A Practical Guide to Amelia: A Domain Specific Language for Automated Software Deployment

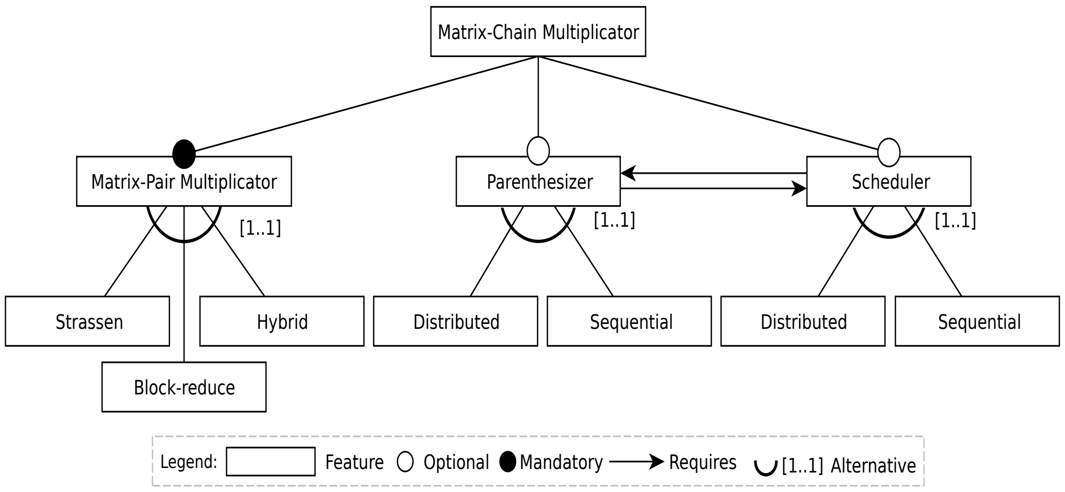
October 2017

1 Introduction

To facilitate and promote the use of the AMELIA DSL, our language that assists application developers in automating the deployment of component-based software systems, we designed a deployment case study to use the main features of AMELIA. Through this practical guide, a *software deployer* will be able to get a deeper understanding of the constructs defined in AMELIA, which will increase the deployer's ability for specifying the required deployment specifications with the language.

2 Case Study: The Matrix-Chain Multiplication Problem

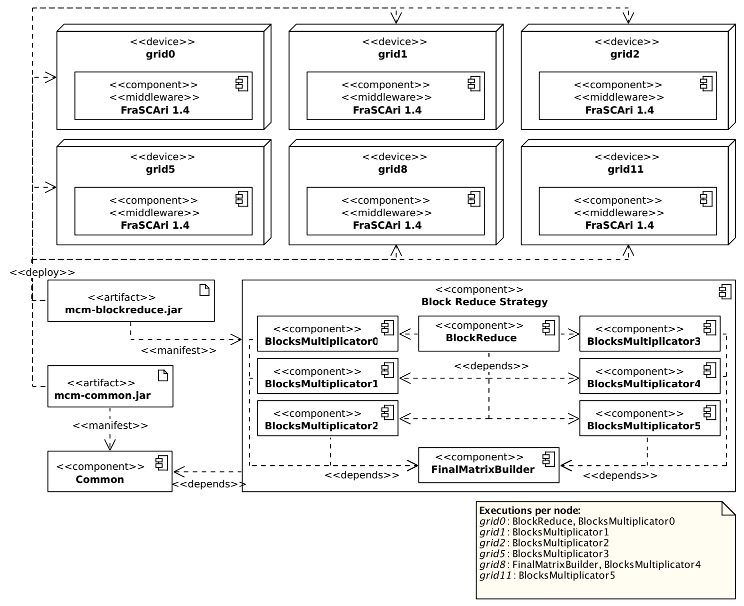
The Matrix-Chain Multiplication (MCM) problem is an optimization problem that consists in finding the most efficient multiplication sequence to multiply a set of given matrices. Our implementation of the MCM, provided to the workshop participants, splits the problem into three different subproblems: the matrix-pair multiplication problem, the matrix-chain parenthesization problem, which finds the optimal sequence of matrix-pair multiplications minimizing the number of individual additions and multiplications, and the matrix-subchain multiplication scheduling problem, which finds subsets of matrix multiplications that can be performed concurrently to decrease the overall multiplication time [1]. In this way, by combining the different solutions to these subproblems, it is possible to configure several different actual solutions to the whole problem, which raises a problem of solution configuration. For instance, by combining the first and second subproblems, one can obtain a solution able to multiply a set of given matrices reducing the number of individual arithmetical operations. In the same sense, by combining the first and third subproblems, one would obtain the same solution aforementioned, but this time reducing multiplication time. And of course, by combining the three subproblems one would reduce both operations and overall processing time. In practice, however, there can be computational limitations and trade-offs that may make infeasible some of the possible solution configurations.


