Developers Guide to the MUSCLE

Jan Hegewald

January 11, 2010

Get the most recent version of this document at `http://muscle.berlios.de`

Contents

1 Preface

The MUSCLE is a platform independent agent system to couple multiscale simulations. It is one work package of the EU founded research project COAST [1]. The MUSCLE provides the software framework to build simulations according to the *complex automata theory*, another achievement of the COAST project. A complex automata [4] consists of a federation of cellular automata and agent-based models [6]. Within the project we focus on the simulation of coronary artery in-stent restenosis [5] and the development of a generic software library which supports the complex automata concept [3]. This software library has been named Multiscale Coupling Library and Environment (MUSCLE).

Back in 2007 and 2008, we used to call this framework the Distributed Space Time Coupling Library (DSCL).

This is a preliminary documentation to guide software developers who want to get started with the MUSCLE framework. The most recent version of this document is available at `http://muscle.berlios.de`. There you can also download the current release of the MUSCLE software.

In the following descriptions we assume that you stick to the standard directory layout of the sources, libraries, resources and such.

Some terminology and acronyms used within this document:

COAST Complex Automata Simulation Technique
MUSCLE Multiscale Coupling Library and Environment
CxA Complex Automaton (a distributed MUSCLE application coupling one ore more kernels)
JVM Java Virtual Machine
JADE Java Agent DEvelopment Framework [2]
platform a group of agents which share the same yellow page and white page services
kernel a single scale simulation wrapped into a controller agent within the MUSCLE
portal an inlet or outlet connection of a kernel
IDE Integrated Development Environment

1.1 Further reading

For further reading about the COAST project and what a CxA is, please take a look at our publication list at `http://www.complex-automata.org/dissemination-material/coast-papers` and the references at the end of this document.

1.2 License

The MUSCLE is released under the GNU Lesser General Public License. See the files `src/lgpl.txt` and `src/gpl.txt`.

1.3 Download and Installation

A bundle of the MUSCLE can be downloaded at our developers area at the BerliOS Open Source Software portal. Go to http://developer.berlios.de/projects/muscle to obtain the latest release. The bundled package contains the source code of the MUSCLE, documentation and precompiled Java archives (jar).

1.3.1 Dependencies

The core functionality of the MUSCLE is written in Java. To be able to integrate native code with the MUSCLE, there is also a native part which is written in C++. Bootstrapping and platform independent configuration is done with a highly flexible setup mechanism written in Ruby. In order to use the MUSCLE, you will need a recent Java runtime to run the MUSCLE. You can obtain it at either `http://www.java.com/en/download/index.jsp` (Sun-Java) or `http://openjdk.java.net/` (Open-JDK). We do most of the development work with the Java SE from Sun. A recent Ruby installation can be found here: `http://www.ruby-lang.org/en/downloads/`. In principle it should be possible to run the MUSCLE with JRuby instead of the standard Ruby.

The Java part of the MUSCLE also uses these third-party libraries:

JADE `http://jade.tilab.com/` (base agent framework) LGPL Version 2.1

JUNG `http://jung.sourceforge.net/` (visualization in GUI) BSD

json_simple `http://code.google.com/p/json-simple/` Apache License, Version 2.0

jscience `http://jscience.org/` (SI-Unit support) Custom

xstream `http://xstream.codehaus.org/` (serialization and debugging) BSD

The Java libraries can be installed to the provided `thirdparty` directory (see below).

For ruby 1.8.x the **json** library is also required. Install with e. g. `sudo gem install json`. The MUSCLE does currently not work with Ruby 1.9.1 or newer.

1.3.2 General Instructions

If the dependencies are in place, you are ready to use the MUSCLE. No compilation required! Extract the download, e. g.

```
$ unzip MUSCLE_2009-10-08_10-39-10.zip
```

Change your working directory to the the MUSCLE directory (e. g.)

```
$ cd MUSCLE_2009-10-08_10-39-10
```

See if the MUSCLE is working correctly

```
$ ./src/ruby/muscle.rb --version
```

In case you plan to use the MUSCLE with native code, you will have to compile the native library part of the MUSCLE using a C++ compiler (such as g++). The build script belonging to the MUSCLE also has a target to build the native libraries. This requires a C++ compiler and CMake (http://www.cmake.org/) to be available. You can also use your favourite IDE/build system instead.

