February 27th 2008

CLIF user manual and programmer's guide

http://clif.objectweb.org/

Table of contents

1. Introduction......4
2. Key concepts......5
3. How to get CLIF working?......7
3.1. Technical requirements......7
3.2. Ready-to-use distributions......7
3.3. Generating a runtime environment (optional)......8
3.4. Configuring CLIF......8
3.5. Checking Clif version and execution environment......9
4. CLIF servers and the Registry......10
4.1. Rationale......10
4.2. Running a Registry......10
4.3. Running CLIF servers......10
5. Probes......11
5.1. Rationale......11
5.2. Available probes......11
5.2.1. cpu probe......11
5.2.2. disk probe......12
5.2.3. memory probe......12
5.2.4. network probe......12
5.2.5. jvm probe......12
5.3. Defining your own probes......13
5.3.1. Relying on the provided probe framework......13
5.3.2. Implementing a Blade component......13
6. Load injectors and ISAC......14
6.1. Rationale......14
6.2. ISAC is a Scenario Architecture for CLIF......14
6.2.1. behaviors......14
6.2.2. load profiles......14
6.2.3. ISAC plug-ins......14
6.2.4. Writing an ISAC scenario......15
6.2.5. Recording an ISAC scenario for Http......15
6.2.6. Deploying and executing an ISAC scenario......16
6.3. Defining your own load injectors (Java programmers)......16
6.3.1. Using MTScenario utility class......16
6.3.2. Writing your own ISAC plug-ins......16
6.3.3. Implementing a Blade component......21
7. Eclipse-based graphical user interface......22
7.1. Introduction......22
7.2. Getting started......22
7.2.1. From an Eclipse-RCP based standalone CLIF distribution......22
7.2.2. From the Eclipse plug-ins......22
7.3. Test plan edition......24
7.4. ISAC scenario edition......25

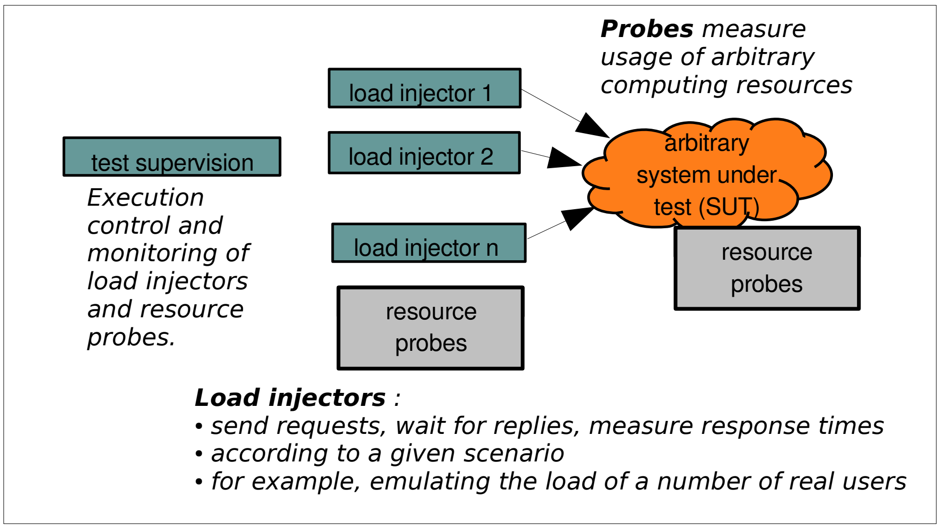
7.5. test deployment and execution....26
7.6. ISAC plug-in creation Wizard....27
8. Java Swing-based graphical user interface....28
8.1. Introduction....28
8.2. Getting started....28
8.3. Test plan edition table....28
8.4. Performance and resource usage monitoring....29
8.5. File Menu....30
8.6. Test plan menu....30
8.7. Tools menu....30
8.7.1. Basic analyzer....30
8.7.2. Quick graphical analyzer....30
8.8. ? (help) menu....35
9. Command line user interface....36
9.1. Introduction....36
9.2. Run CLIF Registry....36
9.3. Test plan deployment: deploy....36
9.4. Test initialization: init....36
9.5. Test execution start: start....37
9.6. Suspend test execution: suspend....37
9.7. Resume test execution: resume....37
9.8. Stop test execution: stop....37
9.9. Wait for a test execution to terminate: join....37
9.10. Collect test results (measurements): collect....37
9.11. Shortcut for full test execution process: run....37
9.12. Shortcut for full deployment and execution process: launch....38
9.13. Get specific runtime parameters of a probe or injector: params....38
9.14. Change a runtime parameter of a probe or injector: change....38
10. Test results and measurements....39
11. Licenses....40
Appendix A: XML DTDs for ISAC....41
Appendix B: injector and probe (aka blade)'s life cycle....46
Appendix C: system properties....47
Appendix D: Class and resource files (remote) loading....50
Appendix E: ISAC plug-ins....51
Appendix F: ISAC execution engine....52

1. Introduction

CLIF is a component-oriented software framework written in Java, designed for load testing purposes of any kind of target system. By load testing, we mean generating traffic on a System Under Test in order to measure its performance, typically in terms of request response time or throughput, and assess its scalability and limits, while observing the computing resources usage.

Basically, CLIF offers the following features:

- deployment, remote control and monitoring of distributed load injectors;
- deployment, remote control and monitoring of distributed probes;
- final collection of measurements produced by these distributed probes and load injectors.



Probes measure usage of arbitrary computing resources
load injector 1
load injector 2
load injector n
test supervision
Execution control and monitoring of load injectors and resource probes.
arbitrary system under test (SUT)
resource probes
resource probes
Load injectors :
• send requests, wait for replies, measure response times
• according to a given scenario
• for example, emulating the load of a number of real users

Analysis tools for these measurements will be provided as soon as possible. For the time being, all measurements are available as CSV (comma separated values)-formated text files.

Thanks to its component-based framework approach, CLIF is easily customizable and extensible to particular needs, for example, in terms of specific injectors and probes, definition of load generation scenarios, storage of measurements, user (tester) skills, integration to a test management platform, etc. For instance, user interfaces are available as command-line tools, Java Swing-based GUI and Eclipse-based GUI.

2. Key concepts

- *blade*
  an active component that can be deployed within a CLIF application, under control of the supervisor component, that provides statistical information about its execution (for monitoring purpose), and produce results stored by the storage component. Blades exist either as load injectors or probes.
- *CLIF application*
  set of deployed components making it possible to run a test. A CLIF application is a distributed component holding as sub-components: one supervisor, one storage, and an arbitrary number of probes and load injectors (aka blades).
- *CLIF server*
  a JVM with a bootstrap component that will locally handle blade deployment requests from the supervisor. In other words, one must run a CLIF server on a given computer in order to be able to deploy load injectors and probes. CLIF server have a name. They register themselves in the Registry with this name in order to be found by the deployment process.
- *code server*
  the code server is responsible for delivering Java byte-code and resource files on demand during the deployment process. This is achieved through a socket server with a specific protocol. As of current version, files greater then 2GB cannot be transfered.
- *collect, collection*
  action of getting all measurements, possibly disseminated through the blades by the storage proxy feature, into the storage component. Collection should not occur before a test is terminated.
- *deployment*
  local or remote instantiation of load injectors and probes (aka blades). During this process, Java byte-code and resource files may be loaded from the code server, through the network, and to the target JVM of the blade being deployed.
- *load injector*
  a component that conforms to the blade component type, whose activity consists in generating traffic on an arbitrary SUT, using arbitrary protocols, according to an arbitrary scenario.
- *probe*
  a component that conforms to the blade component type, whose activity consists in measuring the usage of an arbitrary computing resource. Probes may be deployed at the SUT's side, in order to better analyze and understand its performance, as well as at the load injectors' side, to check that they are performing all right (since saturating injectors may result in unreliable measurements or violated load scenarios).
- *(load) scenario*
  optional concept referring to the way a single load injector generates traffic, for instance by emulating the load of a variable number of users performing a variety of requests on the SUT. In other words, a scenario defines both shape and content of the traffic generated by a load injector.
- *Storage*
  centralized component for storing measurements produced by load injectors and probes (aka blades). The storage component is typically associated to a storage proxy feature supported by each blade.

- *Storage proxy*
  local buffering of measurements feature provided by blades in order to avoid flooding the network and the storage component, which could also disturb the test and spoil measurements.
- *Supervisor or supervision console*
  component responsible for controlling and monitoring of a test execution.
- *System under test (SUT)*
  an arbitrary system one wants to assess the performance of. It is typically composed of one or several computers, networks, etc. It has to be reachable, either directly or indirectly via some gateway, native library or any wrapping mechanism, from the Java Virtual Machine where CLIF servers are running.
- *Registry*
  a distributed naming service used by the deployment process to lookup CLIF servers and deploy load injectors and probes.
- *Test (execution)*
  execution (shot) of an already deployed test plan. A test ends under 3 possible conditions: completed, manually stopped or self-aborted.
- *Test plan*
  specifies a set of distributed load injectors and probes, including their instantiation arguments and the name of the CLIF servers where they must be deployed.

3. How to get CLIF working?

3.1. Technical requirements

CLIF framework and provided load injectors are 100% Java™. CLIF requires a Java runtime environment (JRE) or development kit (JDK), and the Java-based ant utility from Apache.org. Current CLIF version is known to be working with:

- Sun JDK 5.0 (also known as J2SDK™ 1.5)
  Download from http://java.sun.com/javase/downloads/index_jdk5.jsp
- Apache ant utility version 1.5.4 or greater
  download from http://ant.apache.org/bindownload.cgi
  Make sure ant is using the right JDK!
- Linux 2.4 and 2.6 kernels
- MacOS.X tiger
- Microsoft Windows XP™

System probes for Linux are also 100% Java, while system probes for Windows and MacOS.X are native (C-code embedded in Java code via the Java Native Interface).

Since CLIF is written in Java, the only constraint about the SUT is that it must be reachable from a Java Virtual Machine (JVM), either directly or indirectly through some wrapping, gateway or native library.

There are two ways of getting a CLIF runtime environment: either by getting the whole source from the CVS repository, or by getting a ready-to-use binary distribution.

3.2. Ready-to-use distributions

CLIF's site at ObjectWeb Forge offers several binary distributions, available as zip files (see http://forge.objectweb.org/project/showfiles.php?group_id=57):

- `clif`
  full runtime environment with a Java Swing based GUI and support for CLIF servers.
- `server`
  reduced runtime environment just for running CLIF servers.
- `console-Linux`
  Eclipse-RCP based standalone console for Linux/Intel.
- `console-Windows`
  Eclipse-RCP based standalone console for Windows/Intel.
- `console-Macosx`
  Eclipse-RCP based standalone console for Windows/Intel.
- `clif-plugin`
  CLIF console as an Eclipse plug-in. Refer to section 7 for plug-in installation.
- `isac-plugin`
  ISAC editor as an Eclipse plug-in (requires CLIF console plug-in). Refer to section 7 for plug-in installation.

You may unzip these files wherever you like, except Eclipse plug-ins that you may install in your Eclipse environment (see section 7). Avoid installing a distribution in a directory containing a whitespace character in its path (set-up problems have been reported in some conditions).

3.3. Generating a runtime environment (optional)

Optionally, you may want to recompile CLIF and generate your own runtime environment. This task is quite easy using the `ant` utility. Main targets are:

- `ant dist`
  compiles CLIF and generates a runtime environment with a Swing GUI, and Javadoc-style API documentation, available in output/dist subdirectory
- `ant server`
  compiles CLIF and generates a minimal runtime environment to run a CLIF server, zipped in output subdirectory
- `ant product`
  compiles CLIF, generates CLIF plug-ins for Eclipse (console and isac) and a standalone Eclipse™ RCP based full-fledged runtime environment for the current operating system, available as .zip files in `output` subdirectory
- `ant zip`
  compiles CLIF and generates binary distributions including the generic Swing-based GUI and server run-times, in separated zip files available in output subdirectory.
- `ant clean`
  destroys output directory

Then, subsequent operations are given in the following sections, considering the `output/dist` subdirectory as "CLIF's runtime environment root directory".

The source code is available through a CVS repository at ObjectWeb's forge. You may obtain the source code using CVS utility or by downloading a nightly-built snapshot of CLIF's CVS repository as a single zipped file (see information at http://forge.objectweb.org/scm/?group_id=57).

3.4. Configuring CLIF

From now on, you are supposed to set CLIF's runtime environment root directory as your current directory. You may configure CLIF either by editing file `clif.props` in `etc/` subdirectory, or by using command "`ant config`". In the latter case, the following questions will be asked:

- *please enter the host where the console will be run:*
  enter the IP address or name of the computer where you will run the Registry, either embedded in the Swing or Eclipse GUI, or launched by command line.
- *please enter the port number for the console embedded code server:*
  enter the port number used by the code server, for example 1357.

This configuration operation must be done everywhere you want to run a CLIF server or a console. You may also make this configuration step only once, and copy the resulting file `etc/clif.props` wherever needed.

Note that this configuration utility uses file `etc/clif.props.template` as a template. You may edit this file to change some default Java properties so that any further configuration will keep your changes.

Should you edit file `etc/clif.props`, refer to the appendix on System properties page 47.

3.5. Checking Clif version and execution environment

Use command “`ant version`” to get the version numbers of Java environment, operating system and CLIF. Command “`ant -version`” gives the ant version.

4. CLIF servers and the Registry

4.1. Rationale

CLIF servers are necessary to deploy any test plan, since they host load injectors and probes. CLIF servers are designated by a name, which is registered in a Registry. In order to run, CLIF servers must be able to find this Registry, which implies:

1. that the Registry must be running before a CLIF server can be launched;
2. that parameters must be given to tell the CLIF servers where to find the Registry and register themselves.

4.2. Running a Registry

There are three ways of starting a Registry: running the Java Swing console GUI (section 8), using the Eclipse-based console GUI (section 7), or using the appropriate command (section 9).

4.3. Running CLIF servers

CLIF must be configured on each host you plan to run a CLIF server (see section 3), accordingly to where your Registry is running. Then, run a CLIF server with command:

- `ant server`
  to create a CLIF server that registers with the local host name as CLIF server name
- `ant -Dserver.name=myFirstServer`
  to create a CLIF server that registers with the provided name

The second solution is a good practice for defining test plans regardless of the actual execution computers you will have, since the CLIF servers' names are not computer names. You may even first locally try a distributed test plan by running as many CLIF servers as needed on a single computer, with different CLIF server names.

