A Practical Guide for Using and Evaluating Amelia as a Language for Automating Software Deployment

Introduction

Software deployment is a post-development process that takes place within the software release management process [1]. More precisely, deployment starts once the software has been acquired by the *deployer*, after it has been developed, packaged, and published (i.e., advertised) by a software provider [5]. The deployment process is performed across a network of computing nodes partitioned into *sites* administered coordinately [2]. Each site contains a set of *resources* that enable the use of the software system. A resource refers to any artefact, either hardware or software, required by the software or one of its constituents in order to function, such as executable applications, data, memory, and disk space [3]. Deploying a software entails the transfer of the software's constituents from a *producer site*, to one or several *consumer sites* or target environments.

Amelia is a Domain Specific Language (DSL) that assists application developers in automating the deployment of distributed software systems. To evaluate the usability and other characteristics of Amelia, we developed an initial deployment case study, and an evaluation form. The deployment case study is explained in this guide to illustrate the main features of Amelia. By following this practical guide, a software *deployer* will be able to understand the constructs defined in Amelia, which will provide the deployer with a variety of tools for writing more effective deployment specifications with the language. Once this guide is completed, a software deployer will be asked to write an Amelia specification of a real-case deployment scenario. The deployment specification of this scenario will be performed through the realization of a guided session that we call *workshop*, where participants perform the deployment exercise and answer the questions of the evaluation form. This exercise will allow us to assess the effectiveness of Amelia in terms of *functional suitability*, *usability*, *reliability*, *productivity*, *expressiveness*, and *integrability* concerns.

The remainder of this document is structured as follows. Section 1 and 2 presents the initial case study for understanding the main features of Amelia. Section 3 depicts the real-case deployment scenario that will the drive the workshop. Finally, Section 4 presents the evaluation form for the workshop.

1 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 to solve 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 total number of 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 [4]. In this way, by combining the different and independent solutions to these subproblems, it is possible to configure several different actual solutions to the whole problem, which raises a problem of solution configuration, as depicted in Figure 1. 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 as the aforementioned, but this time reducing multiplication time. And of course, by combining the three subproblems both operations and overall processing time are reduced. In practice, however, there can be computational limitations and

trade-offs that may make it infeasible some of the possible solution configurations.

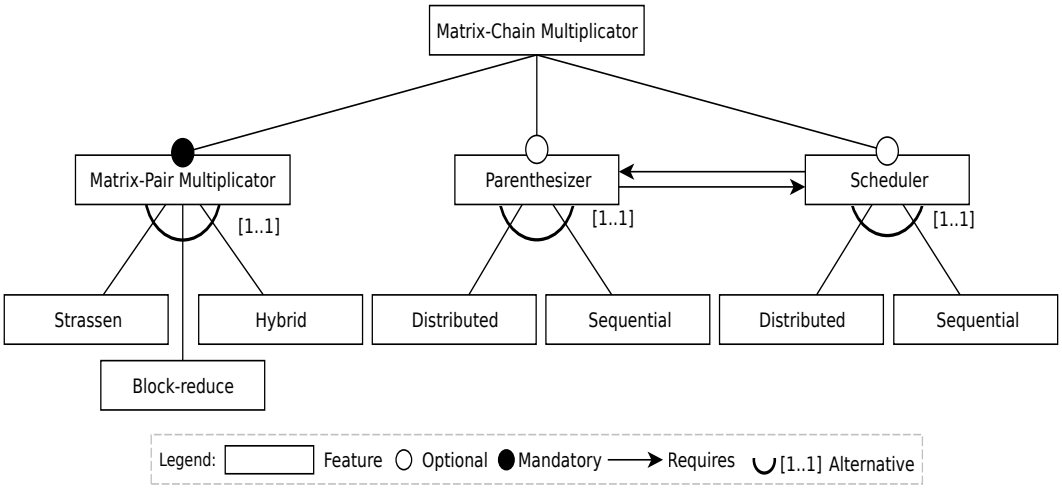

Figure 1: Feature diagram comprising the combinations that yield feasible configurations to solve the MCM problem.

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 paradigm, 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.