1.3.3 What's in the bundle?

The MUSCLE download bundle contains the following items:

build This directory contains the compiled Java bytecode and C++ libraries

build.rb The build script. It can compile the java sources, build jar archives build the native shared libraries.

CMakeLists.txt The configuration file for cmake. Only needed if the native libraries are to be built. This file is used by the build.rb, you do not have to call cmake manually.

src This directory contains the MUSCLE source code.

thirdparty The MUSCLE will automatically load third-party libraries which are in this directory.

1.3.4 Detailed Ubuntu example

This section describes a step-by-step installation on a pristine Ubuntu distribution (Ubuntu 9.04 desktop, 32 bit). You can obtain it from http://www.ubuntu.com

In order to use the MUSCLE, a recent Java SE Runtime Environment (JRE) is required.

Screenshot 1 Find out installed Java version

```
$ java -version
java version "1.6.0_16"
Java(TM) SE Runtime Environment (build 1.6.0_16-b01)
Java HotSpot(TM) 64-Bit Server VM (build 14.2-b01, mixed mode)
```

Note: you usually do not need to compile the Java sources from the MUSCLE, as a precompiled package is already included at build/muscle.jar. If you still want to recompile the Java part of the MUSCLE, a recent Java compiler is required (javac). It is included in the Java SE Development Kit (JDK).

As of this writing, the version of the JRE/JDK is 6u16. See http://java.sun.com/javase/downloads/index.jsp for the most recent version.

Screenshot 2 Find out the version of the installed Java compiler

```
$ javac -version
javac 1.6.0_16
```

The MUSCLE also requires a Ruby interpreter to be installed. It is available via http://www.ruby-lang.org. There has been a major version change in Ruby, which introduces some backwards compatibility problems. For the MUSCLE we rely on Ruby version 1.8.x. The recently released version 1.9.1 and newer are not supported yet, because they impose a major change in the Ruby language. As soon as the new Ruby language becomes more widely used, we will upgrade the MUSCLE to prefer the most recent version. As of this writing, Ruby 1.8.7 will be the installed via the package manager on Ubuntu 9.04.

Note: if you install Ruby with the Ubuntu package manager, you may notice that Ubuntu breaks ruby into multiple packages. So one has to install two Ubuntu-packages in order to get a working ruby environment: *ruby* and *ruby-dev*.

Screenshot 3 Find out installed Ruby version

```
$ ruby --version
ruby 1.8.7 (2008-08-11 patchlevel 72) [x86_64-linux]
```

Apart from the the MUSCLE core modules, which are written in Java, there is also a C++ module to allow bindings to native code. These are only required if you want to integrate a native programming language, such as C++, C or Fortran. We rely on the cross-platform build utility CMake to compile the native modules of the MUSCLE. It is available from http://www.cmake.org/. Also make sure that a C++ compiler is available on your system, e. g. g++.

To compile the native libraries of the MUSCLE, run the build skript with no arguments:

```
$ ./build.rb
```

If you see an error message from cmake, where it states that JAVA_INCLUDE_PATH is set to NOTFOUND, then you can try to define an environment variable called JAVA_HOME and point it to your Java installation.

Screenshot 4 Version of installed CMmake and g++

```
$ cmake --version
cmake version 2.6-patch 2

$ g++ --version
g++ (Ubuntu 4.3.3-5ubuntu4) 4.3.3
Copyright (C) 2008 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There i
s NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PUR
POSE.
```

The most recent bundle of the MUSCLE is available at http://muscle.berlios.de. Download and extract the zip file. The directory listing should look like the listing in Fig. Directory contents of the MUSCLE5.

Screenshot 5 Directory contents of the MUSCLE

```
$ ls
build  build.rb  CMakeLists.txt  doc  src  thirdparty
```

See which version of the MUSCLE you are using with the following command:

```
$ ./src/ruby/muscle.rb --version
```

Screenshot 6 Information about your installation of the MUSCLE

```
$ ./src/ruby/muscle.rb --version
        tmp dir is: </tmp/hegewald_20091007110824_12956>
this is the Multiscale Coupling Library and Environment (MUSCLE) from
2009-09-30 14:14:52, native library available
[12956] executing pid: 12959
        tmp dir was: </tmp/hegewald_20091007110824_12956>
```

I you need to recompile the Java core of the MUSCLE, run the following command:

```
$ ./build.rb java
```

2 Examples

There are some examples which belong to the MUSCLE. Example code for kernels can be found in the directories java/examples and cpp/examples. There are also configurations for coupled example CxA (MUSCLE simulations), which can be found in the cxa directory.