Matrix-Chain Multiplicator
Matrix-Pair Multiplicator
Parenthesizer
Scheduler
[1..1]
[1..1]
[1..1]
Strassen
Hybrid
Block-reduce
Distributed
Sequential
Distributed
Sequential
Legend: Feature Optional Mandatory Requires [1..1] Alternative

In this implementation of the MCM solution, we take advantage of distributed computational resources in order to reduce the execution time when multiplying a large number of considerably big matrices. To this end, we developed two multiplication strategies, one based on the map-reduce architecture, and a variation of it that significantly reduces network usage. At the end, local multiplications are performed using the Strassen algorithm.

The following deployment diagrams depict the high-level elements composing each of the multiplication strategies. For sake of simplicity, we omit the details of the scheduling and parenthesizing subproblems. As there is only one artifact per strategy (*i.e.*, one resulting artifact of the compilation process), a note on each diagram specifies the node in which the components are executed.

2.1 BlockReduce Strategy



<<device>> grid0
<<component>> <<middleware>> FraSCAri 1.4
<<device>> grid1
<<component>> <<middleware>> FraSCAri 1.4
<<device>> grid2
<<component>> <<middleware>> FraSCAri 1.4
<<device>> grid5
<<component>> <<middleware>> FraSCAri 1.4
<<device>> grid8
<<component>> <<middleware>> FraSCAri 1.4
<<device>> grid11
<<component>> <<middleware>> FraSCAri 1.4
<<deploy>>
<<artifact>> mcm-blockreduce.jar
<<manifest>>
<<artifact>> mcm-common.jar
<<manifest>>
<<component>> Common
<<depends>>
<<component>> Block Reduce Strategy
<<component>> BlocksMultiplicator0
<<component>> BlockReduce
<<component>> BlocksMultiplicator3
<<depends>>
<<component>> BlocksMultiplicator1
<<component>> BlocksMultiplicator4
<<component>> BlocksMultiplicator2
<<component>> BlocksMultiplicator5
<<component>> FinalMatrixBuilder
<<depends>>
<<depends>>
Executions per node:
grid0 : BlockReduce, BlocksMultiplicator0
grid1 : BlocksMultiplicator1
grid2 : BlocksMultiplicator2
grid5 : BlocksMultiplicator3
grid8 : FinalMatrixBuilder, BlocksMultiplicator4
grid11 : BlocksMultiplicator5

Figure 1: Deployment Diagram for the BlockReduce Configuration Strategy

The BlockReduce consiguration strategy consists in splitting each matrix into fixed-size blocks (*i.e.*, sub-matrices) and multiply them as if they were one cell instead of a group of them. For instance, having two squared matrices A and B, partitioned into 4 blocks each, the resulting matrix C would be calculated using the same blocks partition strategy. C_{00} represents the first block of C, and would be calculated by operating A and B such that $C_{00} = A_{00} * B_{00} + A_{01} * B_{10} + A_{02} * B_{20} + A_{03} * B_{30}$. In this strategy, the block size is crucial to find the threshold between the amount of data transmitted over the network and the size of the blocks to multiply, in order to reduce the multiplication time. We performed several experiments and found that for matrices of approximately 3600x3600 elements, the block size with best execution times is 200.