1.1 The BlockReduce Strategy

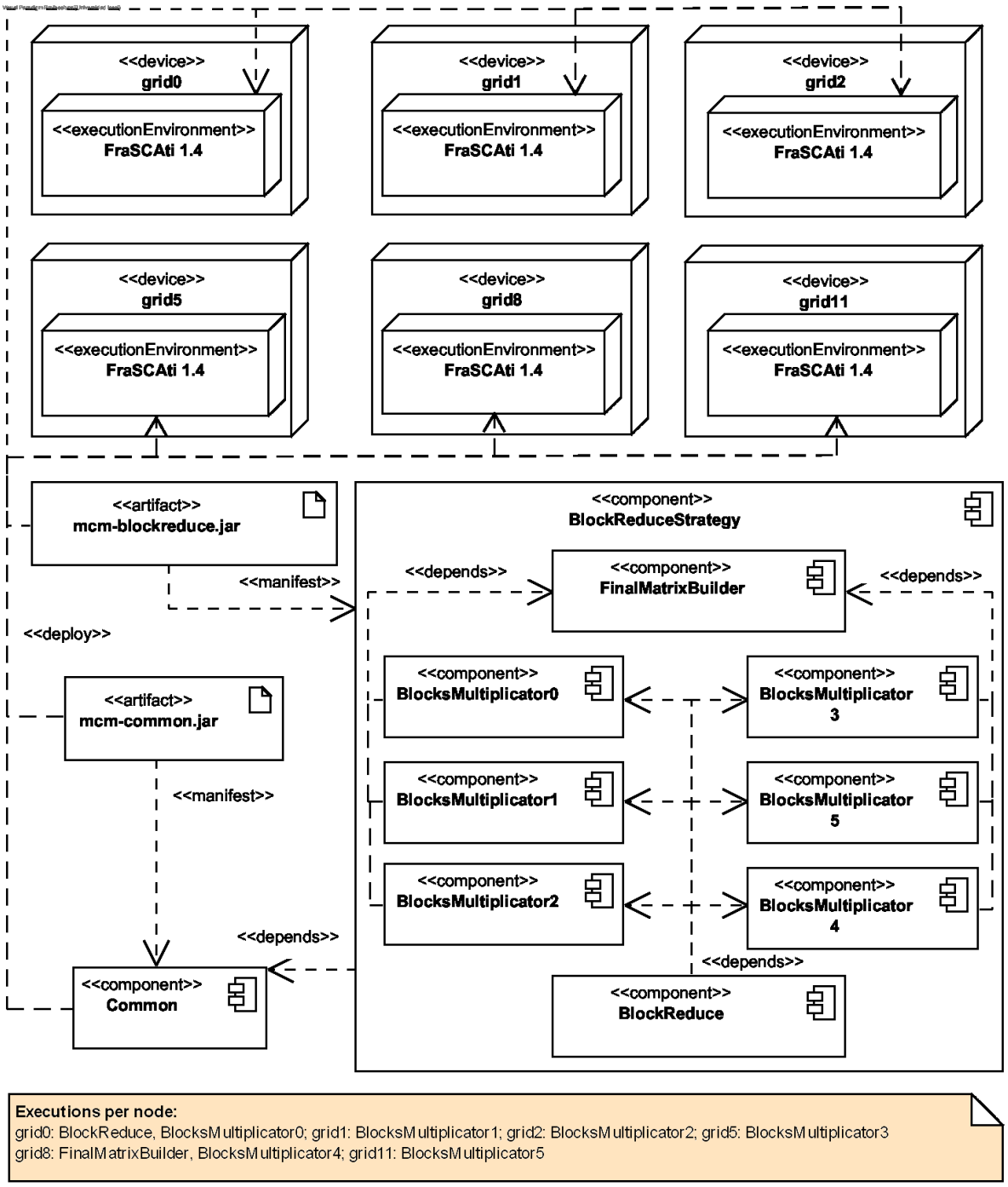

Figure 2: Deployment Diagram for the BlockReduce Configuration Strategy

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.

