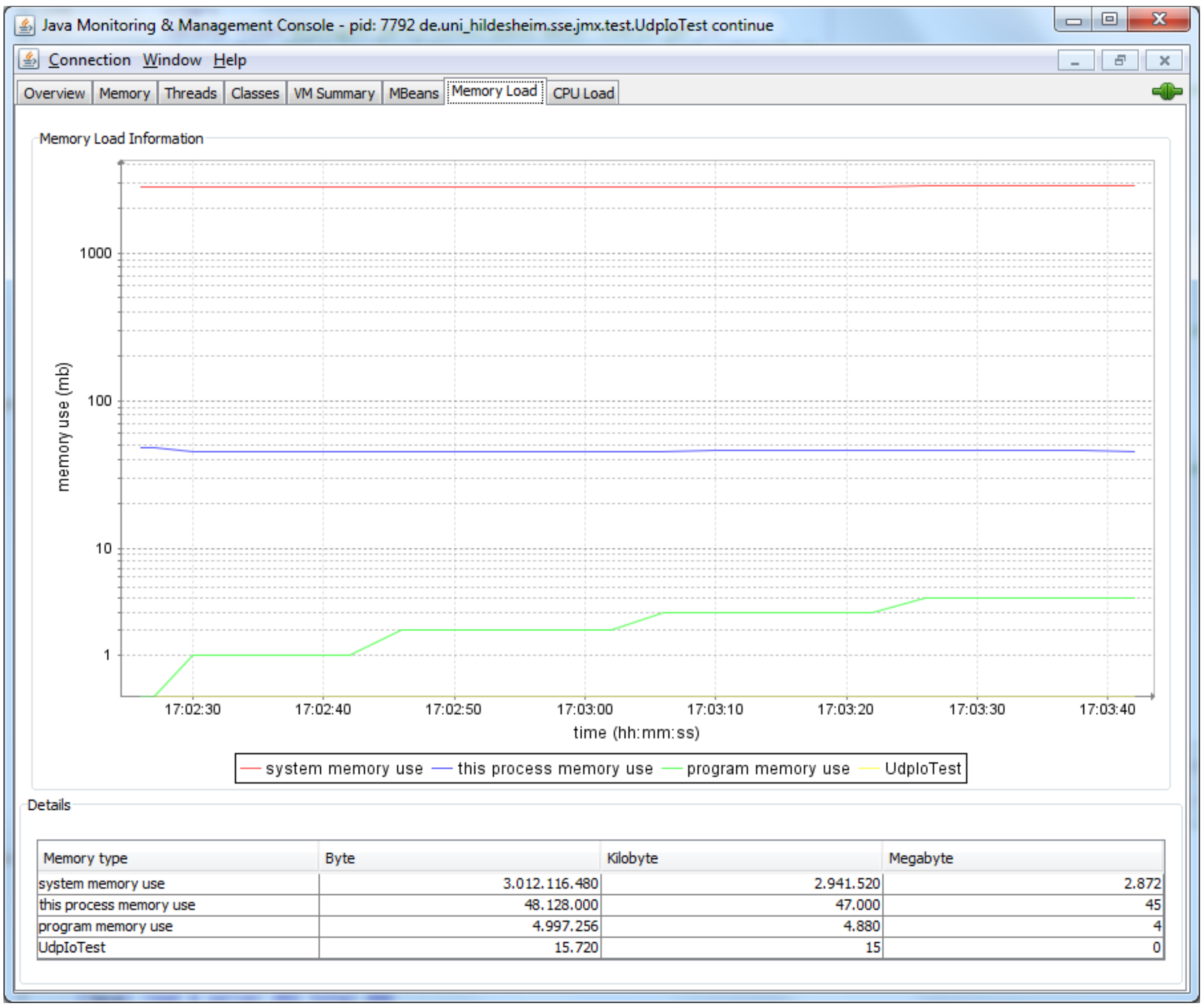
The default JMX console displays the management beans dynamically added to the JVM at runtime as specified above. Integer values obtained from the management beans may be displayed as simple time graphs. The default extension of the JMX console for SPASS-meter allows graphical comparison of related values, e.g. system load, JVM process load, Java program inside JVM, monitoring groups including the SPASS-meter monitoring groups. The required plugin files are part of the SPASS-meter JMX console bundle.