2.2 Hybrid Configuration Strategy

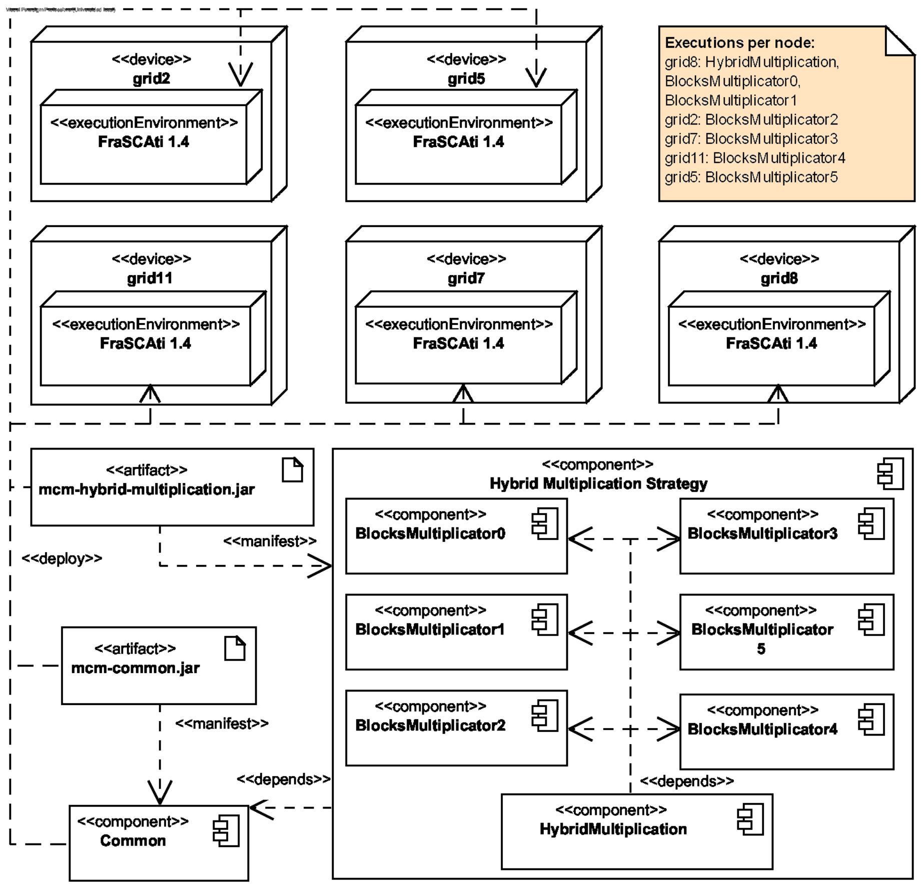

Executions per node:
grid8: HybridMultiplication, BlocksMultiplicator0, BlocksMultiplicator1
grid2: BlocksMultiplicator2
grid7: BlocksMultiplicator3
grid11: BlocksMultiplicator4
grid5: BlocksMultiplicator5
<<device>> grid2
<<executionEnvironment>> FraSCAti 1.4
<<device>> grid5
<<executionEnvironment>> FraSCAti 1.4
<<device>> grid11
<<executionEnvironment>> FraSCAti 1.4
<<device>> grid7
<<executionEnvironment>> FraSCAti 1.4
<<device>> grid8
<<executionEnvironment>> FraSCAti 1.4
<<artifact>> mcm-hybrid-multiplication.jar
<<deploy>>
<<manifest>>
<<component>> Hybrid Multiplication Strategy
<<component>> BlocksMultiplicator0
<<component>> BlocksMultiplicator1
<<component>> BlocksMultiplicator2
<<component>> BlocksMultiplicator3
<<component>> BlocksMultiplicator 5
<<component>> BlocksMultiplicator4
<<artifact>> mcm-common.jar
<<manifest>>
<<depends>>
<<depends>>
<<component>> Common
<<component>> HybridMultiplication

Figure 2: Deployment Diagram for the Hybrid Configuration Strategy

The Hybrid configuration strategy introduces an improvement, in terms of network usage, to the BlockReduce configuration strategy. However, it is more demanding in terms of processor and memory usage. In the strategy above, calculating a block in the resulting matrix requires sending as many pairs of blocks as columns or rows of blocks are, while in this strategy it only requires sending the whole column and row of blocks. Another advantage of this strategy is that it also reduces the amount of processors necessary to multiply the blocks.