5. Probes

5.1. Rationale

When load testing, it is often a good idea to check the usage of computing resources, both at the SUT side and the injectors' side. For instance, one may imagine system probes measuring CPU usage percentage, memory consumption, network bandwidth, etc. But other probes may be imagined that measure the size of a request queue length, a cache usage, or any activity data of any kind of middleware/software element involved in the SUT.

With CLIF, you may include probes in a test plan, as a complement to load injectors. Probes are supposed to have their own activity, typically (but not necessarily) consisting in polling a resource to measure its usage. All measurements are available from the Storage component once the test execution is over and the collection process has completed, while statistical values may be retrieved by the supervision console for monitoring purpose during test execution, directly from the probe. These statistical values are moving statistics computing on the period between two consecutive retrievals.

5.2. Available probes

Probes delivered with CLIF all consist in a periodic measure of the resource. They all take two arguments that must be specified in the test plan: the polling period (in milliseconds) and the execution duration (in seconds). Although probes start measuring once initialized for convenience, this execution time is counted once actually running (i.e. started and not suspended). When terminated, no measure is performed anymore.

To set a probe in a test plan:

- enter its family name as the “class name” information field;
- select the “probe” type;
- select the CLIF server where to deploy this probe, making sure that the target CLIF server actually runs on a computing environment (hardware, operating system or whatever) that is compatible with the probe family (see table below);
- enter the specific argument line, as explained hereafter.

5.2.1. cpu probe

family/class name	cpu
measurements	global used CPU %, user used CPU %, kernel/privileged used CPU %
alarms	*none*
arguments	polling period (ms), execution duration (s)
compatibility	Linux 2.4/2.6, MacOS.X 10.4, Windows XP

5.2.2. disk probe

family/class name	disk
measurements	# issued read operations, # of sectors read, # issued write operations, # of sectors written, time spent for I/O (ms), time spent for read operations (ms), time spent for write operations (ms).
alarms	*none*
arguments	polling period (ms), execution duration (s), disk name (e.g. hda or sda for Linux, disk0 for MacOS.X, C: for Windows XP)
compatibility	Linux 2.4/2.6, MacOS.X 10.4, Windows XP

5.2.3. memory probe

family/class name	memory
measurements	used RAM %, used RAM (MB), cached memory (MB), buffers size (MB), used swap %, used swap (MB)
alarms	*none*
arguments	polling period (ms), execution duration (s)
compatibility	Linux 2.4/2.6, MacOS.X 10.4, Windows XP

5.2.4. network probe

family/class name	network
measurements	received KB, # of packets received, sent KB, # of packets sent
alarms	*none*
arguments	polling period (ms), execution duration (s), network adapter name (e.g. eth0 for Linux, en0 for MacOS.X, Broadcom NetXtreme 57xx Gigabit Controller for Windows XP)
compatibility	Linux 2.4/2.6, MacOS.X 10.4, Windows XP

5.2.5. jvm probe

family/class name	jvm
measurements	free memory in currently allocated heap (MB), used memory % with regard to currently allocated heap, free % of maximum allocatable memory heap
alarms	An alarm with severity level “Info” is generated at each JVM garbage collection.
arguments	polling period (ms), execution duration (s)
compatibility	system independent

5.3. Defining your own probes

5.3.1. Relying on the provided probe framework

You may define your custom probes very easily by using the probe framework used by the provided probes. To do so, you must define a sub-package of package `org.objectweb.clif.probe`, and create three classes:

- a `DataCollector` class extending class, whose role is basically to provide statistical values for monitoring;
- an event class implementing interface `BladeEvent` to hold the set of values produced by each measure;
- an Insert class implementing the method that actually performs the measures and produces the events of the class defined below.

For example, let's assume you want to define a weather probe sensing temperature and pressure. Then you will define the following classes:

- `org.objectweb.clif.probe.weather.DataCollector`
- `org.objectweb.clif.probe.weather.MyWeatherEvent`
- `org.objectweb.clif.probe.weather.Insert`

Note that the package path construction is mandatory, as well as the `DataCollector` and Insert class names, in order the deployment system to find your probe. The event class name is up to you. Once you have compiled your probe, build a jar file with the classes and copy it to CLIF's `lib/ext` directory. Then start a CLIF console and set your probe in the test plan by typing “weather” for the so-called “class name” field.

5.3.2. Implementing a Blade component

[TODO]

6. Load injectors and ISAC

6.1. Rationale

Load injectors are set in a CLIF test plan in order to generate traffic on the SUT. With CLIF, you may use and imagine any kind of way to define and execute your load scenarios, on any kind of SUT. You may even mix a variety of load injectors in the same test plan. This is the reason why you must set a class name for each load injector you define in a test plan, and set an arbitrary line of arguments, specifically to the actual load injector you use. Fortunately for non-programmers, CLIF comes with the ISAC extension in order to provide an easy, powerful and user-friendly way to define load scenarios. Luckily for Java programmers, they may also define their own load injectors.

6.2. ISAC is a Scenario Architecture for CLIF

With ISAC, testers are given a way to define load scenarios by combining:

- definitions of elementary behaviors, typically representing users;
- optional definitions of load profiles setting the population (i.e. the number of active instances) of each behavior as a function of time.

6.2.1. behaviors

An ISAC behavior basically consists in a sequence of actions (requests) on the SUT interlaced with delays (think times). It may be enriched with the following constructs:

- conditional loop: while <condition>
- conditional branches: if <condition> then <true_branch> else <false_branch>
- probabilist branches: nchoice <weight_1, branch_1> <weight_2, branch_2>, ... <weight_n, branch_n>
  where weight_i is an integer representing the chance of executing branch_i (in other words, probability of executing branch_i equals weight_i divided by ∑ weight_j)
- preemptive condition: preemptive <condition, branch>
  program branch will exit as soon as condition is true (this condition is actually evaluated before executing each instruction of branch)

6.2.2. load profiles

Load profiles enables predefining how the population of each behavior will evolve, by setting the number of active instances according to time. A load profile is a sequence of lines or squares. For each load profile, a flag states if active instances shall be stopped to enforce a decrease of the population, or if the extra behaviors shall complete in a kind of a “lazy” approach.

6.2.3. ISAC plug-ins

A behavior can be understood as a logic definition, a kind of a skeleton. In order to actually generate traffic on the SUT, this skeleton must be associated to one or more ISAC plug-ins. Plug-ins are external Java libraries, that are responsible for:

- performing actions (i.e. generating requests) on the SUT, whose response times will be measured, using and managing specific protocols (e.g. HTTP, DNS, JDBC, TCP/IP, DHCP, SIP, LDAP or whatever);
- providing conditions used by the behaviors' conditional statements (if-then-else, while, preemptive);
- providing timers to implement delays (think time), for example with specific random distributions or computed in some arbitrary way;
- providing ad hoc controls for the plug-in itself (e.g. to change some settings);
- providing support for external data provisioning (e.g. a database of product references or a file containing identifier-password pairs for some user accounts), used as parameters by the behaviors.

6.2.4. Writing an ISAC scenario

ISAC scenarios are stored in and read from XML files, with extension ".xis" (standing for XML Isac Scenario). An ISAC scenario holds three main sections:

1. a section for plug-in imports, where default/initialization parameters can be set. A plug-in may be imported more than once if necessary: for each imported plug-in, each instance of each behavior will hold a sort of private context (called session object). Each imported plug-in is designated via a unique identifier.
2. a section for behaviors definition. All actions (aka samples), conditions (aka tests), controls and delays (aka timers) must refer to an imported plug-in using its identifier. For each call to the plug-in, specific parameter strings may be set. Those strings may hold variables: when the pattern `${plugin-identifier:key}` is found, it is replaced at runtime by a value that the designated plug-in associates with the provided key string. The designated plug-in must be a "data provider" type plug-in, and the interpretation of the key depends on it (refer to the documentation of the data provider plug-in).
3. an optional section for load profiles, with (at most) one profile per behavior.

The most user-friendly way to edit a scenario is to use the Eclipse-based ISAC graphical editor (see section 7). The alternative is to use an XML or text editor (the DTD of ISAC scenarios is given in appendix page 41).

6.2.5. Recording an ISAC scenario for Http

In order to make realistic scenarios corresponding to real users behaviors, session web can be recorded in ISAC scenario. It consists on using a proxy called MaxQ, available here: http://maxq.tigris.org/, which will capture user sessions.

To record an ISAC scenario:

1. You have to edit the maxq.properties file and to choose which timer will be used during the injection (ConstantTimer and RamdomTimer are available). You can also specify on which port starts MaxQ. By default, it starts on the port 8090.
2. You have to configure your web browser to go through a proxy for Http requests.
3. Then you have to click on "File" -> "New" -> "ISAC scenario". At this point, the proxy is started but doesn't record ISAC scenario yet : it works as a transparent proxy.

4. Click on "Test" -> "Start Recording". Now, all requests going from the web browser to a server will be stored in the ISAC scenario.
5. At the end of the web session, click on "Test" -> "Stop Recording". A pop-up appears to select a name and a destination to save the file. Give a name with the extension ".xis". Then save.

Now you have a scenario corresponding to a user behavior. You can import it in your Clif Console to edit the load profile in order to replay it on a large scale.

6.2.6. Deploying and executing an ISAC scenario

Remember that a scenario is local to each load injector. When editing your test plan, the key idea is to use the ISAC execution engine as a load injector, and to set the test plan file as argument:

- class name: `IsacRunner`
- arguments: `myScenario.xis`

Your code server path should include the directory where your scenario file is, in order to benefit from the automatic remote loading of the scenario file by every remote ISAC execution engine you may have defined in your test plan (see appendix page 50 for details).

A number of the execution engine's parameters may be modified, including at runtime:

- about the engine itself (size of the thread pool, polling period for load profile management, tolerance on deadlines);
- about the active scenario, in particular the number of active instances (population) of each behavior.

ISAC scenarios end on completion (load profiles time have elapsed), failure (abort), or manual stop. As soon as at least one behavior population has been manually set, or when no load profile is defined for any behavior, the scenario must be manually stopped.

6.3. Defining your own load injectors (Java programmers)

6.3.1. Using MTScenario utility class

`MTScenario` is an abstract Java class dedicated to programmers, although the `webtest` example, based on `MTScenario`, may be used by non-programmers for simple web testing. `MTScenario` makes it easy to define a test scenario as a set of concurrent threads ("sessions") looping on arbitrary actions, with an initial ramp-up time and during a given test duration. The programmer just has to define the session objects and actions.

6.3.2. Writing your own ISAC plug-ins

Principle

Writing your own ISAC plug-in is a simple way to customize the injection capabilities of ISAC, still relying on the generic language for defining behaviors and load profiles. Writing an ISAC plug-in basically consists in defining a Java class that encapsulates (a part of) the state of each behavior instance, and provides specific methods for:

- instantiating new *session objects* for new behavior instances;

- implementing load injection primitives;
- implementing timer primitives (e.g. to implement think times);
- implementing external data provisioning;
- implementing condition primitives;
- session object control primitives.

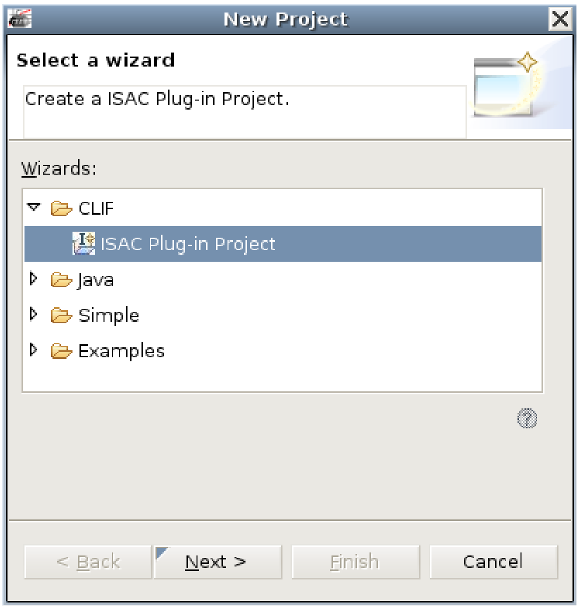
The primitives offered by an ISAC plug-in, as well as a GUI-oriented description for its parameters, are declared through 3 descriptor files:

- `plugin.properties` file specifies Java properties `plugin.name`, `plugin.xmlFile` and `plugin.guiFile` to respectively set the ISAC plug-in name, the name of the XML file describing the list of primitives and parameters, and the name of the XML file describing the GUI concerns. Usual values for these file names respectively are `plugin.xml` and `gui.xml`.
- `plugin.xml` file (or any other name as specified in `plugin.properties` file)
- `gui.xml` file (or any other name as specified in `plugin.properties` file)

To add a new ISAC plug-in, you must create a directory in subdirectory `isac/plugins` of CLIF execution environment. You may also create a local `build.xml` file that will be called by CLIF's main `build.xml` file (at the root of CLIF runtime environment) through targets `isac-plugins` and `isac-clean`, respectively for compiling and cleaning all ISAC plug-ins.

The ISAC plug-in creation Wizard for Eclipse

CLIF's Eclipse-based GUI comes with a wizard for creating ISAC plug-ins. It consists in creating a new ISAC plug-in project which combines a classical Eclipse Java project wizard with specific GUI pages dedicated to the declaration of ISAC primitives and parameters. The wizard generates the three descriptor files as well as a Java class skeleton accordingly to specific code design patterns. This skeleton is supposed to be completed with your specific code, preferably keeping the same design patterns if you want to keep an optimal support from the wizard when modifying your plug-in. In case of consistence troubles between the descriptor files and the Java code, the XML descriptors are regarded as the reference.



The following sections give some explanations about the construction of ISAC plug-ins.