2.1 Example Kernels

Hello World

At java/examples/simplejava/ConsoleWriter we provide a simple kernel written in Java. It receives data (an array of double) and prints its content to the standard output.

Data Provider

At java/examples/simplejava/Sender we provide a minimalist kernel written in Java, which sends data (an array of double).

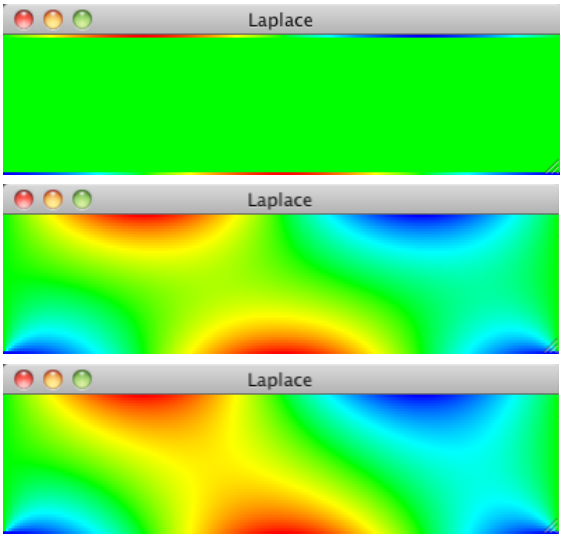
Native C++ Kernel

The directory cpp/examples/simplecpp contains a kernel (Sender) which is using native code to send and receive data. This C++ kernel provides the same behaviour as the Java kernel at java/examples/simplejava/Sender.

Laplace Heat Flow

The simulation at java/examples/laplace solves the Laplace equation to simulate a heat flow. It is implemented as a cellular automaton in 2D. The solver also visualizes its calculated results in a graphical user interface.

Screenshot 7 Output of sequential Laplace simulation



(a) begin

(b) middle (∼3000 iterations)

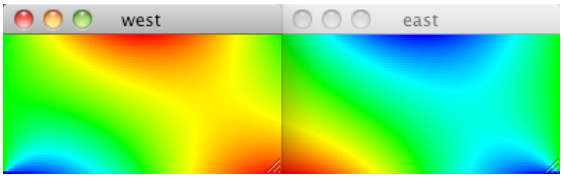
(c) end (∼6000 iterations)

The coloured output shows the temperature, which is continuously changing until it reaches steady state. The colour indicate

blue	−100 % temperature
green	0 % temperature
red	100 % temperature

The two kernels (`KernelWest` and `KernelEast`) calculate one part of the whole domain. Because each kernel uses a similar lattice, we have identical time and space scales for both kernels. Therefore no special kind of data mapping is required: we can just pass the raw data to the other domain. This setup is similar to a standard MPI parallel computation. A ghostnode column is used at the interface to be able to compute the temperature.

Screenshot 8 Output of parallel Laplace simulation (∼6000 iterations)



2.2 Example Simulations (CxA)

The `cxa` directory contains example CxA configurations:

Hello World CxA

The configuration `cxa/SimpleExample.cxa.rb` configures a minimal CxA which couples the `java/examples/simplejava/Sender` and `java/examples/simplejava/ConsoleWriter` kernels. Run with:

```
$ ./src/ruby/muscle.rb --cxa_file src/cxa/SimpleExample.cxa.rb --main plumber w r
```

Hello C++

This CxA is similar to the *Hello World CxA* from above, but it substitutes the Sender with a C++ implementation. Run with:

```
$ ./src/ruby/muscle.rb --cxa_file src/cxa/NativeExample.cxa.rb --main plumber w r
```