3 The Amelia Language: Specification Examples

Amelia allows specifying two types of elements: Subsystems and Deployments. Along with this practical guide you will be given an overview of the language syntax and the associated semantics. The following examples describe two deployment requirements and their corresponding solution in Amelia.

3.1 Requirement 1 (Example)

Specify the deployment of the BlockReduce and Hybrid strategies having into account component dependencies and the computing nodes specified in the deployment diagram. You may use the following file containing a mapping between the hosts in the deployment diagrams and the laboratory's computers.

```
hgrid1   21 22 root  liasondriso_hgrid1   grid0
hgrid17  21 22 root  liasondriso_hgrid2   grid1
hgrid16  21 22 root  liasondriso_hgrid3   grid2
hgrid15  21 22 root  liasondriso_hgrid4   grid3
hgrid12  21 22 root  liasondriso_hgrid5   grid4
hgrid9   21 22 root  liasondriso_hgrid6   grid5
hgrid6   21 22 root  liasondriso_hgrid7   grid6
hgrid10  21 22 root  liasondriso_hgrid10  grid7
```

Listing 1: hosts.txt

3.1.1 Reusing code with Amelia

In order to reduce the size of the deployment specifications with Amelia and promote reuse and maintainability concerns, we should begin the specification process through the identification and definition of elements that could be used repeatedly over this process. In case, we created a subsystem called *Common*, which contains common definitions for all of the deployments/subsystems.

```
package co.edu.icesi.driso.matrices

import java.util.List
import java.util.Map
import org.amelia.dsl.lib.descriptors.Host

/*
 * Common definitions for all of the
     deployments/subsystems.
 */
subsystem Common {

```

```
/*
 * All hosts.
 */
param Map<String, Host> hosts

//--------------------------------------------------------------------------
// Compilation parameters
//--------------------------------------------------------------------------

/*
 * Compilation host.
 */
var Host compilationHost = hosts.get("grid0");

/*
 * Sources to compile.
 */
var String commonSources =
    "/home/sas1/LF_RIVERA/workspace-matrices/org.driso.matrices.common";
var String blockRSources =
    "/home/sas1/LF_RIVERA/workspace-matrices/org.driso.matrices.blockreduce";
var String hybridSources =
    "/home/sas1/LF_RIVERA/workspace-matrices/"
+ "org.driso.matrices.hybrid_multiplication";
var String nMatricesSources =
    "/home/sas1/LF_RIVERA/workspace-matrices/org.driso.matrices.nmatrices";
var String strassenSources =
    "/home/sas1/LF_RIVERA/workspace-matrices/org.driso.matrices.strassen";

/*
 * Compilation sources.
 */
var List<String> sources = #[
     commonSources, blockRSources, hybridSources,
         nMatricesSources,
     strassenSources
]

/*
 * Built sources folder (The site where compilation
     artifacts are located).
 */
var String builtFolder =
    "/home/sas1/LF_RIVERA/workspace-matrices/built-sources";

//--------------------------------------------------------------------------
// Allocation parameters
//--------------------------------------------------------------------------

/*
```

```
     * The folder in the execution nodes where artifacts
        are allocated.
     */
    var String allocationTargetFolder = "/home/sas1/";

    /*
     * Target sources folder in the execution nodes
     * (The site where the jars are executed).
     */
    var String builtsFolder =
        'allocationTargetFolderbuilt-sources';
}
```

Listing 2: Subsystem *Common* for reusable definitions.

As you probably noted, AMELIA allows string interpolation through the use of guillemet symbols («») between single quotation marks. Since string interpolation makes reading code much easier, especially when the string is part of a command declaration, string interpolations should be used instead of string concatenation.

3.1.2 Compiling sources with AMELIA

Since the addressed problem comprises multiple deployment strategies, we must compile certain software project sources for generating the desired specific artifacts to be deployed. In contemplation of standardization, we define a Java Enum for defining the possible deployment configurations for the matrix-chain multiplication problem.

```
package co.edu.icesi.driso.matrices.classes;

/**
 * The mcm strategies.
 * @author Miguel Jimenez (miguel@uvic.ca)
 * @date 2017-08-19
 * @version $Id$
 * @since 0.0.1
 */
public enum Strategy {
    BLOCK_REDUCE,
    HYBRID_MULTIPLICATION,
    N_MATRICES,
    STRASSEN
}
```

Listing 3: Java Enum for possible deployment configurations.