XML descriptor files

The plug-in descriptor file specifies:

- the plug-in name, which must match the plug-in's directory name,
- the associated session object class and the initial settings parameters, with some help
- the samples, controls, conditions and timers with their parameters and help.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE plugin PUBLIC "-//objectweb.org//DTD CLIF Isac 1.0//EN"
"org/objectweb/clif/scenario/isac/dtd/plugin.dtd">
<plugin name="DnsInjector">
	<object class="org.objectweb.clif.isac.plugins.DnsInjector">
		<params>
			<param name="server_arg" type="String" />
		</params>
		<help>
This plugin sends UDP-based type A DNS queries to the specified server
		</help>
	</object>
	<sample name="query" number="0" >
		<params>
			<param name="name_arg" type="String" />
		</params>
		<help>
Resolves a name
		</help>
	</sample>
</plugin>
```

The user interface descriptor file adds explicit labels to primitives and parameters, and associates each parameter to GUI-related information. Possible graphical widgets are available through the following tags : `radiobutton`, `field`, `checkbox`, `nfield` (variable number of fields), `combo`. Parameters may also be visually grouped together with the `group` tag. The parameter value resulting from a `nfield` widget is the concatenation of the variable number of fields separated by one ';' character.

```
<gui>
	<object name="DnsInjector" >
		<params>
			<param name="server_arg" label="IP address or name of DNS server"
type="String" >
				<field/>
			</param>
		</params>
	</object>
	<sample name="query" number="0" label="query" >
		<params>
			<param name="name_arg" label="DNS name to resolve" type="String" >
				<field/>
			</param>
		</params>
	</sample>
</gui>
```

Session object instantiation

When writing an ISAC scenario, each imported plug-in will result in a session object associated to each behavior instance. If a plug-in is imported several times by a single scenario, each behavior instance will be associated to as many session objects as plug-in imports. For each import, different settings may be entered. So, for each import, the ISAC execution engine instantiates and initializes with these settings a specimen session object. For that purpose, your plug-in class must implement a public constructor taking a Map as a single argument. This map will hold the specimen settings with the parameters names as keys, as specified in the plug-in XML descriptor file. The specimen objects will be used just for replication, according to the load profiles, but will never be associated to behavior instances.

Then, your plug-in class must implement the `SessionObjectAction` interface to handle replication of specimens for creation of session objects that will be actually associated to behavior instances (method `createNewSessionObject()`). This interface is also used for freeing resources used by session objects before they are discarded (method `close()`), and recycling old session objects into fresh ones (method `reset()`).

```
public class MyPluginSessionObject implements
org.objectweb.clif.scenario.isac.util.SessionObjectAction {
        public MyPluginSessionObject(java.util.Map arguments) {...} // mandatory constructor for
session object specimen
        public Object () {...} // called on a specimen to instantiate a new session object and
return it
        public void reset() {...} // called on a used session object for recycling (i.e. turning it to a
fresh session object)
        public void close() {...} // called on a used session object for cleaning before being
discarded
...
```

Load injection primitives

Load injection primitives are declared in the XML plug-in descriptor using tag `sample`, and identifying each primitive with a unique integer value. All load injection primitives for a given plug-in are implemented by method `doSample(int, Map, ActionEvent)`, as specified by interface `SampleAction`.

- The first argument gives the primitive identifier;
- the second parameter gives the list of parameter values indexed by their names, as set in the plug-in descriptor file using tag `params`;
- the third argument gives a report object whose fields will have to be filled before being returned.

Basically, the `doSample()` method is supposed to perform a load injection request, wait for some kind of response, state if this request is a success or a failure, measure its response time and return a sample report. Returning null is also possible, to make CLIF ignore this sample.

```
public class MyPluginSessionObject implements
org.objectweb.clif.scenario.isac.plugin.SessionObjectAction,
org.objectweb.clif.scenario.isac.plugin.SampleAction {
        public ActionEvent doSample(int number, Map params, ActionEvent report) {
                switch (number)
...
```

Timer primitives

Timer primitives are declared in the XML plug-in descriptor using tag `timer`, and identifying each primitive with a unique integer value. All timer primitives for a given plug-in are implemented by method `doTimer(int, Map)`, as specified by interface `TimerAction`.

- The first argument gives the primitive identifier;

- the second parameter gives the list of parameter values indexed by their names, as set in the plug-in descriptor file using tag `params`.

The `doTimer()` method must return a number of milliseconds that will be taken into account by the execution engine to make the calling behavior instance sleep. This method shall not perform a sleep period by itself!

```
public class MyPluginSessionObject implements
org.objectweb.clif.scenario.isac.plugin.SessionObjectAction,
org.objectweb.clif.scenario.isac.plugin.TimerAction {
	public ActionEvent doTimer(int number, Map params) {
		switch (number)
...
```

Condition primitives

Condition primitives are used by the conditional constructs of behaviors (while, if, preemption).Condition primitives are declared in the XML plug-in descriptor using tag `test`, and identifying each primitive with a unique integer value. All condition primitives for a given plug-in are implemented by method `doTest(int, Map)`, as specified by interface `TestAction`.

- The first argument gives the primitive identifier;
- the second parameter gives the list of parameter values indexed by their names, as set in the plug-in descriptor file using tag `params`.

The `doTest()` method must return a boolean according to whether the condition is met or not.

```
public class MyPluginSessionObject implements
org.objectweb.clif.scenario.isac.util.SessionObjectAction,
org.objectweb.clif.scenario.isac.plugin.TestAction {
	public ActionEvent doTest(int number, Map params) {
		switch (number)
...
```

Control primitives

Control primitives are used to perform an arbitrary control action on a session object (e.g. increment a counter session object). Control primitives are declared in the XML plug-in descriptor using tag `test`, and identifying each primitive with a unique integer value. All condition primitives for a given plug-in are implemented by method `doControl(int, Map)`, as specified by interface `ControlAction`.

- The first argument gives the primitive identifier;
- the second parameter gives the list of parameter values indexed by their names, as set in the plug-in descriptor file using tag `params`.

The `doControl()` method just performs the control action and returns.

```
public class MyPluginSessionObject implements
org.objectweb.clif.scenario.isac.util.SessionObjectAction,
org.objectweb.clif.scenario.isac.plugin.ControlAction {
	public ActionEvent doControl(int number, Map params) {
		switch (number)
...
```

External data provisioning

All parameters set in an Isac scenario may contain an external data reference, through an expression of this form : `${dataProviderIdentifier:reference}`. At runtime, this expression will be replaced by the String returned by the `doGet(reference)` call on the plug-in session object identified by `dataProviderIdentifier`. The format of reference is unspecified and typically depends on the data provider plug-in implementation.

To implement a data provider plug-in, just implement interface `DataProvider` and the corresponding `doGet(String)` method. You are free to interpret the string argument and return any String (computed or picked up from any source).

Note that the XML plug-in descriptor does not declare the data provisioning capability. On the other hand, this capability is not checked and the outcome of trying to get data from a plug-in that does not implement the `DataProvider` interface is unspecified.

```
public class MyPluginSessionObject implements
org.objectweb.clif.scenario.isac.util.SessionObjectAction,
org.objectweb.clif.scenario.isac.plugin.DataProvider {
	public ActionEvent doGet(String reference) {
...
```

6.3.3. Implementing a Blade component

[TODO]

7. Eclipse-based graphical user interface

7.1. Introduction

CLIF comes with an Eclipse-based Graphical User Interface. This GUI has 3 functions:

- a CLIF console for test deployment, execution and monitoring, including a test plan editor;
- a graphical editor for ISAC scenarios;
- a programming environment for ISAC plug-ins.

All parts are available as separate Eclipse plug-ins, or delivered as a single ready-to-use standalone program. Note that CLIF's runtime environment directory, as often referred to in this documentation, is located in the console plug-in path, i.e. something like `plugins/org.objectweb.clif.console.plugin_x.x.x/`

The console GUI uses the same project-and-files pattern as Eclipse. You must begin by creating a new project (using the project wizard and choosing the "Simple" project). Then you can use the "New ClifTestPlan" wizard to create one or several test plans in your load test project. Similarly, you may also use the "New Isac Scenario" wizard to create ISAC scenarios. To create a new ISAC plug-in, create a new project using the “New ISAC plug-in” wizard.

Please refer to the on-line help for detailed documentation about these parts.

As an Eclipse applications, a number a useful options may be set on the command line, such as:

- `-consoleLog` to see messages printed out to your terminal;
- `-vm /path/to/the/jvm` to set the right Java Virtual Machine to be used;
- `-data /path/to/my/workspace` to use a different workspace directory from the default one.

Should you require to set some specific system properties, please edit file `clif.props.template` in directory `plugins/org.objectweb.clif.console.plugin_x.x.x/etc`.

7.2. Getting started

7.2.1. From an Eclipse-RCP based standalone CLIF distribution

Eclipse RCP based binary distributions of CLIF are full-fledged standalone executables that include Eclipse RCP environment for a specific operating system. They require a JDK, and most often Apache ant utility to run CLIF servers (see requirements in section 3.1).

You just have to get the right binary distribution for your operating system (see section 3.2), or generate it from the source (see section 3.3). Then, unzip it wherever you want on your computer, avoiding path names with whitespace characters. Finally, run `clif-console` program with any useful argument (as detailed above).

7.2.2. From the Eclipse plug-ins

CLIF provides 2 Eclipse plug-ins, namely the clif.console plug-in and the clif.isac plug-in. The clif.isac Eclipse plug-in contains the Isac scenario editor and the Isac plug-in creation wizard. The clif.console Eclipse plug-in contains all the remaining parts of CLIF (the test plan editor, the supervision console, the analysis tools).

The clif.isac Eclipse plug-in depends on the clif.console Eclipse plug-in.

Dependencies

CLIF Eclipse plug-ins depends on a number of Eclipse plug-ins that may not be present by default in your Eclipse Workbench installation. We recommend to download the Eclipse-based CLIF console (whatever the target operating system) and get the missing Eclipse plug-ins from the `plugins` subdirectory. The list of necessary Eclipse plug-ins is given below:

- Directories:
  - `org.apache.xerces_2.7.0`
  - `org.eclipse.jem.util_1.1.0`
- Jar files:
  - `org.eclipse.emf.common_2.1.0`
  - `org.eclipse.emf.ecore.edit_2.1.0`
  - `org.eclipse.emf.ecore.xml_2.1.0`
  - `org.eclipse.emf.ecore_2.1.0`
  - `org.eclipse.emf.edit_2.1.0`
  - `org.eclipse.wst.common.emf_1.0.0`
  - `org.eclipse.wst.common.emfworkbench.integration_1.0.0`
  - `org.eclipse.wst.common.environment_1.0.0`
  - `org.eclipse.wst.common.project.facet.core_1.0.0`
  - `org.eclipse.wst.common.ui_1.0.0`
  - `org.eclipse.wst.common.uriresolver_1.0.0`
  - `org.eclipse.wst.common.frameworks_1.0.0`
  - `org.eclipse.wst.dtd.core_1.0.0`
  - `org.eclipse.wst.sse.core_1.0.0`
  - `org.eclipse.wst.sse.ui_1.0.0`
  - `org.eclipse.wst.validation_1.0.0`
  - `org.eclipse.wst.xml.core_1.0.0`
  - `org.eclipse.wst.xml.ui_1.0.0`
  - `org.eclipse.xsd_2.1.0`

Installation

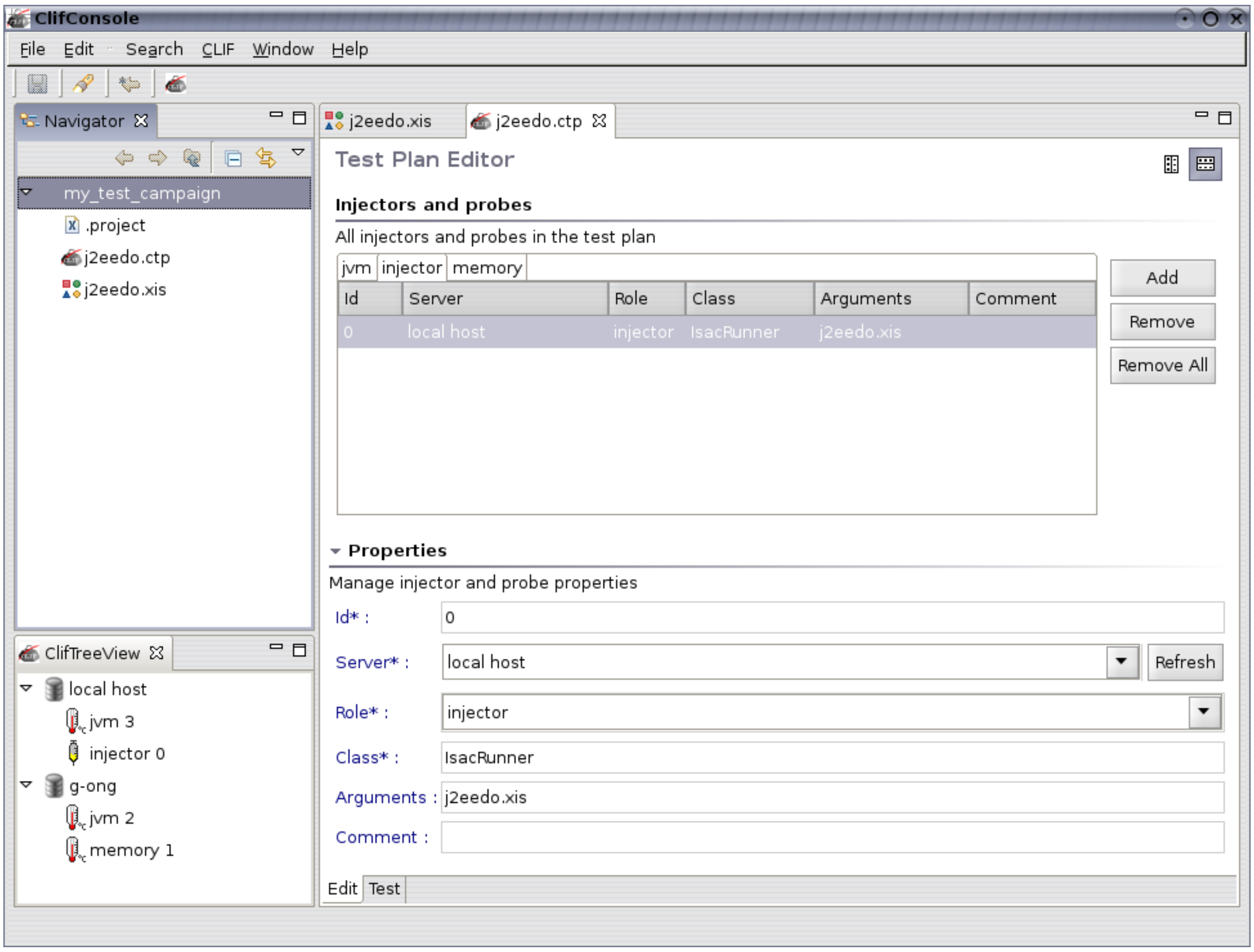
Up to version 1.2.1, CLIF plug-ins rely on Eclipse 3.1, while starting from version 1.2.2, they rely on Eclipse 3.2. You may simply copy all the Eclipse plug-ins in the `plugins` directory of your Eclipse installation. Make sure you get the right CLIF version with regard to your Eclipse version.

Take care to set write permissions for the clif.console Eclipse plug-in directory. Although this is not really satisfactory and it is going to be fixed, you would not be able to run tests without being granted write permissions.

Execution

Remember to run Eclipse with the right JVM, as mentioned in section 3.1. Use option `-vm` to make sure Eclipse uses the right JVM. See section 7.1 for the most important options you may set when launching your Eclipse workbench.

7.3. Test plan edition



Test Plan Editor

Injectors and probes

All injectors and probes in the test plan

jvm | injector | memory

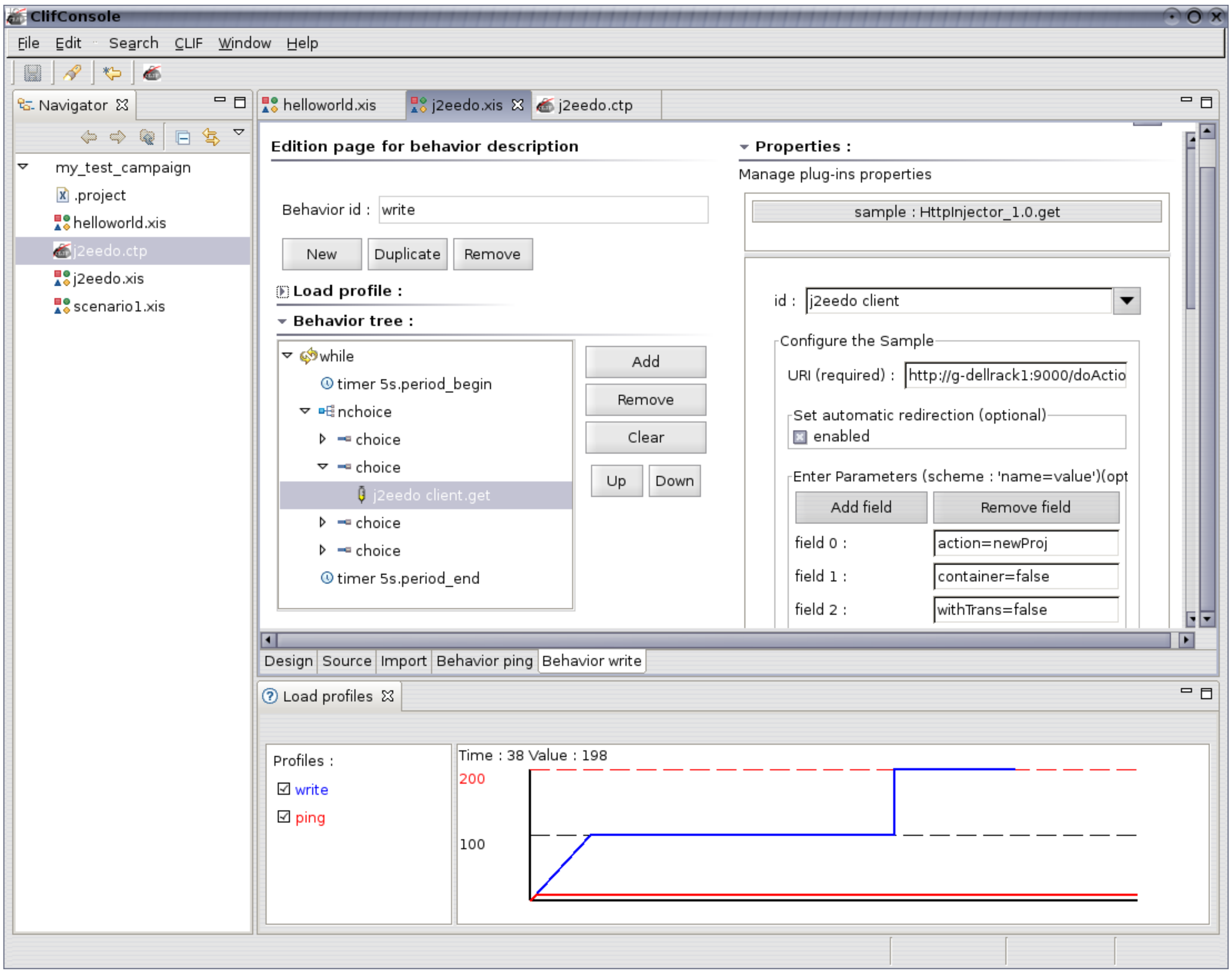
Id	Server	Role	Class	Arguments	Comment
0	local host	injector	IsacRunner	j2eedo.xis	

Properties

Manage injector and probe properties

Id* :	0
Server* :	local host
Role* :	injector
Class* :	IsacRunner
Arguments :	j2eedo.xis
Comment :	

7.4. ISAC scenario edition



ClifConsole

File Edit Search CLIF Window Help

Navigator

my_test_campaign
- .project
- helloworld.xis
- j2eedo.ctp
- j2eedo.xis
- scenario1.xis

helloworld.xis | j2eedo.xis | j2eedo.ctp

Edition page for behavior description

Behavior id : write

New | Duplicate | Remove

Load profile :

Behavior tree :