1.2 The Hybrid Configuration Strategy

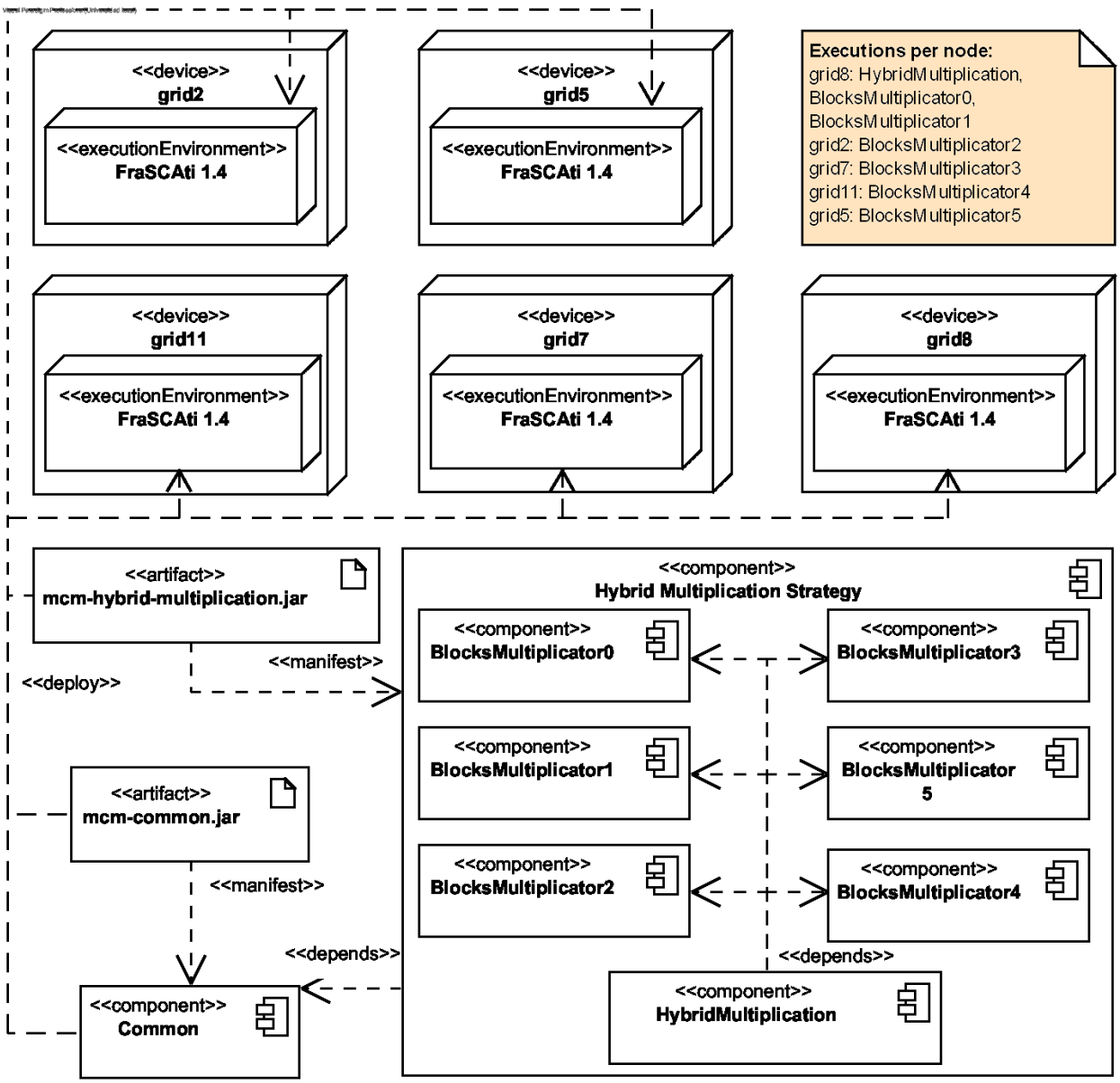

Figure 3: Deployment Diagram for the Hybrid Configuration Strategy

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.

2 Learning Amelia by Example

The following examples describe two deployment requirements and their corresponding solution in Amelia. These examples provide an overview of the language syntax and the associated semantics, which will guide the learning process of the language. Every deployment specification created for the examples are based on the computing nodes available in the software engineering laboratory, refered to as *LIASON*, at Icesi University.

Firstly, for a better comprehension of the subsequent examples, the main features of Amelia are depicted below.

2.1 Amelia basics

Amelia allows specifying two types of elements: Subsystems and Deployments.

2.1.1 Subsystems

A *subsystem* is a modular unit representing the operations required to deploy a cohesive set of software artefacts; it is composable and dependable by other subsystems, thus potentially reflecting the overall structure of the system to deploy. Privately-scoped variables, parameters, and execution rules can be defined within a subsystem. Parameters allow to configure a subsystem from a deployment strategy. Execution rules group commands in a logical way that describes the life cycle of a subsystem; that is, they represent the various phases of a subsystem's deployment.