Distributed Laplace Heat Flow

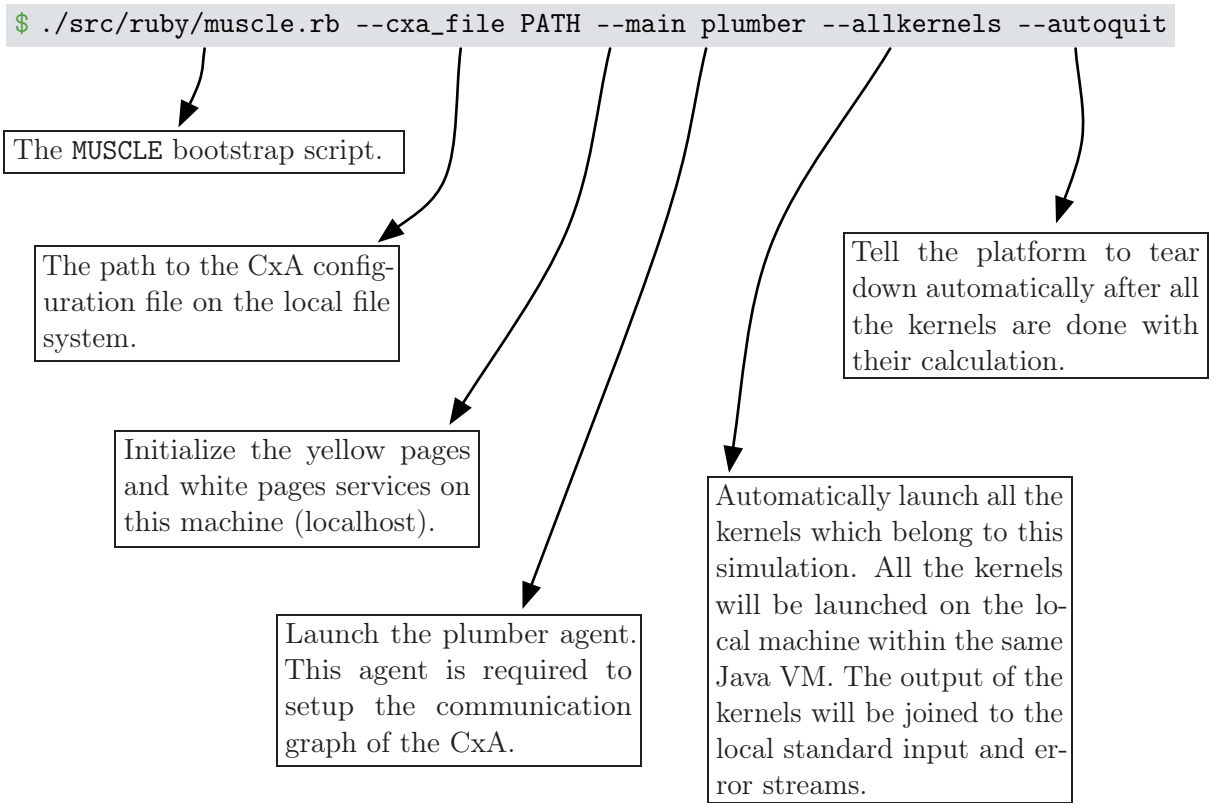
The simulation at `cxa/LaplaceExample.cxa.rb` configures the distributed Laplace temperature calculation using the kernels KernelWest and KernelEast from `java/examples/laplace`. Run with:

```
$ ./src/ruby/muscle.rb --cxa_file src/cxa/LaplaceExample.cxa.rb --main plumber east west
```

3 Launch a Simulation

This section describes the general procedure to launch a `MUSCLE` simulation (CxA). To quit a running CxA at any time, hit `ctrl-c`. This will terminate the whole platform if the `--main` option was present (see below), otherwise only the current shell command will be aborted.

Use the following command to launch a simulation from a single command:



There may be situations where one wants to have a separate shell session for a specific kernel in the simulation. To launch an individual kernel, one has to know the kernel short name. This is configured in the CxA configuration file. To get a list of used kernels, call

```
$ ./src/ruby/muscle.rb --cxa_file PATH
```

without any additional arguments. This will print a list of the kernels used in this CxA.