```
while
   timer 5s.period_begin
   nchoice
      choice
      choice
         j2eedo client.get
      choice
      choice
   timer 5s.period_end
```

Add | Remove | Clear | Up | Down

Properties :

Manage plug-ins properties

sample : HttpInjector_1.0.get

id : j2eedo client

Configure the Sample

URI (required) : http://g-dellrack1:9000/doActio

Set automatic redirection (optional)

☒ enabled

Enter Parameters (scheme : 'name=value')(opt

Add field | Remove field

field 0 : action=newProj

field 1 : container=false

field 2 : withTrans=false

Design | Source | Import | Behavior ping | Behavior write

Load profiles

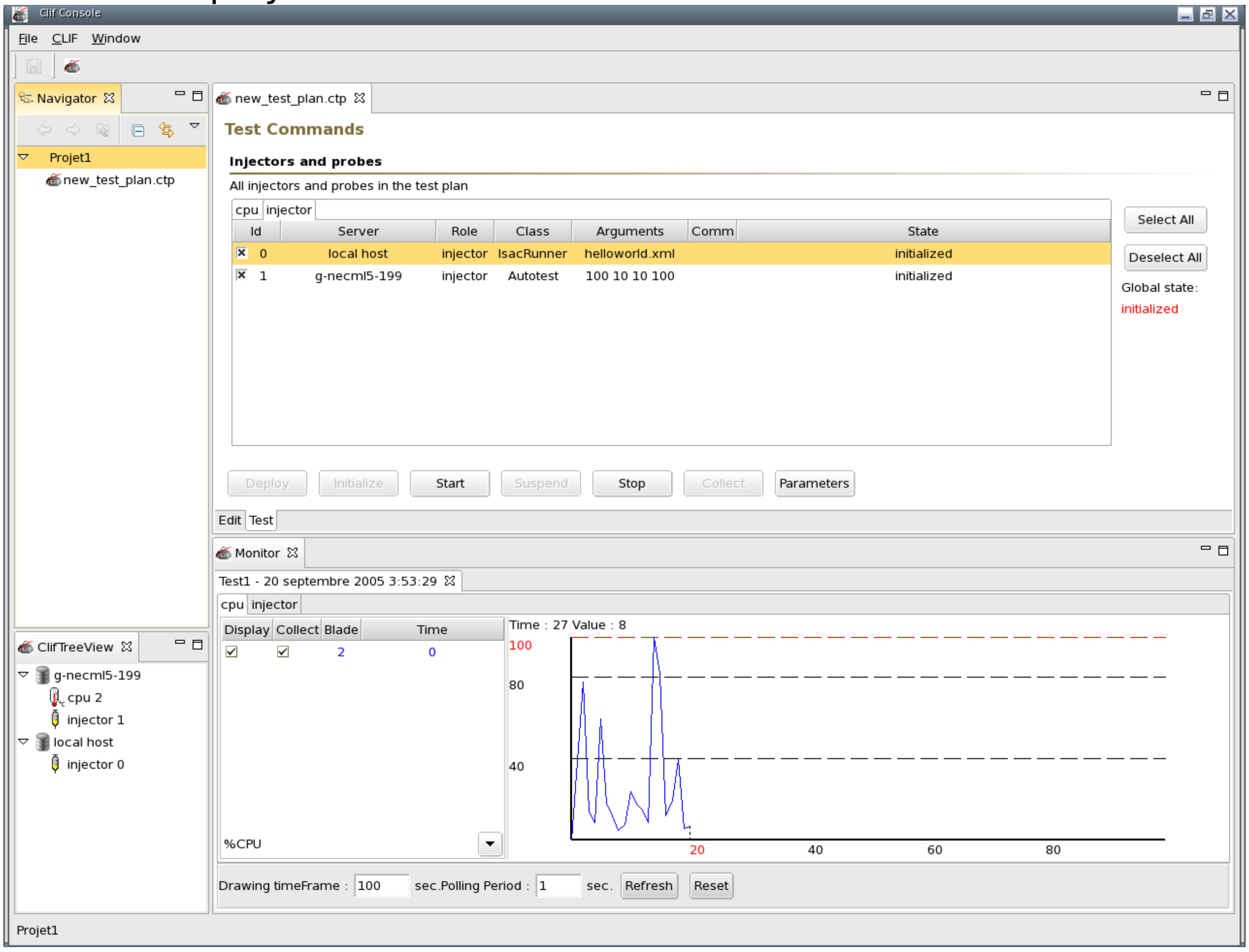
Profiles :

☑ write

☑ ping

Time : 38 Value : 198

7.5. test deployment and execution



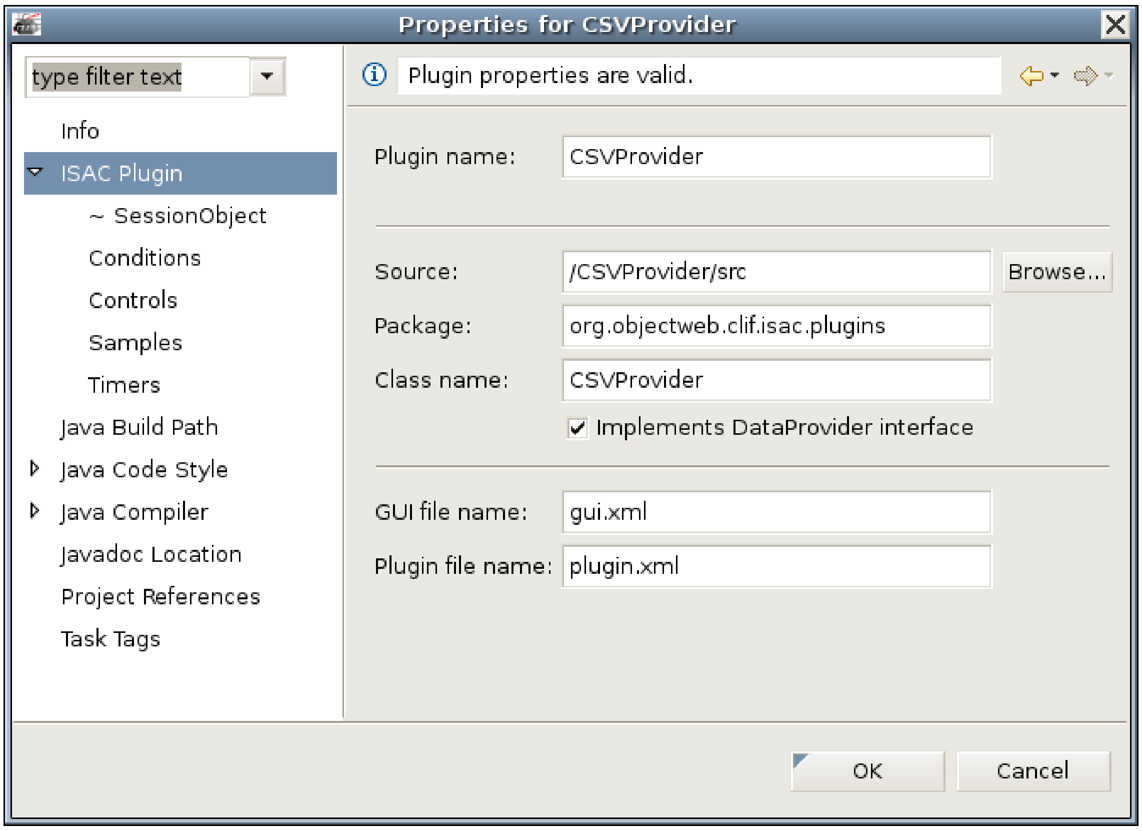
Id	Server	Role	Class	Arguments	Comm	State
0	local host	injector	IsacRunner	helloworld.xml		initialized
1	g-necml5-199	injector	Autotest	100 10 10 100		initialized

Global state: initialized

Display	Collect	Blade	Time
✓	✓	2	0

Drawing timeFrame : 100 sec.Polling Period : 1 sec.

7.6. ISAC plug-in creation Wizard



Plugin properties are valid.

Plugin name: CSVProvider

Source: /CSVProvider/src

Package: org.objectweb.clif.isac.plugins

Class name: CSVProvider

[✓] Implements DataProvider interface

GUI file name: gui.xml

Plugin file name: plugin.xml

8. Java Swing-based graphical user interface

8.1. Introduction

CLIF comes with a Java/Swing-based Graphical User Interface. This GUI consists of a console for test deployment, execution and monitoring, including a test plan editor. It also provides an analysis tool to help produce test reports.

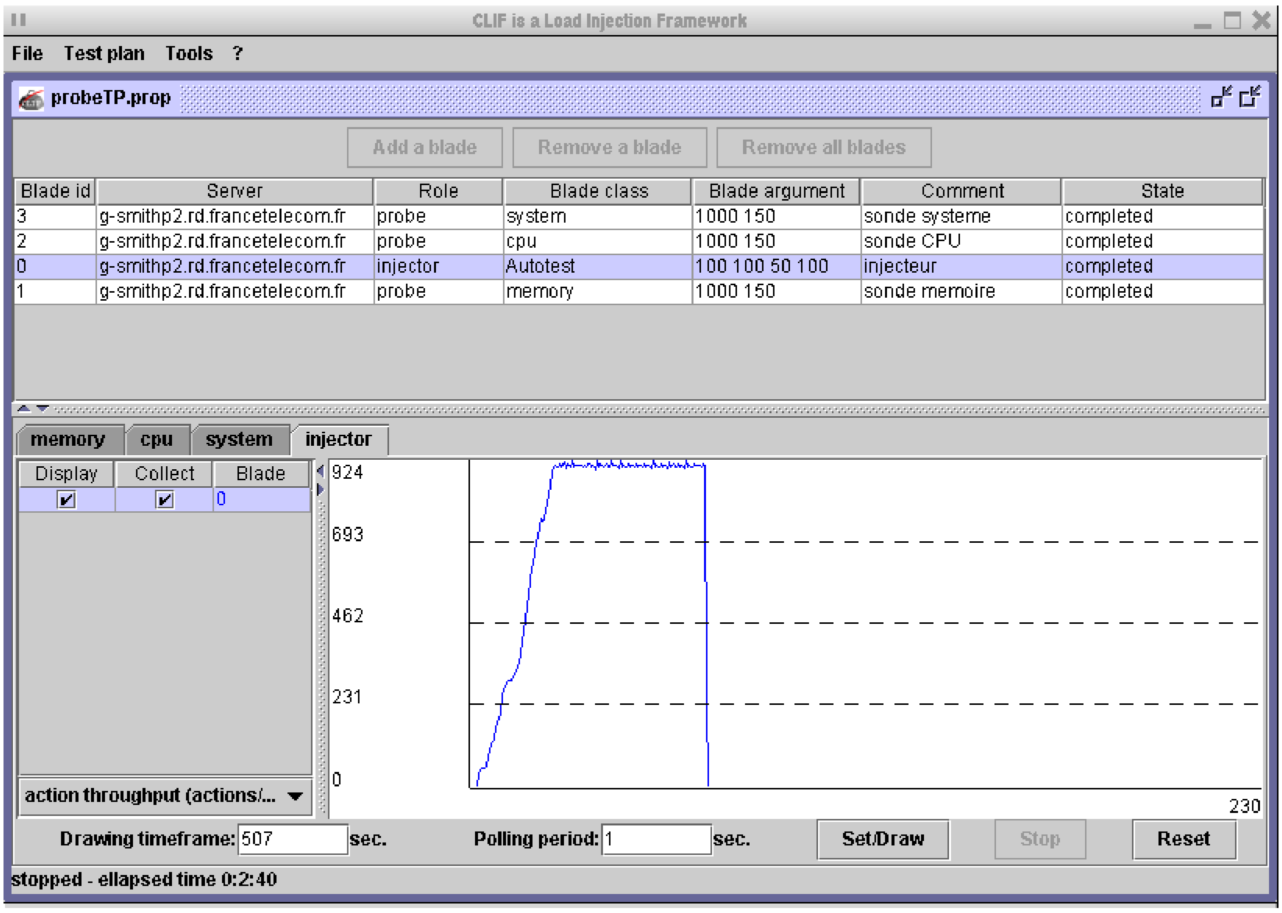
Compared to the Eclipse RCP-based console (see section 7), the Swing-based console has the advantage of light-weight, simplicity and operating-system independence. On the negative side, its simplicity springs from a reduced set of features. In particular, it does not provide an ISAC scenario editor nor an ISAC plug-ins creation wizard. As far as the test results analysis is concerned, the consoles provide different tools that suit different needs. The one provided by the Swing console is probably more straightforward to use, and rapidly gives graphical views, while the one provided by the Eclipse console is suited to the creation of long reports based on well-structured report templates. Of course, once a test has been run, any analysis tool may be used regardlessly of the user interface that has been used to run the test.

Note that the Swing console is actually embedded in the CLIF Eclipse-RCP distribution, since it provides the so-called CLIF runtime environment directory, located in the console plug-in path, i.e. something like `plugins/org.objectweb.clif.console.plugin_x.x.x/`.

8.2. Getting started

To run the CLIF Swing console, the mandatory requirement is the right JDK and preferably the Apache ant utility (see requirements in section 3.1).

You just have to get the right CLIF distribution (see section 3.2), or generate it from the source (see section 3.3). Then, unzip it wherever you want on your computer, avoiding path names with whitespace characters. To run the console, use command `ant console` in the root directory of the unzipped distribution.



Blade id	Server	Role	Blade class	Blade argument	Comment	State
3	g-smithp2.rd.francetelecom.fr	probe	system	1000 150	sonde systeme	completed
2	g-smithp2.rd.francetelecom.fr	probe	cpu	1000 150	sonde CPU	completed
0	g-smithp2.rd.francetelecom.fr	injector	Autotest	100 100 50 100	injecteur	completed
1	g-smithp2.rd.francetelecom.fr	probe	memory	1000 150	sonde memoire	completed

Display	Collect	Blade
✓	✓	0

action throughput (actions/...

Drawing timeframe: 507 sec. Polling period: 1 sec.

stopped - ellapsed time 0:2:40

8.3. Test plan edition table

A test plan defines the probes and the injectors to be used, with their parameters, and where to deploy them. Remember that injectors and probes are uniformly designated as "blades". The table in the upper part is the test plan editor. Note that the bottom part (monitoring) is hidden as long as the test is not initialized. Note also that the test plan is not editable when the monitoring area is shown.

Each row of the test plan table defines a blade configuration, through 6 columns:

- **Blade id** is a unique identifier for the injector or probe to be deployed. A default id is automatically set when adding a new blade, but it may be freely changed by the user as long as it remains unique within current test plan;
- **Server** offers a choice between available CLIF servers, where the blade is to be deployed. The list of CLIF servers may be updated using option "Window > Refresh server list";
- **Role** specifies whether the blade is a probe or an injector;
- **Blade class** is where the user sets:
  - either the Java class to be instantiated as a load injector (fully qualified name, without trailing .class extension - see section 6),
  - or a family name in case of a probe (see section 5);
- **Blade argument** is an argument line that will be passed to the new blade instance at deployment time;
- **Comment** is an arbitrary user comment line.

The last column **State** is not editable. It shows state information about the blade (undeployed, deploying, deployed, starting, running, stopping, suspending, resuming, completed, aborted...).

Test plans may be saved and restored using options in the File menu.

8.4. Performance and resource usage monitoring

As soon as the test plan is deployed and initialized, the monitoring area pops up in the test plan window's bottom part. This area holds a set of tabbed panels:

- one for all injectors
- one for each probe family

For each panel, the user may set the monitoring timeframe, the polling period, and start or stop the monitoring process. Moreover, a checkbox table at the left side of each panel makes it possible to selectively disable or enable the collect and display of monitoring data, for each blade.

8.5. File Menu

From this menu, the user can find options for saving and loading a test plan.

This menu also holds the "Quit" option to exit from CLIF console, which also terminates the registry where CLIF servers are registered. As a result, whenever you terminate a CLIF console, any remaining CLIF server will then become unreachable - you may stop these unreachable CLIF servers manually. Running the CLIF console again will create a new, empty registry, and then you may launch new CLIF servers. The user may not quit the console while a test is running (other wise, the behavior is undefined).

8.6. Test plan menu

This menu holds test deployment and control commands. There are 2 subsets of options:

- the first set holds test plan definition and deployment commands
  - option **Refresh server list** updates the list of available CLIF servers,
  - option **Edit** switches to test plan edition mode, when enabled (i.e. when not already in edition mode, and when no deployed test is currently running),
  - option **Deploy** deploys the probes and injectors defined by current test plan
- the second set holds test control commands
  - command **initialize** initializes all the blades so that they are actually ready to start;
  - commands **start**, **suspend**, **resume** and **stop** respectively start, suspend, resume and stop the execution of all blades;
  - command **collect** tells the storage system to collect all test data from the blades (the actual effect of this command fully depends on the Storage component). This option may be used only after a test run. Collecting more than once after a test run has no effect; collecting is not mandatory, which means that the user may not collect data if s/he is not interested in the test results.

8.7. Tools menu

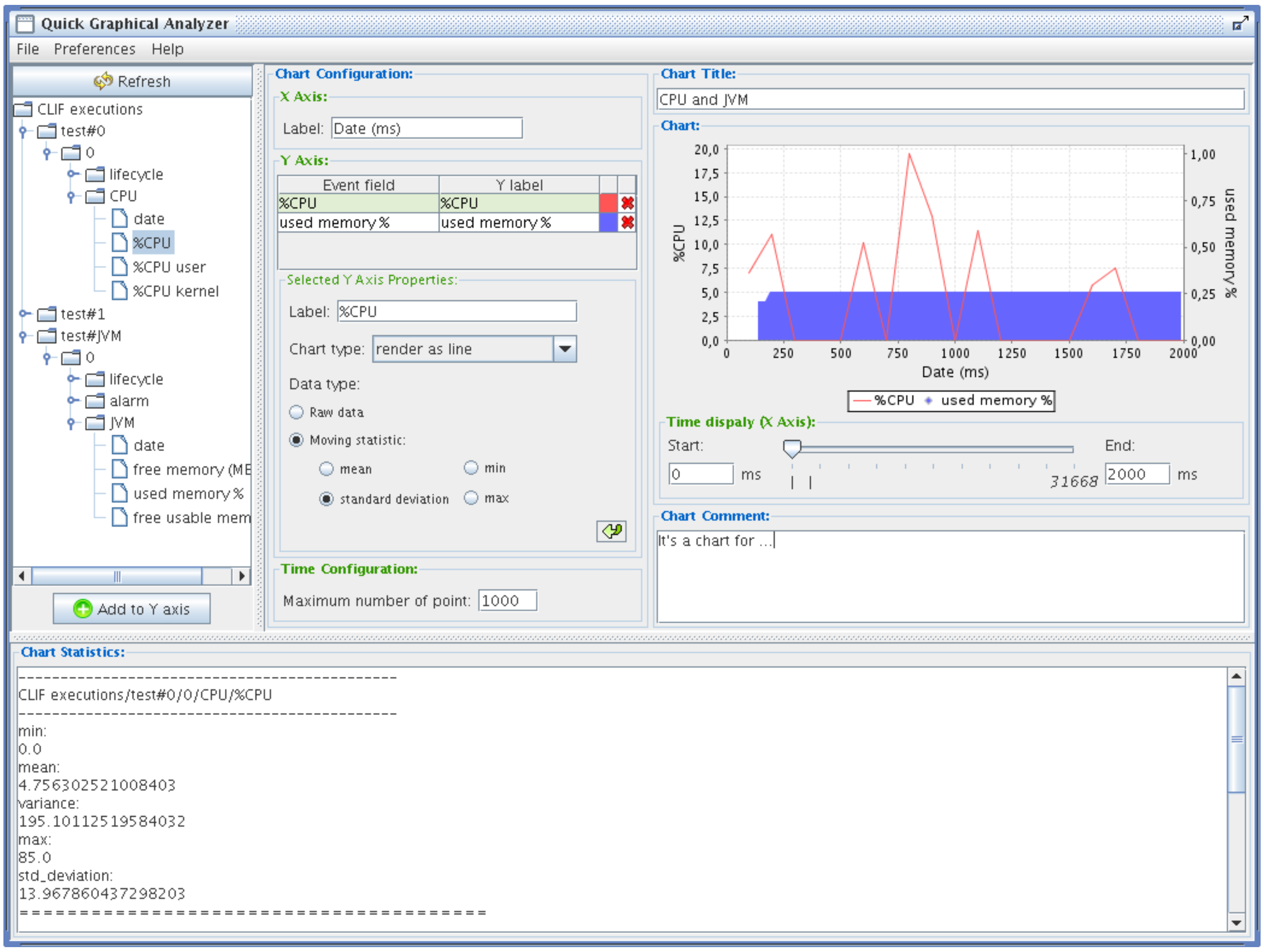
This menu displays on/off additional tools:

8.7.1. Basic analyzer

Basic analyzer tool provides an analysis tool/sample of test results (after test run) - this is just a preview.

8.7.2. Quick graphical analyzer

Graphical analyzer tool provides functions to analyze quickly test results after test run.



X Axis: Label: Date (ms)

Y Axis:

Event field	Y label		
%CPU	%CPU	🟥	✖
used memory %	used memory %	🟦	✖

Selected Y Axis Properties:
Label: %CPU
Chart type: render as line
Data type: ○ Raw data [✓ Moving statistic:] ○ mean ○ min [✓ standard deviation] ○ max

Time Configuration: Maximum number of point: 1000

Chart Title: CPU and JVM

Time dispaly (X Axis): Start: 0 ms End: 2000 ms 31668

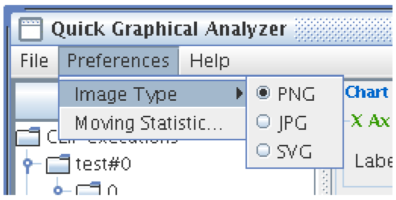
Chart Comment: It's a chart for ...

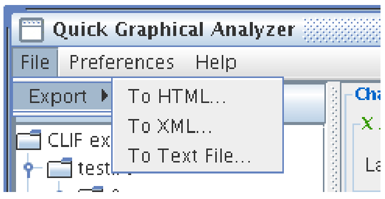
Chart Statistics:

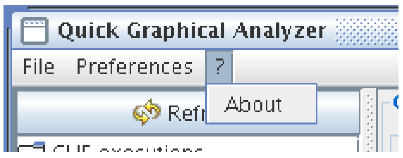
```
---------------------------------------------
CLIF executions/test#0/0/CPU/%CPU
---------------------------------------------
min:
0.0
mean:
4.756302521008403
variance:
195.10112519584032
max:
85.0
std_deviation:
13.967860437298203
=========================================
```

Menu:

By the file menu, the user can export his analyze in various formats (Text, XML or HTML).

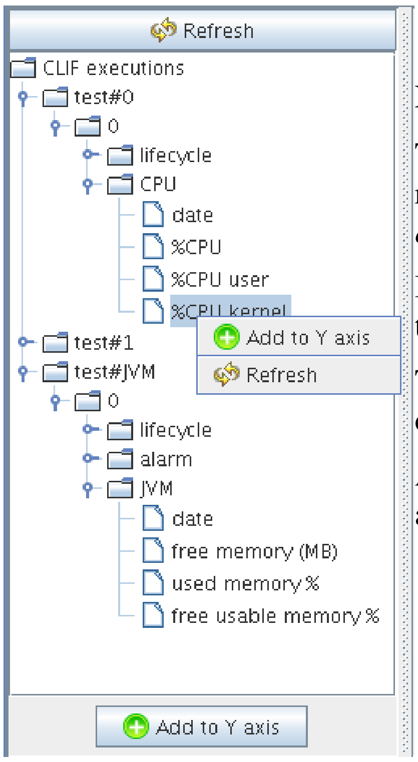






The Preferences menu contains export options and moving statistics options.

The help menu holds a single About entry which displays informations about the graphical analyzer.



Execution tree:

The test execution tree lists the available tests under a tree representation with the following hierarchy : *test / blade / event / event field*

If a test execution doesn't appear in the hierarchy the user can press the “*Refresh*” button to update the tree.

The “A*dd to Y axis*” button add the value of the selected leaf to the chart. The user can also add it by doing a double click on the leaf

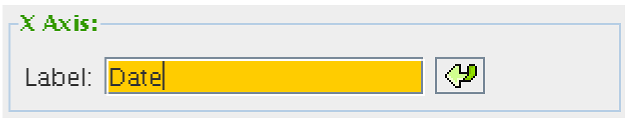
A double click on a leaf do the same action. The user can do this action by doing a “*right click*” in the tree.

Chart configuration:

The user can configure the chart using the "*chart configuration*" frame. In this frame he can define the X axis label, modify the curve, and define the maximum number of point to display.

You can configure the curve by modifying her label, her color, her appearance (line, dot, area or bar), and the type of data to display (raw or moving statistic).

After modifying a value, this values is colored in yellow and to apply the modification you validate them by clicking on the button.


X Axis:

Label: Date

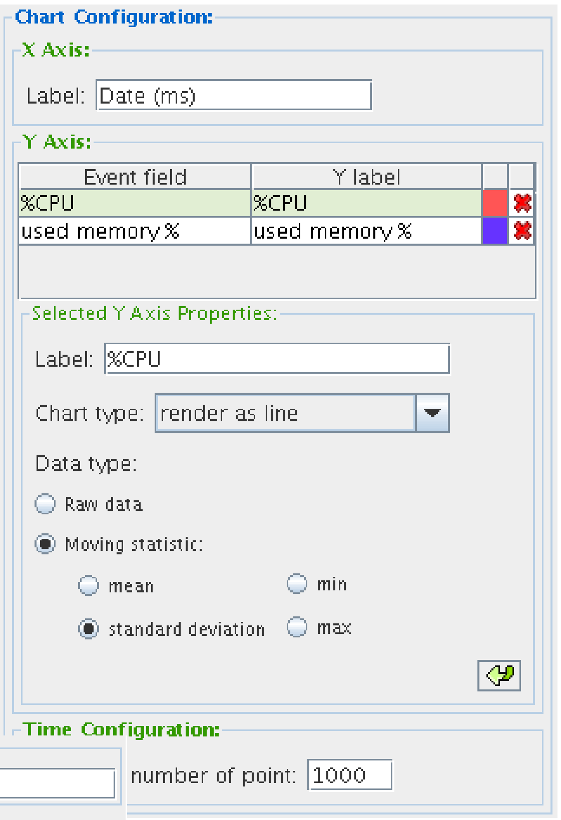


Chart Configuration:

X Axis:

Label: Date (ms)

Y Axis:

Event field	Y label		
%CPU	%CPU	🟥	✖
used memory %	used memory %	🟦	✖

Selected Y Axis Properties:

Label: %CPU

Chart type: render as line

Data type:

Raw data [✓ Moving statistic:]

mean, min, [✓ standard deviation], max

Time Configuration:

number of point: 1000

To remove an event field from the chart select the event in the table and click on the red cross.

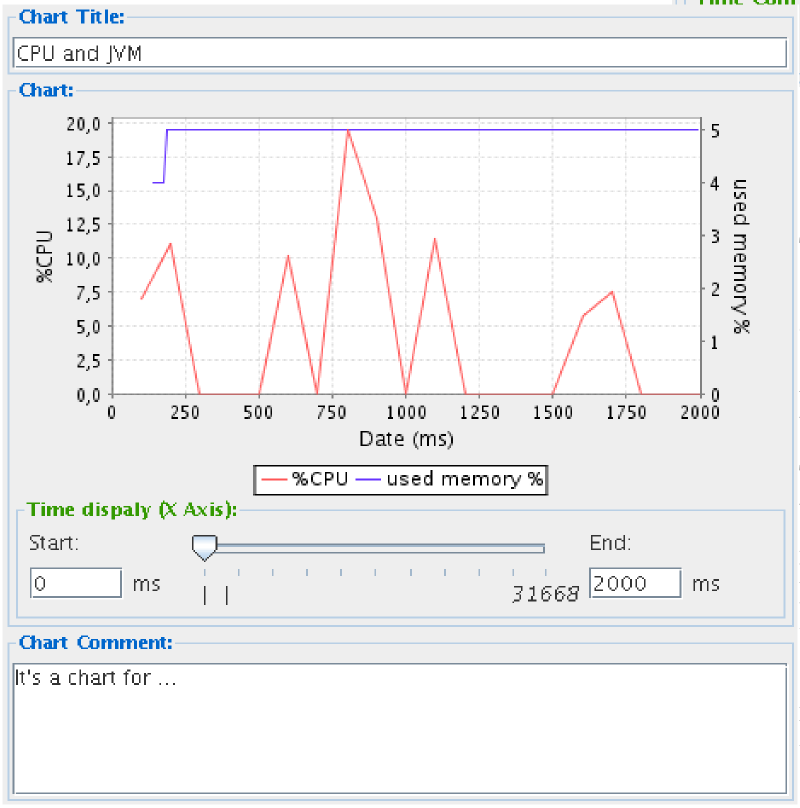


Chart Title:

CPU and JVM

Chart:

Time dispaly (X Axis):

Start: 0 ms

End: 2000 ms

31668

Chart Comment:

It's a chart for ...

Chart panel:

The chart panel displays the chart generate using the chart configuration.

At the top of the panel you can add a title to the chart.

There is a time line under the chart. Using the cursor, the user can modify the time window to display. He can also modify this time window by modifying the start and stop values. To validate the modifications he should click on the validation button.

At the bottom of the frame, a text area allows the user to comment the chart.

Statistics frame:

The Statistics frame display statistics about each event field for the displayed time window.

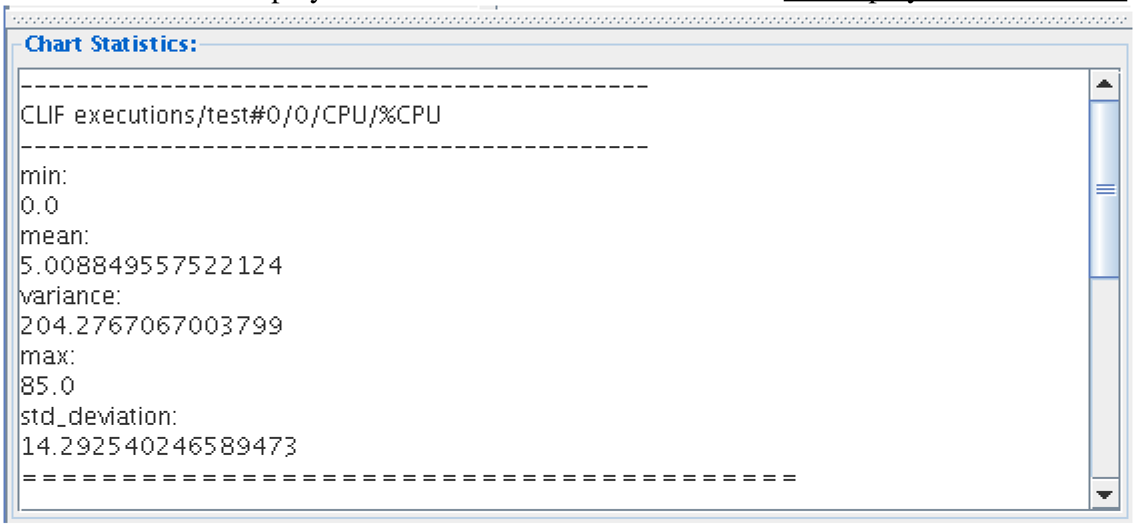


Chart Statistics:

```
--------------------------------------------
CLIF executions/test#0/0/CPU/%CPU
--------------------------------------------
min:
0.0
mean:
5.008849557522124
variance:
204.2767067003799
max:
85.0
std_deviation:
14.292540246589473
========================================
```

Export the chart :

The chart export create various file in accordance with the type of export.

Export as text:

When the user export the chart as text three files are created : a picture, a text file that contains the comments and one that contains statistics.

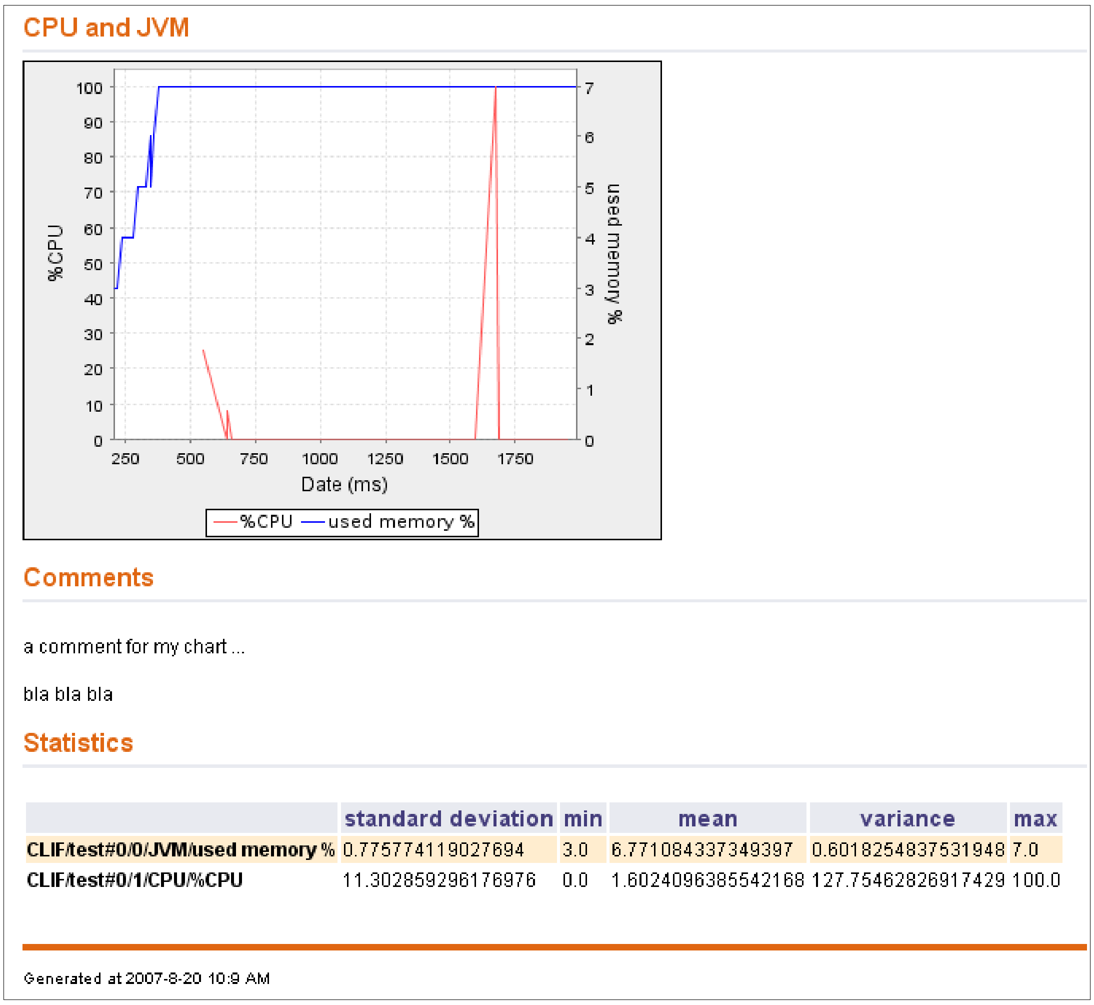
Export as XML:

If the user choose to export as XML, two files are create the picture of the chart and an XML file with the following Document Type Definition :