2.1.2 Deployments

A *deployment* is an execution flow specification that dictates how to perform the deployment. For example, it allows to retry on failure or systematically repeat the same deployment, which is useful to warm up a system before running performance tests. Deployment strategies are simpler than subsystems. Their purpose is twofold: to configure subsystem instances and to determine how many times and including which subsystems is the system deployed.

2.2 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. In order to define the computing nodes to be used in the specifications, you may use the following file containing a mapping between the hosts in the deployment diagrams and the laboratory's computers (see Listing 9).

```
1 hgrid1  21  22  root  liasondriso_hgrid1  grid0
2 hgrid17 21  22  root  liasondriso_hgrid2  grid1
3 hgrid16 21  22  root  liasondriso_hgrid3  grid2
4 hgrid15 21  22  root  liasondriso_hgrid4  grid3
5 hgrid12 21  22  root  liasondriso_hgrid5  grid4
6 hgrid9  21  22  root  liasondriso_hgrid6  grid5
7 hgrid6  21  22  root  liasondriso_hgrid7  grid6
8 hgrid10 21  22  root  liasondriso_hgrid10 grid7
```

Listing 1: hosts.txt

2.2.1 Reusing code with Amelia

In order to reduce the size of the deployment specifications with Amelia and promote reuse and maintainability concerns, we begin the specification process with the identification and definition of elements that could be used repeatedly over this process. In this 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";

  /*
   * 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 = '«allocationTargetFolder»built-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 variable or rule declaration, string interpolations should be used instead of string concatenation.

2.2.2 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. This Enum will allow to refer to a specific deployment configuration in subsequent subsystems or deployments, especially when defining control flow.

```
1 package co.edu.icesi.driso.matrices.classes;
2
3 /**
4  * The mcm strategies.
5  * @author Miguel Jimenez (miguel@uvic.ca)
6  * @date 2017-08-19
7  * @version $Id$
8  * @since 0.0.1
9  */
10 public enum Strategy {
11     BLOCK_REDUCE,
12     HYBRID_MULTIPLICATION
13 }
```

Listing 3: Java Enum for possible deployment configurations.

We defined a subsystem called *Compile* for the compilation of the associated source projects (cf. Line ?? in Figure ??). 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*.

```
1 package co.edu.icesi.driso.matrices
2
3 import co.edu.icesi.driso.matrices.classes.Strategy
4
5 includes co.edu.icesi.driso.matrices.Common
6
7 /*
8  * Compile each strategy's source code.
9  */
10 subsystem Compile {
11     /*
12     * The multiplication strategy to compile.
13     */
14   param Strategy strategy
15
16   on compilationHost {
17     compileCommon:
18       cd commonSources
19       compile "src" "mcm-common"
20       cmd 'yes | cp -f mcm-common.jar «builtFolder»/'
21   }
22
23   on compilationHost ? strategy == Strategy.BLOCK_REDUCE {
24     compileBlockRStrategy: compileCommon;
25       cd blockRSources
26       compile "src" "mcm-blockreduce" -classpath '«builtFolder»/mcm-common.jar'
27       cmd 'yes | cp -f mcm-blockreduce.jar «builtFolder»'
28   }
```

```
29
30   on compilationHost ? strategy == Strategy.HYBRID_MULTIPLICATION {
31     compileHybridStrategy: compileCommon;
32       cd hybridSources
33       compile "src" "mcm-hybrid-multiplication" -classpath '«builtFolder»/mcm-common.jar'
34       cmd 'yes | cp -f mcm-hybrid-multiplication.jar «builtFolder»'
35   }
36
37 }
```

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

2.2.3 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. Nevertheless, 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.

```
1  package co.edu.icesi.driso.matrices.classes;
2
3  import java.io.IOException;
4  import org.amelia.dsl.lib.CallableTask;
5  import org.amelia.dsl.lib.descriptors.CommandDescriptor;
6  import org.amelia.dsl.lib.descriptors.Host;
7  import net.sf.expectit.Expect;
8  import net.sf.expectit.matcher.Matchers;
9
10 public class SCPLogin {
11
12   public static CommandDescriptor scpCommand(final String hostName,
13       final String hostPassword, final String source, final String target) {
14     return new CommandDescriptor.Builder()
15         .withSuccessMessage("Copied file!")
16       .withErrorMessage("File couldn't be copied!")
17       .withCallable(new CallableTask<Object>() {
18         @Override
19         public Boolean call(Host host, String prompt, boolean quiet)
20             throws Exception {
21           final Expect session = host.ssh().expect();
22           session.sendLine(
23               String.format(
24                   "scp -r root@%s:%s %s",
25                   hostName,
26                   source,
27                   target
28               )
29           );
30           try {
31             session.expect(
32                 Matchers.regexp("Are you sure you want to continue connecting (yes/no)?")
33             );
34             session.sendLine("yes");
35           } catch (IOException e) {
36             // Do nothing
37           }
38           session.expect(
39               Matchers.regexp(
40                   String.format(
41                       "root@%s's password:",
42                       hostName
43                   )
```

```
44                )
45            );
46            session.sendLine(hostPassword);
47            session.expect(Matchers.regexp(prompt));
48            return true;
49          }
50        }).build();
51    }
52
53 }
```