For example the command:

```
$ ./src/ruby/muscle.rb --cxa_file src/cxa/SimpleExample.cxa.rb
```

will show you the two kernels (`w` and `r`) from this CxA:

```
2 known kernels in CxA:
      w: examples.simplejava.Sender
      r: examples.simplejava.ConsoleWriter
1 known administration agents in CxA:
      plumber: muscle.core.Plumber
```

If e. g. the kernel `w` should be launched in a separate shell, first launch the `MUSCLE` platform with all other kernels:

```
$ ./src/ruby/muscle.rb --cxa_file src/cxa/SimpleExample.cxa.rb --main plumber r
```

Then add the remaining kernel from another shell:

```
$ ./src/ruby/muscle.rb --cxa_file src/cxa/SimpleExample.cxa.rb w
```

The computation will start as soon as all kernels are up and running.

If one kernel should be executed on a remote machine, the IP address and port of the machine where you issued the `-main` flag must be added to the command. Because `MUSCLE` allows multiple users to run independent CxAs at the same time, the default port where a kernel can connect to a CxA is based on the uid. One can find out the default port which is used for the current user with the following command:

```
$ ./src/ruby/muscle.rb --print_env=mainport
```

Now pass the IP address and the port to the flags `--mainhost` and `--mainport` respectively.

Sometimes there are several network names associated with the same machine. If you can not connect to the `--main` machine, try to overwrite the IP address when you launch the `--main` command using the `--localhost` flag.

4 Couple Your Own Solver

4.1 Implement a Kernel

Every kernel must inherit from the `muscle.core.kernel.CAController`. Here you have to define its three abstract methods (see Java documentation of the `MUSCLE`):

- Return the SI scale your kernel is operating at (a `muscle.core.Scale`) from the `getScale` method.
- Announce the portals which your kernel can use to communicate with other kernels in the `addPortals` method.
- Finally implement the runtime logic in the `execute` method.

4.1.1 Plain Java

To get you started with a pure Java kernel, there exists a very simplistic example (`src/java/examples/simplejava`). You can make a copy from its sources and use it as a draft for your own implementation.

4.1.2 Native C++

In addition to the pure Java core of the MUSCLE, there is also a supplementary library for support of native code (C++/C/Fortran). See section 1.3 for installation instructions. Look at the simplecpp example for a very simple sample of a `muscle.core.kernel.CAController` using native code in its execute method.

4.2 Setup a CxA

If you want to build a multi scale simulation using the MUSCLE, you will need at least two different kernels (although it is possible to use only one kernel and connect it to itself, this would not make much sense). A CxA then requires a configuration file, wherein you specify the setup:

```
1  # comment lines begin with a #
2
3  # get hold on the CxA handle
4  cxa = Cxa.LAST
5
6  # set some environment variables for the CxA
7  cxa.env['mykey'] = 'myvalue' # assign text to a key
8  cxa.env['another_key'] = 2 # numbers are also fine
9  cxa.env['yet_another_key'] = 4E-6 # numbers in scientific
       notation are fine, too
10
11 # declare kernels which can be launched in the CxA (use fully-
       qualified java class names)
12 cxa.add_kernel('k0', 'full.java.classname')
13 cxa.add_kernel('k1', 'another.java.classname')
14
15 # configure connection scheme of the CxA
16 cs = cxa.cs
17
18 # configure unidirectional connection from k0 to k1
19 cs.attach 'k0' => 'k1' do
20     tie 'entrance0', 'exit0' # entrance0 of k0 will feed exit0
          of k1
21     tie 'mydata' # if the names of entrance and exit are equal,
          just pass the name once
22     tie 'entrance2', 'exit2', Conduit.new("classname.for.conduit
          ") # you can connect entrance and exit with a custom
          conduit
23     tie 'entrance3', 'exit3', Conduit.new("muscle.core.conduit.
          AutomaticConduit", ["classname.for.filter0", "classname.
          for.filter1"]) # use the AutomaticConduit if you
          automatically want to connect conduit filters which are
          muscle.core.conduit.filter.WrapperFilter<DataWrapper>
24 end
```

Every one of the kernel connections is a unidirectional pipeline which should be dedicated to pass a specific kind of data from/to your kernel. The key elements of such a pipeline are the conduit (`muscle.core.conduit.BasicConduit`) and a connecting portal

on either side. These are a `muscle.core.ConduitEntrance` on the sending side and a `muscle.core.ConduitExit` on the receiving side.

