March, 19th 2013

CLIFv2 developer manual

http://clif.ow2.org/

Copyright © 2006-2010, 2013 France Telecom SA

Table of contents

1. Compile from sources

You may want to recompile CLIF and generate your own runtime environment. This task is quite easy using the `ant` utility (version 1.8.0 or greater is required). A Java5-compliant JDK (or greater) is also required; Sun/Oracle's JDK 1.5 and 1.6, as well as OpenJDK6 have been successfully tested. Java 7 is also supported.

You have to get a variable number of modules from CLIF's repository, according to your needs. Each module is actually a distinct Eclipse project, providing its own .project and .classpath files.

- `clif-core` contains the core code of CLIF (**mandatory**). It also includes the ISAC execution engine, the probes and the command line interface.
- `dist` contains the resources and build chain to generate all CLIF binary distributions (**mandatory**).
- `clif-swingGui` contains the source code for the Java/Swing based console GUI, which is a simplified console. It is system-independent, but does not contain the ISAC scenario editor.
- `clif-web` contains the resources and build chain to generate CLIF's public web site.
- `isac-commons` provides a number of common features for CLIF's ISAC scenario environment: timers, random generators, data set providers, etc. It is almost mandatory when using ISAC (which should be typically the case).
- `selfbench` relies on, and extends `clif-core` with extra components for automatic control of load test runs: resource saturation detection and load injection level. This is still an early research transfer, addressing advanced CLIF users.
- `xxxInjector` projects contain the source code and build chain to generate load injection plug-ins, according to a variety of protocols, within the ISAC scenario environment.
- `xxxProvider` projects contain the source code and build chain to generate external data set provisioning plug-ins, according to a variety of modes, within the ISAC scenario environment.
- `org.ow2.clif.console.plugin` provides the Eclipse-RCP-based CLIF console as an Eclipse plug-in. This module is mandatory to build the Eclipse console, as well as the 3 other Eclipse plug-ins provided by CLIF (see below).
- `org.ow2.clif.isac` provides the Eclipse plug-in featuring the ISAC scenario editor and the wizard for extending the ISAC scenario environment.

Apache's `ant` utility is used to generate anything in this list, thanks to the `build.xml` file located in the `dist` project. Main targets are:

- `ant clif-console`
  compiles CLIF and generates CLIF standalone Eclipse™ RCP based full-fledged runtime environment for the current operating system, available as .zip files in `output` subdirectory.
- `ant clif-plugins`
  compiles CLIF, generates CLIF plugins for Eclipse (console and isac) for the current operating system, available as .zip files in `output` subdirectory
- `ant clif-server`
  compiles CLIF and generates a minimal runtime environment to run a CLIF server, zipped in output subdirectory
- `ant clif-swingGui`
  compiles CLIF and generates a runtime environment with a Swing GUI available in output/dist subdirectory

- `ant isac-plugins`
  compiles CLIF and generates a runtime environment with a Swing GUI available in output/dist subdirectory
- `ant dev-env`
  compiles and copies needed files into each project to be able to develop with your Eclipse environment.
- `ant clean-all`
  destroys output directories in all the different CLIF projects
- `ant products`
  generates everything
- `ant clif-selfbench`
  generates the `selfbench` extension for automatic test control

Then, subsequent operations are given in the following sections.

The source code is available through a SVN repository at OW2's forge. You may obtain the source code using SVN utility or by downloading a nightly-built snapshot of CLIF's SVN repository as a single zipped file.

see practical information at http://forge.objectweb.org/plugins/scmsvn/index.php?group_id=57

2. Defining your own probes

You may define your custom probes very easily by using the probe framework, as it is used by the provided probe implementations (see the user manual for probe implementations provided by CLIF). To do so, you must define a sub-package of package `org.ow2.clif.probe`, and create three classes:

- extend class `org.ow2.clif.datacollector.lib.AbstractDataCollector` to create your own `DataCollector` class. Its role is basically to provide statistical values for monitoring purpose;
- create an event class implementing interface `BladeEvent` to hold the set of values produced by each measure. Alternatively, a simpler way is to extend class `org.ow2.clif.storage.api.ProbeEvent`;
- create an `Insert` class implementing the method that actually performs measurements and creates instances of the event 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.ow2.clif.probe.weather.DataCollector`
- `org.ow2.clif.probe.weather.MyWeatherEvent`
- `org.ow2.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.