We defined a subsystem called *Compile* for the compilation of the associated source projects. This subsystem has a parameter for indicating the desired deployment strategy to be compiled. When defining the compilation subsystem, we can include the *Common* subsystem that we have created previously through the clause *includes*. All of the variables and rules defined in the *Common* subsystem can be reused in the new *Compile* subsystem, either to declare new variables or for defining new rules. This allows to avoid duplicating code and promotes reusability and composability concerns. For sake of simplicity, once artifacts have been generated, we need to place them in a shared folder. Therefore, we need to specify a dependency between two rules in the subsystem (compilation and relocation). AMELIA supports this through the use of a colon symbol (:) separating the rules, following a pattern like *dependantRule*:*ruleDependency*.

```
package co.edu.icesi.driso.matrices

import co.edu.icesi.driso.matrices.classes.Strategy

includes co.edu.icesi.driso.matrices.Common

/*
 * Compile each strategy's source code.
 */
subsystem Compile {
     /*
     * The multiplication strategy to compile.
     */
   param Strategy strategy

   on compilationHost {
      compileCommon:
         cd sources.get(0)
         compile "src" "mcm-common"
         cmd 'yes | cp -f mcm-common.jar builtFolder/'
   }

   on compilationHost ? strategy ==
       Strategy.BLOCK_REDUCE {
      strategy1: compileCommon;
         cd sources.get(1)
         compile "src" "mcm-blockreduce" -classpath
             'builtFolder/mcm-common.jar'
         cmd 'yes | cp -f mcm-blockreduce.jar
             builtFolder'
   }

   on compilationHost ? strategy ==
       Strategy.HYBRID_MULTIPLICATION {
      strategy2: compileCommon;
         cd sources.get(2)
         compile "src" "mcm-hybrid-multiplication"
             -classpath 'builtFolder/mcm-common.jar'
```

```
            cmd 'yes | cp -f mcm-hybrid-multiplication.jar
                builtFolder'
    }

    on compilationHost ? strategy == Strategy.N_MATRICES {
        strategy3: compileCommon;
            cd sources.get(3)
            compile "src" "mcm-nmatrices" -classpath
                'builtFolder/mcm-common.jar'
            cmd 'yes | cp -f mcm-nmatrices.jar builtFolder'
    }

    on compilationHost ? strategy == Strategy.STRASSEN {
        strategy4: compileCommon;
            cd sources.get(4)
            compile "src" "mcm-strassen" -classpath
                'builtFolder/mcm-common.jar'
            cmd 'yes | cp -f mcm-strassen.jar builtFolder'
    }

}
```

Listing 4: Subsystem *Compile* for compiling the source code.

3.1.3 Artifacts transportation with Amelia

Once artifacts have been generated from source code, they need to be transported (along with dependency libraries) to the computing nodes where they will be deployed. Amelia supports this requirement through the use of a command called *scp*, nevertheless, in order to work, this command requires the configuration of an FTP Server. For sake of simplicity, we decided to exploit the extensibility of Amelia and created a new user-defined command that leverage the SSH-based secure copy command of Linux. For this purpose, we created a new class which contains a method for describing the user-defined Amelia command.

```
package co.edu.icesi.driso.matrices.classes;

import java.io.IOException;
import org.amelia.dsl.lib.CallableTask;
import org.amelia.dsl.lib.descriptors.CommandDescriptor;
import org.amelia.dsl.lib.descriptors.Host;
import net.sf.expectit.Expect;
import net.sf.expectit.matcher.Matchers;

public class SCPLogin {

   public static CommandDescriptor scpCommand(final String hostName,
       final String hostPassword, final String source, final String target) {
      return new CommandDescriptor.Builder()
          .withSuccessMessage("Copied file!")
         .withErrorMessage("File couldn't be copied!")
         .withCallable(new CallableTask<Object>() {
            @Override
            public Boolean call(Host host, String prompt, boolean quiet)
                throws Exception {
               final Expect session = host.ssh().expect();
               session.sendLine(
                   String.format(
                       "scp -r root@%s:%s %s",
                       hostName,
                       source,
                       target
                   )
               );
               try {
                  session.expect(
                      Matchers.regexp("Are you sure you want to continue connecting (yes/no)?")
                  );
                  session.sendLine("yes");
               } catch (IOException e) {
                   // Do nothing
               }
               session.expect(
                   Matchers.regexp(
                       String.format(
                           "root@%s's password:",
                           hostName
                       )
                   )
               );
               session.sendLine(hostPassword);
               session.expect(Matchers.regexp(prompt));
               return true;
            }
         }).build();
   }

}
```

Listing 5: Class for defining a user-defined AMELIA command.