The command

```
$ muscle --cxa_file /path/to/cxa --nojade muscle.core.ConnectionScheme
```

should launch a GUI with a drawing of the communication graph of the specified CxA.

4.3 Integrate Legacy Code

The MUSCLE offers straight forward support to integrate legacy code as a single scale model into any CxA configuration. You just have to write the corresponding `muscle.core.kernel.CAController` as described in section 4.1, then call your legacy code within the `execute` method. Without any portals added yet, you should be able to launch the kernel in sandbox mode

```
$ muscle --sandbox name.of.kernel
```

where your programme should execute as before.

Now you can introduce portals for sending and receiving data to the framework. Use a conduit entrance (`muscle.core.ConduitEntrance`) to send data to the framework and a conduit exit (`muscle.core.ConduitExit`) to receive data from the framework. The sends are always non-blocking operations, whereas the receives are always blocking calls. This is a great benefit which enables the MUSCLE to synchronize all the message passing within a CxA. You do not have to implement a steering logic to your code, e. g. there is no need for your execution code to respond to any calls from the framework. This is a great advantage as the whole runtime control remains within your legacy code.

Assume you have a programme which will produce a time series of output files. If you want this data to be available for other kernels in the framework, just replace the point where you write to a file with a send call on a conduit entrance.

```
42 // former: write to file
43 //fileWriter.write(data);
44 // now: dump to coupling environment
45 entrance.send(data);
```

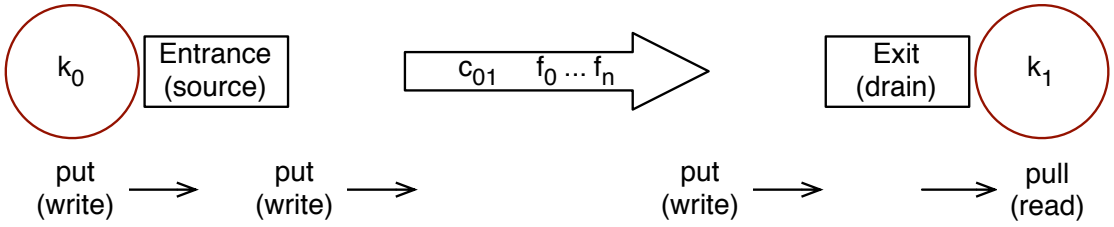
4.4 Write a Conduit Filter

If you want to connect a pair of kernels whose I/O data share similar descriptions, but differ in detail (e.g. because they use different coordinate systems), you can use a conduit filter to modify the data as it is passed to the target kernel. This way you do not have to alter either of the kernels – just use the conduit to create a matching data set for the receiver.

Your dedicated filter must implement the `muscle.core.conduit.filter.Filter` interface to be used within a conduit. Inside a conduit any number of filters can be used in

succession to alter a data package. Maybe you want to copy the `muscle.core.conduit.filter.ConsoleWriterFilter` to have a code skeleton to begin with.

Figure 1 Components involved in message passing (kernel k_i, conduit c_j, filter f_k)



4.5 Use the Logging System

Within the MUSCLE a logging system similar to the standard Java logging mechanism can be used. To obtain the logger for a kernel, call its `getLogger` method. If you want to log from non-kernel classes, please use `muscle.logging.Logger.getLogger` to get hold on a logger. This will ensure an identical naming scheme for all loggers, which simplifies individual logger configuration.

You can also use this logging system from within your native code. There you can choose to log to the standard Java logging system, or just to stdout in case you want to run your code without any JVM. The simplecpp example contains a sample usage of the `logging/Logger.h`.

4.6 Manage Filesystem Input/Output

With larger setups it becomes more and more important to manage a structured directory layout for output files. Imagine a CxA with dozens of kernels, which may be launched by multiple users on the same or different machines. To reduce the clutter, the MUSCLE provides standard temporary directories per kernel. Its path can be accessed at runtime via the `getTmpPath()` of the `muscle.core.kernel.CAController`. You should never rely on things like the CWD or paths which may not be writable by all users.