To start the JMX console add the following line in a command line

```
jconsole.exe -pluginpath <Path_to_Plugin>\LoadPlugin-rt.jar
```

`<Path_to_Plugin>` must be replaced by the location of the JAR file *LoadPlugin.jar*.

The following screenshot shows the memory load tab in the JMX console while monitoring the program `UdpIoTest` declared as a SPASS-meter monitoring group.



Memory type	Byte	Kilobyte	Megabyte
system memory use	3.012.116.480	2.941.520	2.872
this process memory use	48.128.000	47.000	45
program memory use	4.997.256	4.880	4
UdpIoTest	15.720	15	0

9.3. WildCat support for SPASS-meter

WildCAT[8] is a generic framework for context-aware applications. It allows the monitoring of applications by organizing and access sensors through a hierarchical organization backed with a powerful SQL-like language to inspect sensors data and to trigger actions upon particular conditions. The WildCAT support for SPASS-meter is external to the SPASS-meter project, as not all Java platforms really support WildCAT, e.g. the Android platform. The WildCAT support JARs are build in an own project which relies on SPASS-meter.

The following additional SPASS-meter global parameters are recognized by the WildCAT extension

name	description and possible values	type and defaut values
wildcatConfig	path to the XML-based configuration file for the WildCAT extension	FileLocation (no default)
wildcatGUI	enables the WildCAT console	`TRUE`, `FALSE`

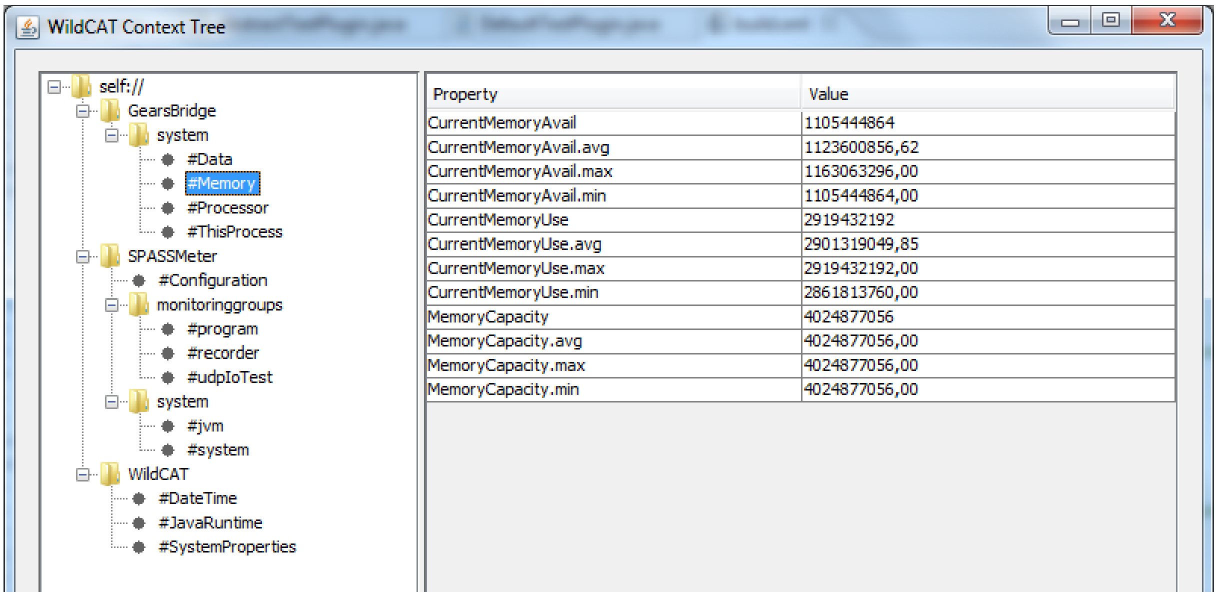
As the JMX support, also the WildCAT support can be configured by a XML file. In detail, the structure of the XML configuration file is exactly the same as by JMX support. For more information have a look at Section 9.1.