3. Defining your own load injectors

3.1. Writing your own ISAC plug-ins

3.1.1. 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 (aka virtual user), and provides specific methods for:

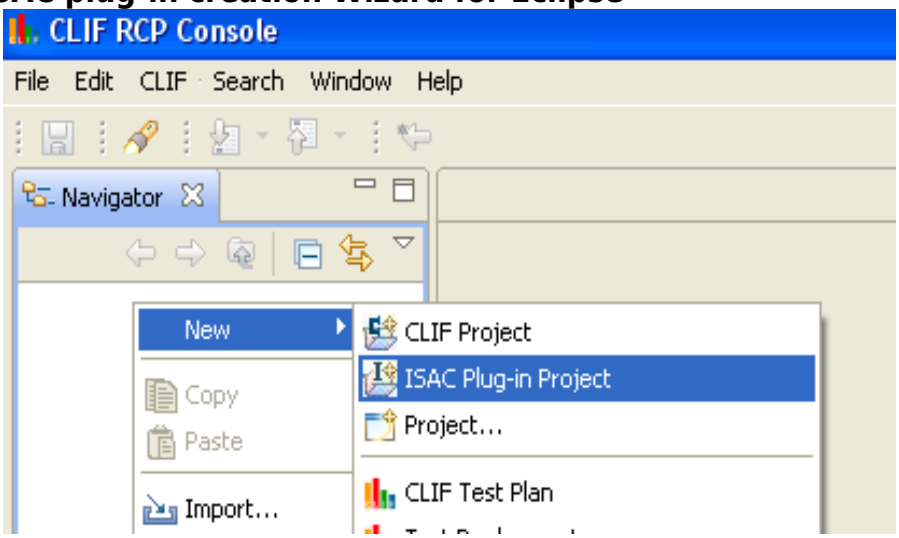
- instantiating new *session objects* for new behavior instances;
- implementing load injection primitives;
- implementing timer primitives (e.g. to implement think times);
- implementing external data provisioning;
- implementing condition primitives;
- session object control primitives.

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.

3.1.2. 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.

3.1.3. 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 SYSTEM "org/ow2/clif/scenario/isac/dtd/plugin.dtd">
<plugin name="DnsInjector">
    <object class="org.ow2.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>
```

3.1.4. 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.ow2.clif.scenario.isac.util.SessionObjectAction {
    public MyPluginSessionObject(java.util.Map<String,String> 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
...
```

3.1.5. 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.ow2.clif.scenario.isac.plugin.SessionObjectAction,
org.ow2.clif.scenario.isac.plugin.SampleAction {
    public ActionEvent doSample(int number, Map<String,String> params, ActionEvent report) {
        switch (number)
...
```

3.1.6. 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.ow2.clif.scenario.isac.plugin.SessionObjectAction,
org.ow2.clif.scenario.isac.plugin.TimerAction {
    public ActionEvent doTimer(int number, Map<String,String> params) {
        switch (number)
...
```

3.1.7. 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.ow2.clif.scenario.isac.util.SessionObjectAction,
org.ow2.clif.scenario.isac.plugin.TestAction {
    public ActionEvent doTest(int number, Map<String,String> params) {
        switch (number)
...
```

3.1.8. 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.ow2.clif.scenario.isac.util.SessionObjectAction,
org.ow2.clif.scenario.isac.plugin.ControlAction {
    public ActionEvent doControl(int number, Map<String,String> params) {
        switch (number)
...
```

3.1.9. 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.ow2.clif.scenario.isac.util.SessionObjectAction,
org.ow2.clif.scenario.isac.plugin.DataProvider {
    public ActionEvent doGet(String reference) {
...
```

3.2. Extending the MTScenario class

`MTScenario` is an abstract Java class dedicated to programmers. It is a light weight alternative to the ISAC environment, providing minimal management of virtual users. `MTScenario` makes it easy to define a test scenario as a set of concurrent threads ("sessions") looping on arbitrary actions (requests), with an initial ramp-up time and during a given test duration.

```
package org.ow2.clif.scenario.multithread;
[...]
/**
 * Abstract implementation of a multi-thread based scenario component.
 * Method newSession() should be implemented by a derived class in order to provide
 * actual action to be performed. Each session loops on calling its action() method, animated
 * by its own thread.
 * MTscenario must be given an argument line (as a single String) beginning with 3 integer
parameters:
```

```
 * <UL>
 * <LI>the number of threads
 * <LI>the loop time-out in seconds
 * <LI>the ramp-up time in seconds (added to the loop time-out)
 * </UL>
 * The trailing characters of the argument String is passed as argument to the newSession() method.
 * @see #newSession(int, String)
 * @see #setArgument(String)
 */
public abstract class MTScenario
```

The programmer just has to define the session objects and actions by providing an implementation of the MTScenarioSession interface.

```
package org.ow2.clif.scenario.multithread;

import org.ow2.clif.storage.api.ActionEvent;


/**
 * Interface to be implemented by session objects when using a MTScenario-derived scenario.
 * @see MTScenario#newSession(int, String)
 */
public interface MTScenarioSession
{
       /**
        * Performs an injection action and returns the corresponding test report (response time,
etc.).
        * @param report a pre-filled test report. Fields date, iteration, sessionId and
        * testId are filled with valid values. Field duration is set to 0, field successful is set
to
        * true, and fields type, comment, result are set to null.
        * @return the actual, complete test report, or null if this action must not
        * be taken into account
        */
       public ActionEvent action(ActionEvent report);
}
```

Refer to package org.ow2.clif.scenario.multithread in CLIF's javadoc.

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