With that in mind, we proceed to define a subsystem called *Allocation* that allows the transportation of compiled artifacts and dependency libraries to the computing nodes where they will be deployed, i.e., the subsystem allocates software artifacts to executing computing nodes. This subsystem uses the user-defined command described previously. Transportation would not be possible if compilation was not successful. AMELIA allows to model this requirement through the specification of dependencies between subsystems and rules. For dependencies between subsystems, it is necessary to specify the *depends on* clause in conjunction with the subsystem in need for indicating that the dependant subsystem would not instantiated until its dependencies complete their execution.

```
package co.edu.icesi.driso.matrices

import co.edu.icesi.driso.matrices.classes.SCPLogin
import java.util.List
import org.amelia.dsl.lib.descriptors.Host

includes co.edu.icesi.driso.matrices.Common

depends on co.edu.icesi.driso.matrices.Compile

subsystem Allocation {

   param List<Host> executionHosts

   on executionHosts {
      move:
         SCPLogin.scpCommand(
               compilationHost.hostname,
               compilationHost.password,
               builtFolder,
               allocationTargetFolder
         )
   }

}
```

Listing 6: Subsystem for artifacts allocation to computing nodes.

Once we granted the compilation and transportation of artifacts to the computing nodes that will execute them, we proceed to define two subsystems called *BlockReduce* and *Hybrid* that will execute their associated strategy.

3.1.4 Execution of BlockReduce strategy

```
package co.edu.icesi.driso.matrices
```

```

import java.util.List
import org.amelia.dsl.lib.descriptors.Host

includes co.edu.icesi.driso.matrices.Common

depends on co.edu.icesi.driso.matrices.Allocation

/*
 * Execute the BlockReduce multiplication strategy.
 */
subsystem BlockReduce {

   param List<Host> executionHosts

   var String common = "mcm-common"
   var String artifact = "mcm-blockreduce"
   var Iterable<String> libpath = #[
    'builtsFolder/common.jar',
    'builtsFolder/artifact.jar'
   ]

   on executionHosts {
      init:
            cd builtsFolder
   }

   on hosts.get("grid1") {
      reducer0: matrixBuilder;
         run "BlocksMultiplicator0" -libpath libpath

      control: reducer0, reducer1, reducer2, reducer3,
          reducer4, reducer5;
         run "Blockreduce" -libpath libpath
   }

   on hosts.get("grid2") {
      reducer1: matrixBuilder;
         run "BlocksMultiplicator1" -libpath libpath
   }

   on hosts.get("grid3") {
      reducer2: matrixBuilder;
         run "BlocksMultiplicator2" -libpath libpath
   }

   on hosts.get("grid4") {
      reducer3: matrixBuilder;
         run "BlocksMultiplicator3" -libpath libpath
   }

   on hosts.get("grid5") {
      matrixBuilder: init;
         run "FinalMatrixBuilder" -libpath libpath

      reducer4: matrixBuilder;
         run "BlocksMultiplicator4" -libpath libpath
   }

   on hosts.get("grid6") {
      reducer5: matrixBuilder;
         run "BlocksMultiplicator5" -libpath libpath
   }

}
```

Listing 7: Subsystem for executing BlockReduce strategy.