```
<!ELEMENT chart (image, comments, statistics, generation)>
<!ATTLIST chart     title CDATA #REQUIRED>
<!ELEMENT image >
<!ATTLIST image     file CDATA #REQUIRED>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT statistics (serie)>
<!ELEMENT serie (measure)>
<!ATTLIST serie     name CDATA #REQUIRED >
<!ELEMENT measure >
<!ATTLIST measure   name CDATA #REQUIRED
                    value CDATA #REQUIRED>
<!ELEMENT generation (#PCDATA)>
```

Export as HTML:

At last if the user choose the HTML format, an HTML file and a folder are created



Comments

a comment for my chart ...

bla bla bla

Statistics

	standard deviation	min	mean	variance	max
CLIF/test#0/0/JVM/used memory %	0.775774119027694	3.0	6.771084337349397	0.6018254837531948	7.0
CLIF/test#0/1/CPU/%CPU	11.302859296176976	0.0	1.6024096385542168	127.75462826917429	100.0

Generated at 2007-8-20 10:9 AM

8.8. ? (help) menu

This menu holds a single "**About...**" option, which displays CLIF version and compilation information. This information is important to get and mention whenever you report a problem using CLIF.

9. Command line user interface

9.1. Introduction

Once you have created a test plan file (either using the Eclipse-based or the Java Swing-based GUI, or editing a text file with the appropriate syntax), you may deploy and run tests using the following commands. Those commands are packaged as Apache ant targets defined in the build.xml file available at CLIF runtime environment's root.

Prior to any command, one Registry must be run for the whole test. It will be used by every command to register or lookup the components of the deployed test plan (aka CLIF application): injectors, probes, supervisor, storage.

Most of these commands apply either to every probe and injectors from a deployed test plan, or to a subset of them. To do this, you must specify an extra argument to give the list of the target injectors and probes identifiers (so-called blade identifier, as defined in the test plan): -Dblades .id=id1:id2:...idn. Note that separately managing probes and injectors can become tricky in big test plans... A typical usage of CLIF may not need this feature, and would only make use of the commands' default global scope.

Note that authorized commands depend on the state of the injectors and probes. Refer to appendix page 46 for details about the blade life-cycle.

9.2. Run CLIF Registry