5 Code Changes

```
Code changes since version 2009-12-02_13-07-28
- use MiscTool to determine amount of physical RAM, then use a fraction (
    currently 90%) as Xmx for the JVM
- detailed information about native library in version message
- fixed an issue when detecting if an agent uses native extensions
- data messages were not properly transferred to remote containers
- ruby 1.9 adjustments in [cli.rb, muscle.rb]
- tested with ruby 1.9.1p378 and ruby 1.8.7
```

Code changes since version 2009-11-09_09-18-22
- ConduitExit deserializes the message contents itself, no longer done by JADE
- ConduitEntrance does not use JADE anymore to serialize a message
- assertions are now disabled by default
- modified MiscTool to be able to determine amount of physical memory, regardless of the values of the -Xms and -Xmx JVM flags
- removed output for tmp directory for plain java commands without jade
- major rewrite of the message passing mechanism: the kernel agent now has a separate thread to deserialize data messages and push them ASAP to the ConduitExits via a separate data channel. there is now also a separate data channel to send special administrative messages to a kernel
- the kernel will now limit the incoming message queue according to the estimated memory size it consumes. to be able to do this, the kernels have to inform each other if a message buffer is full, to be able to pause the sending kernel
- the conduit mechanism now allows a much faster message passing, if no filters are being used (80% faster with a shared memory JVM, more than 100% faster with a distributed setup)
- the Invoker did not properly use passed arguments
- the sandbox environment has been temporarily disabled

Code changes since version 2009-10-08_10-39-10
- native build now also possible if OpenJDK is installed instead of Sun-Java
- redesigned the module which finds the JNI components for a cmake setup (the FindJNI.cmake installed with cmake is additionally being used for the search and we now also locate openjdk-6 on ubuntu)
- version info tells if JVM is running in 32 or 64-bit
- fixed an issue where the JadeLeap library was not correctly detected (e. g. if name was something other than Jade.Leap.jar)
- abort if one of the port is out of range [0--65535]
- print help message from cmake.rb if called with incorrect number of arguments
- fixed an issue with cmake.rb where paths where not properly escaped
- some paths in the examples section of the developers manual where incomplete
- the developers manual still contained obsolete bootstrap flags
- fixed a configuration issue which did not load the jade mobility service, this prevented the conduits to initialize on remote containers
- the bootstrap utility now treats ports as integers
- simplified directory structure of the example CxAs

```
- new native example kernel and CxA which provides the functionality of
    the Java Sender but in C++
- new ruby class to represent memory ranges
- the --heap flag now allows to specify a memory range instead of
    identical values to min and max
- changed default Java heap allocation range to 100m..600m
- additional cpp example which is the native version of the simple java
    example
- restructured transmutable example
- new not-so-simple example CxA calculating 2D heat flow
- changed level for some logging messages to be below info level
- changed logging configuration to be less verbose
```

6 Acknowledgments

This research is supported by the European Commission, through the COAST project (EU-FP6-IST-FET Contract 033664).

References

[1] Mission of coast – complex automata. `http://www.complex-automata.org/`.

[2] Fabio L. Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley Series in Agent Technology. Wiley, 2007.

[3] Jan Hegewald, Manfred Krafczyk, Jonas Tölke, Alfons G. Hoekstra, and Bastien Chopard. An agent-based coupling platform for complex automata. In Marian Bubak, G. Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *ICCS (2)*, volume 5102 of *Lecture Notes in Computer Science*, pages 227–233. Springer, 2008.

[4] A. G. Hoekstra, B. Chopard, P. Lawford, R. Hose, M. Krafczyk, and J. Bernsdorf. Introducing complex automata for modelling multi-scale complex systems. In *Proceedings of European Complex Systems Conference*. European Complex Systems Society, Oxford, UK, 2006.

[5] Harald Mudra, Evelyn Regar, Volker Klauss, Frank Werner, Karl-Heinz Henneke, Efthia Sbarouni, and Karl Theisen. Serial Follow-up After Optimized Ultrasound-Guided Deployment of Palmaz-Schatz Stents: In-Stent Neointimal Proliferation Without Significant Reference Segment Response. *Circulation*, 95(2):363–370, 1997.

[6] D. C. Walker, J. Southgate, M. Holcombe, D. R. Hose, S. M. Wood, Mac S. Neil, and R. H. Smallwood. The epitheliome: Agent-based modelling of the social behaviour of cells. *Biosystems*, 76((1-3)):89–100, August 2004.