9.4. WildCAT console support for SPASS-meter

WildCAT cannot be used in combination with the JMX console. For accessing the managed values the WildCAT integration contains a build-in user interface. For using the user interface the JVM parameter must be set

```
wildcatGUI=true
```

The following screenshot shows the WildCAT console.



```
self://
  GearsBridge
    system
      #Data
      #Memory
      #Processor
      #ThisProcess
  SPASSMeter
    #Configuration
    monitoringgroups
      #program
      #recorder
      #udpIoTest
    system
      #jvm
      #system
  WildCAT
    #DateTime
    #JavaRuntime
    #SystemProperties
```

Property	Value
CurrentMemoryAvail	1105444864
CurrentMemoryAvail.avg	1123600856,62
CurrentMemoryAvail.max	1163063296,00
CurrentMemoryAvail.min	1105444864,00
CurrentMemoryUse	2919432192
CurrentMemoryUse.avg	2901319049,85
CurrentMemoryUse.max	2919432192,00
CurrentMemoryUse.min	2861813760,00
MemoryCapacity	4024877056
MemoryCapacity.avg	4024877056,00
MemoryCapacity.max	4024877056,00
MemoryCapacity.min	4024877056,00

10. Acknowledgements

We would like to thank Stephan Dederichs for his contributions to SPASS-meter and his work on the JMX and WildCat integration.

[8] http://wildcat.ow2.org/

This work is partially supported by the INDENICA project, funded by the European Commission grant 257483, area Internet of Services, Software & Virtualisation (ICT- 2009.1.2) in the 7th framework programme. Further, this work is partially supported by the QualiMaster project, funded by the European Commission grant 619525 in the 7th framework programme.

11. Appendix: Advanced global configuration options

In this section we provide the list of advanced configuration option which extends the basic configuration options discussed in Section 5.1.

name	description and possible values	type and default
`logLevel`	One of the Java logging levels.	`FINE`, `FINER`, `FINEST`, `INFO`, `OFF`, `SEVERE`, `WARNING`
`bootjar`	Additional jar file to be added dynamically to the boot classpath (experimental)	File location
`debuglog`	Write debugging specific information to the given file. This is helpful in case that logging and standard output is intensively used by the SUM, e.g., a web server.	File location
`pruneAnnotations`	If enabled, unused SPASS-meter annotations are removed from the resulting byte code (experimental).	`TRUE`, `FALSE`
`memAccounting`	Defines the strategy for memory accounting; in general, unallocation is better supported on JVMTI-enabled virtual machines.	o `NONE` (no memory accounting at all) o `CONSTRUCTION_UNALLOCATION` (objects in constructor and finalize, default) o `CONSTRUCTION_NATIVEUNALLOCATION` (objects in constructor, native unallocation) o `CONSTRUCTION` (objects in constructor only) o `CONSTRUCTION_NATIVEUNALLOCATION_ARRAYS` (objects in constructor, arrays at new, native unallocation) o `CONSTRUCTION_ ARRAYS` (objects at new, arrays at new) o `CREATION_UNALLOCATION` (objects at new and in finalize) o `CREATION_NATIVEUNALLOCATION`(objects at new, native unallocation) o `CREATION` (objects at new only) o `CREATION_UNALLOCATION_ARRAYS` (objects at new and finalize, arrays at new) o `CREATION_NATIVEUNALLOCATION_ARRAYS` (objects and arrays at

		new, native unallocation) o `CREATION_ ARRAYS` (objects at new, arrays at new)
`accountableResources`	Defines the default accountable resources.	`CPU_TIME`, `FILE_IO`, `MEMORY`, `NET_IO`, default is empty, i.e., all
`sumResources`	Defines the accountable resources for the SUM.	`CPU_TIME`, `FILE_IO`, `MEMORY`, `NET_IO`, default is empty, i.e., all
`defaultGroupResources`	Defines the default resources for groups which do not explicitly specify resources.	`CPU_TIME`, `FILE_IO`, `MEMORY`, `NET_IO`, default is empty, i.e., all
`annotationSearch`	How should annotations in interfaces and superclasses be considered during instrumentation? May enable “inheritance” of annotations if required.	`NONE, INTERFACES, SUPERCLASSES, ALL`
`multiDistributeValues`	Should resource consumption in multi groups (multiple ids) be accounted as such to all related groups or evenly distributed?	`TRUE, FALSE`
`multiConsiderContained`	Should accountable resources of related groups in multi groups be considered or only the accountable resources of the concrete multi group?	`TRUE, FALSE`
`accountExcluded`	Explicitly account excluded SUM parts to a monitoring group or account them only for the entire program.	`TRUE, FALSE`

Further options provided by SPASS-meter are either deprecated or experimental and, thus, not listed here.