```
ant registry
```

Runs a Registry on the local host. All CLIF servers that will be involved in the test plan the user is planning to deploy must then be launched with the right configuration. See sections 3 and 4 for details. Only one Registry shall me launched on a given host (further attempts will just fail).

9.3. Test plan deployment: deploy

```
ant -Dtestplan.name=name -Dtestplan.file=myTestPlan.ctp deploy
```

Deploys a new test plan (probes and injectors) as defined by a given test plan file. This deployed test plan is given a name that is further required for all others commands. When successful, this command does not return, and should not be manually terminated as long as you want to use the deployed test plan. The resulting process' role is similar to a (graphical) console's role, in that it contains the Supervisor and Storage components, as well as the code server.

9.4. Test initialization: init

```
ant -Dtestplan.name=name -Dtestrun.id=testId [-Dblades.id=id1:id2:...idn] init
```

Initializes all probes and injectors in a deployed test plan, or just a subset of them when mentioned. The target deployed test plan is designated by its name (as set with deploy command). An identifier for this new test being initialized must be provided. This identifier will only be used to identify this test run, for instance when accessing to results.

9.5. Test execution start: start

ant -Dtestplan.name=name [-Dblades.id=id1:id2:...idn] start

Starts probes and injectors of the given deployed test plan, or just a subset of them when mentioned. They must be initialized prior to this command.

9.6. Suspend test execution: suspend

ant -Dtestplan.name=name [-Dblades.id=id1:id2:...idn] suspend

Suspends probes and injectors of the given deployed test plan, or just a subset of them when mentioned. They must be running (started or resumed) prior to this command.

9.7. Resume test execution: resume

ant -Dtestplan.name=name [-Dblades.id=id1:id2:...idn] resume

Resumes probes and injectors of the given deployed test plan, or just a subset of them when mentioned. They must be suspended prior to this command.

9.8. Stop test execution: stop

ant -Dtestplan.name=name [-Dblades.id=id1:id2:...idn] stop

Definitively and immediately (in a best effort sense) stops probes and injectors of the given deployed test plan, or just a subset of them when mentioned. Stopping is possible for both running and suspended probes/injectors, as well as right after initialization. Don't forget to use the collect command to gather all measurements to the local site. Once a test is stopped, the same deployed test plan may be initialized again to run another test.

9.9. Wait for a test execution to terminate: join

ant -Dtestplan.name=name [-Dblades.id=id1:id2:...idn] join

Waits until the probes and injectors of the given deployed test plan, or just a subset of them when mentioned, terminate. Probes and injectors should be running to prevent this command from blocking forever.

9.10. Collect test results (measurements): collect

ant -Dtestplan.name=name [-Dblades.id=id1:id2:...idn] collect

Collects results generated by the probes and injectors of the given deployed test plan, or just a subset of them when mentioned. Collecting is optional, i.e. the user may not collect results s/he is not interested in. Injectors and probes must be terminated prior to this command.

9.11. Shortcut for full test execution process: run

ant -Dtestplan.name=name -Dtestrun.id=testId [-Dblades.id=id1:...idn] run

Shortcut for init, start, join and collect on the probes and injectors of the given deployed test plan, or just a subset of them when mentioned.

9.12. Shortcut for full deployment and execution process: launch

ant -Dtestplan.name=name -Dtestrun.id=testId -Dtestplan.file=myTestPlan.ctp launch

Shortcut for deploy, init, start, join and collect on all probes and injectors of the given test plan. The command exits when the full process is complete. As a major difference with the use of target deploy that enables several consecutive runs on the same deployed test plan, here the test plan is deployed and executed only once.

9.13. Get specific runtime parameters of a probe or injector: params

ant -Dtestplan.name=name -Dblade.id=id params

Lists all parameters of a probe or injector that may be changed (including while running). These parameters and corresponding possible values are specific to the target probe or injector.

9.14. Change a runtime parameter of a probe or injector: change

ant -Dtestplan.name=name -Dblade.id=id -Dparam.name=param -Dparam.value=value change

Changes a parameter's value for a given injector or probe in a given deployed test plan.

10. Test results and measurements

CLIF tests gather the following data:

- test plan copy,
- Java system properties at test execution time for all probes and injectors,
- measurements from all probes and load injectors,
- life-cycle events for all probes and injectors,
- alarms generated by injectors or probes (if any).

As of current Storage component implementation, all these data are gathered in a hierarchy of CSV-files in a subdirectory of CLIF's runtime environment named "report" by default. This target directory may be changed with a system property (see section 11).

Both the Eclipse RCP-based console (section 22) and the Java Swing-based console (section 8) provide graphical and statistical analysis tools.

11. Licenses

CLIF is open source software licensed under the GNU Lesser General Public License (LGPL).

CLIF comes with facilities including the following open source software libraries:

- Jakarta commons Httpclient, from the Apache Software Foundation, released under Apache License;
- OpenLDAP from the OpenLDAP Foundation, released under OpenLDAP Public License
- Htmlparser from Source Forge, released under LGPL license;
- Eclipse graphical user interface libraries and Rich Client Platform, released under Common Public License;
- PostgreSQL JDBC driver, released under BSD license;
- DnsJava for DNS injection support, released under BSD License;
- JDOM for XML parsing in ISAC, released with a specific license.

Appendix A: XML DTDs for ISAC

ISAC scenarios (org/objectweb/clif/scenario/isac/dtd/scenario.dtd)

```
<!-- A scenario is composed of two parts :-->
<!--  - behaviors, to define some behavior...-->
<!--  - load, to define the load repartition...-->
<!ELEMENT scenario (behaviors,loadprofile)>
<!-- In the part behaviors, we must define the plugins that will be used in behaviors-->
<!ELEMENT behaviors (plugins,behavior+)>
<!-- For each plugin we define the plugin with the use tag-->
<!ELEMENT plugins (use*)>
<!-- We can add some parameters if it's needed-->
<!ELEMENT use (params?)>
<!-- We define an id which can be used in the next parts, to reference the plugin used-->
<!-- The name is the name of the plugin that will be used-->
<!ATTLIST use
  id        ID     #REQUIRED
  name      CDATA  #REQUIRED
>
<!-- Now we can define the behaviors-->
<!-- a behavior begin with the behavior tag, and can be composed of: -->
<!--   - A sample : reference to a specified sample plugin... -->
<!--   - A timer : it's a reference to a timer plugin... -->
<!--   - A while controller : it's a while loop... -->
<!--   - A preemptive : it's a controller adding a preemptive for all it children... -->
<!--   - An if controller : it's a controller doing the if / then /else task... -->
<!--   - A nchoice controller : it's a controller which permits doing random choices between some
sub-behaviors with a weight factor -->
<!ELEMENT behavior (sample|timer|control|while|preemptive|if|nchoice)*>
<!-- When we define a behavior we must define the id parameter too, -->
<!-- it will be used to reference behavior in load part-->
<!ATTLIST behavior
  id        ID     #REQUIRED
>
<!-- A sample element could need some parameters-->
<!-- the parameters needed are defined in the plugin, which will be used, definition file-->
<!ELEMENT sample (params?)>
<!-- A sample element have for parameter : -->
<!--  - use : the id of the plugin that will be used for this sample-->
<!--          the id of this plugin must be defined into the plugins part-->
<!--  - name : the name of the action that is referenced by the sample tag-->
<!--           this action name must be specified in the plugin, which is used, definition file-->
<!ATTLIST sample
  use       CDATA  #REQUIRED
  name      CDATA  #REQUIRED
>
<!-- A timer element could need some parameters-->
<!-- the parameters needed are defined in the plugin, which will be used, definition file-->
<!ELEMENT timer (params?)>
<!-- The timer have got the same parameters of a sample element-->
<!ATTLIST timer
  use       CDATA  #REQUIRED
  name      CDATA  #REQUIRED
>
<!ELEMENT control (params?)>
<!ATTLIST control
```

```
 use      CDATA  #REQUIRED
 name      CDATA  #REQUIRED
>
<!-- A while controller must contain a condition and a sub-behavior-->
<!ELEMENT while (condition,(sample|timer|control|while|preemptive|if|nchoice)*)>
<!-- A condition is a reference to a test of a specified plugin-->
<!-- it could need some parameters-->
<!ELEMENT condition (params?)>
<!-- we need specified as parameters for this tag, the plugin used and the name of the test, like
sample or timer tag-->
<!ATTLIST condition
 use      CDATA  #REQUIRED
 name      CDATA  #REQUIRED
>
<!-- A preemptive element is defined as a while element, the difference is in the execution
process-->
<!-- For a while we evaluate the condition before each loop, in a preemptive before each
action...-->
<!ELEMENT preemptive (condition,(sample|timer|control|while|preemptive|if|nchoice)*)>
<!-- An if controller must contains a condition and a sub-behavior ('then' tag)-->
<!-- And optionally it could contain another sub-behavior ('else' tag)-->
<!ELEMENT if (condition,then,else?)>
<!-- A then tag delimited the sub-behavior that will be executed if the condition is true-->
<!ELEMENT then (sample|timer|control|while|preemptive|if|nchoice)*>
<!-- A else element contains a sub-behavior too-->
<!ELEMENT else (sample|timer|control|while|preemptive|if|nchoice)*>
<!-- A nchoice plugin contains n sub-behavior, each sub-behavior have a probability to be
executed-->
<!ELEMENT nchoice (choice+)>
<!-- An choice element contain a sub-behavior-->
<!ELEMENT choice (sample|timer|control|while|preemptive|if|nchoice)*>
<!-- And this element take for parameter a probability-->
<!ATTLIST choice
 proba   CDATA  #REQUIRED
>
<!-- Now we define the params element, this element begin the part to define parameters for
the parent element-->
<!ELEMENT params (param+)>
<!-- For each param we need to define it with the param tag-->
<!ELEMENT param EMPTY>
<!-- This tag take for parameters the name of the parameter and it value-->
<!ATTLIST param
 name      CDATA  #REQUIRED
 value     CDATA  #REQUIRED
>
<!-- Now let's define the load part, this part is used to define the ramps, each ramps represent
the load for a behavior-->
<!-- We can define some ramps together in a group element, this element is used to launch
several behaviors in the same time-->
<!ELEMENT loadprofile (group*)>
<!-- A group is a composition of 'ramp' elements-->
<!ELEMENT group (ramp+)>
<!-- We need define the behavior id of the group and optionally -->
<!-- the force stop mode, default is true -->
<!ATTLIST group
 behavior       CDATA       #REQUIRED
 forceStop                 (true|false) "true"
```

```
>
<!-- each ramp could take some parameters-->
<!ELEMENT ramp (points)>
<!-- For a ramp we must define the style of the ramp, which will be used-->
<!ATTLIST ramp
  style     CDATA  #REQUIRED
>
<!ELEMENT points (point,point)>
<!ELEMENT point EMPTY>
<!-- For a ramp we must define the style of the ramp and the reference of the behavior, which
will be used-->
<!ATTLIST point
  x     CDATA  #REQUIRED
  y     CDATA  #REQUIRED
>
```

ISAC plug-ins declaration (org/objectweb/clif/scenario/isac/dtd/plugin.dtd)

```
<!-- A plugin definition begin with the plugin tag, and may contain: -->
<!-- samples, timers, tests and a session object-->
<!ELEMENT plugin (object,(sample|timer|control|test)*,help?)>
<!-- We must define the name of the plugin -->
<!ATTLIST plugin
  name     CDATA  #REQUIRED
>
<!-- We could define an object session -->
<!-- The session object could need some parameters to be initialized-->
<!ELEMENT object (params?,help?)>
<!-- We must define the id of the object and the class of the object-->
<!ATTLIST object
  class    CDATA  #REQUIRED
>
<!-- We can define some samples -->
<!-- The sample could need some params-->
<!ELEMENT sample (params?,help?)>
<!-- We now define the sample name, the sample class,-->
<!ATTLIST sample
  name     CDATA  #REQUIRED
  number   CDATA  #REQUIRED
>
<!-- We could define some tests-->
<!ELEMENT test (params?,help?)>
<!-- We have the name of the test,-->
<!--  and the number of the test stored in the session object -->
<!ATTLIST test
  name    CDATA  #REQUIRED
  number  CDATA  #REQUIRED
>
<!-- A timer have a same functioning as a test element -->
<!-- takes a name to be referenced in a behavior, and must define the number of the timer to be
used -->
<!ELEMENT timer (params?,help?)>
<!ATTLIST timer
  name    CDATA  #REQUIRED
  number  CDATA  #REQUIRED
>
<!ELEMENT control (params?,help?)>
<!ATTLIST control
```

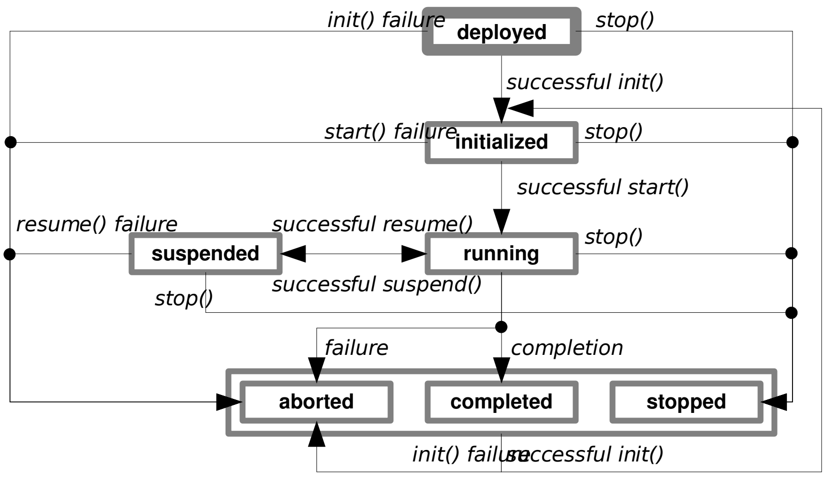
```
  name     CDATA  #REQUIRED
  number   CDATA  #REQUIRED
>
<!-- Definition of the parameters element, contains several param-->
<!ELEMENT params (param+)>
<!ELEMENT param EMPTY>
<!-- For each param we define the name and the type of the parameter, -->
<!-- that will be asked when the user will used the parent element in a behavior-->
<!ATTLIST param
  name     CDATA  #REQUIRED
  type     CDATA  #REQUIRED
>
<!-- We can define an help for the plugin, or for a sample or an object... -->
<!ELEMENT help (#PCDATA)>
```

ISAC plug-ins GUI aspects (org/objectweb/clif/scenario/isac/dtd/gui.dtd)