3.1.5 Execution of Hybrid strategy

```
package co.edu.icesi.driso.matrices

includes co.edu.icesi.driso.matrices.Common

depends on co.edu.icesi.driso.matrices.Allocation

/*
 * Execute the Hybrid multiplication strategy.
 */
subsystem HybridMultiplication {

   var String common = "mcm-common"
   var String artifact = "mcm-hybrid-multiplication"
   var Iterable<String> libpath = #[
         'builtsFolder/artifact.jar',
         'builtsFolder/common.jar'
   ]

    on hosts.get("grid5") {
        reducer0:
         run "BlocksMultiplicator0" -libpath libpath

        reducer1:
         run "BlocksMultiplicator1" -libpath libpath

        control: reducer0, reducer1, reducer2, reducer3,
            reducer4, reducer5;
         run "HybridMultiplication" -libpath libpath
   }
```

```

    on hosts.get("grid3") {
        reducer2:
            run "BlocksMultiplicator2" -libpath libpath
    }

    on hosts.get("grid7") {
        reducer3:
            run "BlocksMultiplicator3" -libpath libpath
    }

    on hosts.get("grid6") {
        reducer4:
            run "BlocksMultiplicator4" -libpath libpath
    }

    on hosts.get("grid4") {
        reducer5:
            run "BlocksMultiplicator5" -libpath libpath
    }

}
```

Listing 8: Subsystem for executing Hybrid strategy.

Finally, we proceed to define one custom deployment for each strategy to be deployed. The custom deployment includes the subsystems defined previously and contains control flow statements that execute the deployment of a set of subsystems in a particular way.

3.1.6 Custom deployment for the BlockReduce strategy

```
package co.edu.icesi.driso.matrices.deployments

import co.edu.icesi.driso.matrices.classes.Strategy
import java.util.List
import java.util.Map
import org.amelia.dsl.lib.descriptors.Host
import org.amelia.dsl.lib.util.Hosts

includes co.edu.icesi.driso.matrices.Common
includes co.edu.icesi.driso.matrices.Compile
includes co.edu.icesi.driso.matrices.Allocation
includes co.edu.icesi.driso.matrices.BlockReduce

/*
 * Deploy the BlockReduce strategy once and then
 * stop the executed components.
 */
deployment SimpleBlockReduce {
   // Load all hosts and then filter
   val Map<String, Host> hosts =
       Hosts.hosts("hosts.txt").toMap[h|h.identifier]
   val List<Host> executionHosts = #[
      hosts.get("grid1"), hosts.get("grid2"),
          hosts.get("grid3"),
      hosts.get("grid4"), hosts.get("grid5"),
          hosts.get("grid6")
   ]

   // Add subsystems to deploy
   add(new Common(hosts))
   add(new Compile(Strategy.BLOCK_REDUCE, hosts))
   add(new Allocation(executionHosts, hosts))
   add(new BlockReduce(executionHosts, hosts))

   // Deploy and then stop the executed components
   start(true)
}
```

Listing 9: Custom deployment for BlockReduce strategy.

3.1.7 Custom deployment for the Hybrid strategy

```
package co.edu.icesi.driso.matrices.deployments

import co.edu.icesi.driso.matrices.classes.Strategy
import java.util.List
import java.util.Map
import org.amelia.dsl.lib.descriptors.Host
import org.amelia.dsl.lib.util.Hosts

includes co.edu.icesi.driso.matrices.Common
includes co.edu.icesi.driso.matrices.Compile
includes co.edu.icesi.driso.matrices.Allocation
includes co.edu.icesi.driso.matrices.HybridMultiplication

/*
 * Deploy the HybridMultiplication strategy once and then
 * stop the executed components.
 */
deployment SimpleHybridMultiplication {
    // Load all hosts and then filter
    val Map<String, Host> hosts =
        Hosts.hosts("hosts.txt").toMap[h|h.identifier]
    val List<Host> executionHosts = #[
        hosts.get("grid3"), hosts.get("grid4"),
            hosts.get("grid5"),
        hosts.get("grid6"), hosts.get("grid7")
    ]

    // Add subsystems to deploy
    add(new Common(hosts))
    add(new Compile(Strategy.HYBRID_MULTIPLICATION,
        hosts))
    add(new Allocation(executionHosts, hosts))
    add(new HybridMultiplication(hosts))

    // Deploy and then stop the executed components
    start(true)
}
```

Listing 10: Custom deployment for Hybrid strategy.

References

[1] H. Lee, J. Kim, S. J. Hong, and S. Lee. Processor allocation and task scheduling of matrix chain products on parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):394–407, April 2003.