```
<!ELEMENT gui (object,(sample|test|timer|control)*)>
<!ELEMENT object (params)>
<!ATTLIST object
  name       CDATA        #REQUIRED
>
<!ELEMENT sample (params)>
<!ATTLIST sample
  name       CDATA        #REQUIRED
>
<!ELEMENT test (params)>
<!ATTLIST test
  name       CDATA        #REQUIRED
>
<!ELEMENT timer (params)>
<!ATTLIST timer
  name       CDATA        #REQUIRED
>
<!ELEMENT control (params)>
<!ATTLIST control
  name       CDATA        #REQUIRED
>
<!ELEMENT params (param|group)*>
<!ELEMENT param (radiobutton|field|checkbox|nfield|combo)>
<!ATTLIST param
  name     CDATA        #REQUIRED
  label    CDATA        #IMPLIED
>
<!ELEMENT group (param|group)*>
<!ATTLIST group
  name     CDATA        #REQUIRED
>
<!ELEMENT radiobutton (choice*)>
<!ELEMENT choice EMPTY>
<!ATTLIST choice
  value    CDATA        #REQUIRED
  default  (true|false)  "false"
>
<!ELEMENT checkbox (choice*)>
<!ELEMENT field EMPTY>
<!ATTLIST field
  size     CDATA        #REQUIRED
```

```
 text     CDATA         ""
>
<!ELEMENT nfield EMPTY>
<!ELEMENT table EMPTY>
<!ATTLIST table
 cols     CDATA         #REQUIRED
>
<!ELEMENT combo (choice*)>
```

Appendix B: injector and probe (aka blade)'s life cycle



Appendix C: system properties

A number of Java system properties are set in file etc/clif.props of CLIF runtime environment. This file is used by the helper ant targets provided in file build.xml located at the root of CLIF runtime environment. Should you need to use CLIF without ant, don't forget to set all these system properties when launching the appropriate class in your Java Virtual Machine.

System properties used by CLIF are listed in the table hereafter:

system property	comment	default value in file etc/clif.props	default value in binary code
java.security.policy	set Java security policy file	etc/java.policy	*none*
fractal.provider	set Fractal implementation	org.objectweb.fractal.julia.Julia	*none*
fractal.registry.host	set hostname running FractalRMI registry. The registry is now integrated to the console (so the host is the console's host)	localhost	
fractal.registry.port	set port number for the FractalRMI registry launched by the console.	1234	
julia.config	using Julia as Fractal implementation, set Julia configuration file	etc/julia.cfg	*none*
julia.loader	using Julia as Fractal implementation, set class loader	org.objectweb. fractal.julia.loader .DynamicLoader	*none*
clif.codeserver.port	set port number for class and resource server embedded in the console	1357	*none*
clif.codeserver.host	set host name for class and resource server embedded in the console	localhost	*none*

clif.codeserver.path	ordered set of directories where the codeserver may look for classes and resources it is asked for, separated by ; character. Note that, whatever the value of this property, classes and resources are first looked for in the jar files in lib/ext/ directory, and in the console's current directory. Absolute paths are used as is, while relative paths are interpreted from the root of CLIF's runtime environment.	examples/classes/ *(just to make examples run)*	*none*
clif.datacollector.delay_s	Sets the delay (in seconds) before writing an event to the storage system. Typical value should be greater than the variation of response times to get events stored in chronological order.	10	10
clif.filestorage.dir	Sets the file system directory to be created (if necessary) and used to store the generated measures. An absolute path is used as is, while a relative path is interpreted from the root of CLIF's runtime environment.	report	report
clif.isac.threads	Size of ISAC execution engine's pool of thread. The optimal value depends on the average requests throughput and the average response time.	10	10
clif.isac.groupperiod	update period (in ms) of active virtual users populations to match the specified load profiles	100	100
clif.isac.schedulerperiod	polling period (in ms) for the threads of the thread pool asking for something to do	1	1
clif.isac.jobdelay	When positive, gives the delay threshold (in ms) before an alarm is generated when a think time is longer than specified. -1 disables this feature.	-1	-1

clif.filestorage.host	sets a local IP address or a subnet number to be elected by the filestorage component when collecting events through TCP/IP sockets	*commented out*	*random choice among locally available*
jonathan.connectionfactory.host	sets a local IP address or a subnet number to be used by the FractalRMI remote object references	*commented out*	*random choice among locally available*

Other system properties may be useful for a variety of use cases (they are given in comments in file `etc/clif.props.template`):

- for remote Java debugging:
  `-agentlib:jdwp=transport=dt_socket,address=8000,server=y,suspend=n`
- for SSL certificates (for example for HTTPS support):
  `-Djavax.net.ssl.trustStore=/path/to/keystore`
  `-Djavax.net.ssl.trustStorePassword=the_keystore_password`

Appendix D: Class and resource files (remote) loading

Principle

When components are deployed in a CLIF server (probe, injector), the corresponding classes are automatically downloaded from the console if they are locally missing. Moreover, those components may require resource files (see `webtest.urls` file in `webtest` example, or `helloworld.xis` file in `isac-helloworld` example), which the user would rather not have to copy on every CLIF server. The content of these resource files can be remotely read via the console too.

This feature relies on a specific Java class loader and its associated system property `clif.codeserver.path` on the one hand, and on a so-called "code server" embedded in the console on the other hand.

Where classes and resource files are looked for?

The code server embedded in the console looks for the requested classes and resources successively in the following places:

- jar files in CLIF distribution's lib/ext/ directory where the console is running. Note: since the code server indexes the contents of all jar files in lib/ext/ at console start-up, all necessary jar files must be present before running the console;
- the console's current directory (which should be CLIF's root directory);
- the directories declared by `clif.codeserver.path` property, relative to the console's current directory.

See appendix on system properties page 47 for details on how to set the `clif.codeserver.path` property, and how to set the port number for the code server.

Appendix E: ISAC plug-ins

[TODO]

Appendix F: ISAC execution engine

The ISAC execution engine is the interpreter class for ISAC scenarios. When editing a test plan, just select the “injector” role and type `IsacRunner` in the “class” field. Then, fill the “arguments” field with the file name of the ISAC scenario you want to run. As a general advice, don't set the full path name but simply the file name, and add the directory where the scenario file resides to the code server path (see appendix p. 50). When using the Eclipse console, the file typically resides in the project directory.

The ISAC thread pool

The ISAC execution engine uses a pool of threads to run virtual users (aka behavior instances). When a virtual user is engaged in a think time, its execution thread is used to activate another virtual user. This way, the size of the thread pool is typically far smaller than the maximum of simultaneously running virtual users that is specified by the load profile. This pool has a default size that may be changed:

- before runtime:
  - either by setting system property `clif.isac.threads`
  - or by adding option `threads=my_custom_pool_size` in the “arguments” field;
- at runtime, by changing the value of parameter “threads”.

Millions of virtual users per execution engine can easily be reached. The issue is that the think times must be much greater than the response times in order to really support such a number of virtual users without violating the specified behaviors. The theoretical optimal thread pool size is:

$$\text{optimal pool size} = \frac{\text{maximum number of virtual users} * \text{average response time}}{(\text{average think time} + \text{average response time})}$$

The actual optimal pool size shall be a little greater to face possible transient variations of the global activity (when many virtual users simultaneously exit from a think time) and the overhead of context switching between virtual users. The default size is 10, but should be adjusted to your particular test case. Of course, setting an over-sized pool of threads will waste computing resources and result in performance degradation.

Deadline violation alarms (Job delay)

When the execution engine becomes overloaded, a consequence is that virtual users' think times become longer than specified. In other words, the deadline for performing the action next to the think time is violated. It is possible to get an alarm event when a given tolerance threshold is reached. This feature is enabled as soon as a positive value is set for this threshold, expressed in milliseconds. To set the threshold:

- before runtime:
  - either set system property `clif.isac.jobdelay`
  - or add option `jobdelay=my_custom_threshold_in_ms` to the “arguments” field;
- at runtime, by changing the value of parameter “jobdelay”.

Note that enabling this alarm results in a slight overhead in the execution engine functioning. Moreover, setting a small threshold value may result in a profusion of meaningless alarms: a small deadline violation from time to time does not necessarily mean the engine is overloaded. The relevant threshold value depends a lot on your use case, but a 100ms to 1000ms delay is probably a good order of magnitude. However, when analyzing the meaning of such an alarm, be careful also about the Java garbage collector that blocks the JVM and may cause deadline violations.

The default value is -1 (disabled).

Group period

The execution engine periodically checks if the current number of virtual users matches the specified load profile: in case some virtual users are missing, new ones are instantiated; in case virtual users are too numerous, some of them are stopped once their current action is complete. Stopping virtual users before the normal completion of their behaviors is performed only if the "force stop" option has been enabled in the load profile definition. Otherwise, the execution engine will just wait for the population to naturally decrease as behaviors complete.

The population checking period is set in milliseconds:

- before runtime:
  - by setting system property `clif.isac.groupperiod`
  - or by adding option `groupperiod=my_custom_group_period_ms` to the "arguments" field;
- at runtime, by changing the value of parameter "groupperiod".

The good period value is a trade-off between performance and accuracy of the engine: a short period will increase the engine overhead but the virtual users' population will be closer to the load profile specification. The default 100ms period is probably a good order of magnitude for common test cases.

Scheduler period

When a thread from the pool has just completed an action for a virtual user which is entering a think time period, it asks the engine for an action to do for another virtual user. If there is nothing to do at this time, the thread makes a small sleep before asking again, and so on until it gets something to do. The small sleep duration is given in milliseconds by the scheduler period parameter. This parameter may be changed:

- before runtime:
  - by setting system property `clif.isac.schedulerperiod`
  - or by adding option `groupperiod=my_custom_scheduler_period_ms` to the "arguments" field;
- at runtime, by changing the value of parameter "schedulerperiod".

The good period value is a trade-off between engine reactiveness and performance. A zero value should be avoided since the threads waiting for something to do would enter a frenetic polling loop on interrogating the engine, which typically wastes all processing power. A big value should be avoided too for the sake of think times accuracy. The formula below gives the possible variation range of think times:

$$\text{specified think time} \leqslant \text{actual think time} \leqslant \text{specified think time} + \text{scheduler period} + \text{context switching overhead}$$

The default 1ms value seems to be a good value for common test cases. In the general case, you should ensure that: (1) the scheduler period is significantly less than the think times, and (2) the scheduler period is significantly less than the job delay setting (when positive/enabled).

Storage options

As a CLIF load injector, the ISAC execution engine produces a number of events:

- one lifecycle event is produced each time the engine state changes: initializing, initialized, starting, running, suspended, etc. (see appendix p. 46 for details about the life-cycle specification);
- one action event is produced for each request (aka sample) on the SUT;
- one alarm event may be generated each time a think time is actually longer than specified, according to the given tolerance threshold (see Job delay parameter described above).

These events are stored unless you specify not to do so, through the following parameters:

- `store-lifecycle-events`
- `store-action-events`
- `store-alarm-events`

Acceptable enabling values are: on yes true

Acceptable disabling values are: off no false

Disabling storage for an event type has the following advantages: increased ISAC engine power, reduced time for final data collection, reduced storage space. As a matter of fact, some test cases may generate gigabytes of data that may be too heavy to analyze. Moreover, high events throughputs (thousands of events per second) may overwhelm the disk transfer rate. The drawback of disabling event storage is that you won't keep any data for this event type on this injector.

A possible smart use of this feature is to disable action events storage for some massive load injectors (heavy background load), but to store and analyze the results from a couple of load injectors generating a light load. This way, you get a reduced amount of data, and data is quite accurate because the corresponding load injectors were far from saturating.

Note that disabling storage of life-cycle events and alarm events is possible but not recommended in common test cases:

- life-cycle events give an interesting and very lightweight trace of the injector's activity steps, whatever the test duration, with no noticeable impact on the engine performance;
- the occurrence of alarm events shows that something did wrong during the test, which is key to the test analysis, while no alarm event is generated when everything goes well.

As a conclusion, storage of life-cycle and alarm events is commonly always useful and never disturbing.

Dynamic load profile change

In case your scenario defines no load profile, or when you want to dynamically change the predefined load profile while a test is running, you can change parameter "population" of the ISAC execution engine. This parameter has the following form: $b_1=n_1;b_2=n_2;$... where b_1 is the name of a behavior in the ISAC scenario and n_1 is the number of instances (aka virtual users) of this behavior.

When getting the current value of "population" parameter, if the current population is ruled by a specified load profile, you will get empty values: $b_1=;b_2=;$... Since the population may change accordingly to the load profile, no value is given. Once a population is set for a behavior, the population for this behavior becomes constant and the load profile for this behavior is definitively lost. As a result, the test will never complete by itself: you will have to stop it by yourself, at the moment that seems relevant for you.

Note that increasing a behavior's population through the setting of "population" parameter should be made carefully: all necessary new virtual users are created at once, and may result in a brutal load increase on your injector and SUT. Depending on the desired effect, it might be wise to add a linearly distributed random think time at the beginning of your behavior definition so that virtual users don't simultaneously start their actual load activity even though their are created at the same time. Of course, you must anticipate on this when writing the scenario.