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

With the aforementioned class 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.

```
1  package co.edu.icesi.driso.matrices
2
3  import co.edu.icesi.driso.matrices.classes.SCPLogin
4  import org.amelia.dsl.lib.descriptors.Host
5
6  includes co.edu.icesi.driso.matrices.Common
7
8  depends on co.edu.icesi.driso.matrices.Compile
9
10 subsystem Allocation {
11
12   param Iterable<Host> hosts
13
14   on hosts {
15     transfer:
16       SCPLogin.scpCommand(
17               compilationHost.hostname,
18               compilationHost.password,
19               builtFolder,
20               allocationTargetFolder
21       )
22   }
23
24 }
```

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.

2.2.4 Execution of BlockReduce strategy

```
1  package co.edu.icesi.driso.matrices
2
3  import org.amelia.dsl.lib.descriptors.Host
4
5  includes co.edu.icesi.driso.matrices.Common
6
7  depends on co.edu.icesi.driso.matrices.Allocation
```

```
8
9  /*
10  * Execute the BlockReduce multiplication strategy.
11  */
12 subsystem BlockReduce {
13
14   param Iterable<Host> hosts
15
16   var String common = "mcm-common"
17   var String artifact = "mcm-blockreduce"
18   var Iterable<String> libpath = #[
19    '«builtsFolder»/«common».jar',
20    '«builtsFolder»/«artifact».jar'
21   ]
22
23   on hosts {
24     init:
25             cd builtsFolder
26   }
27
28   on hosts.get("grid1") {
29     reducer0: matrixBuilder;
30       run "BlocksMultiplicator0" -libpath libpath
31
32     control: reducer0, reducer1, reducer2, reducer3, reducer4, reducer5;
33       run "Blockreduce" -libpath libpath
34   }
35
36   on hosts.get("grid2") {
37     reducer1: matrixBuilder;
38       run "BlocksMultiplicator1" -libpath libpath
39   }
40
41   on hosts.get("grid3") {
42     reducer2: matrixBuilder;
43       run "BlocksMultiplicator2" -libpath libpath
44   }
45
46   on hosts.get("grid4") {
47     reducer3: matrixBuilder;
48       run "BlocksMultiplicator3" -libpath libpath
49   }
50
51   on hosts.get("grid5") {
52     matrixBuilder: init;
53       run "FinalMatrixBuilder" -libpath libpath
54
55     reducer4: matrixBuilder;
56       run "BlocksMultiplicator4" -libpath libpath
57   }
58
59   on hosts.get("grid6") {
60     reducer5: matrixBuilder;
61       run "BlocksMultiplicator5" -libpath libpath
62   }
63
64 }
```

Listing 7: Subsystem for executing BlockReduce strategy.

2.2.5 Execution of Hybrid strategy

```
1 package co.edu.icesi.driso.matrices
2
3 includes co.edu.icesi.driso.matrices.Common
4
```

```
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. As you can see in Listing 9, a deployment includes the subsystems defined previously and contains control flow statements that execute the deployment of a set of subsystems in a particular way.

2.2.6 Custom deployment for the BlockReduce strategy

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

import co.edu.icesi.driso.matrices.classes.Strategy
import java.util.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 BlockReduce {
  // Load all hosts and then filter
  val Map<String, Host> hosts = Hosts.hosts("hosts.txt").toMap[h|h.identifier]
  val 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  FraSCAti components
  start(true)
}
```

Listing 9: Custom deployment for BlockReduce strategy.

2.2.7 Custom deployment for the Hybrid strategy

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

import co.edu.icesi.driso.matrices.classes.Strategy
import java.util.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 HybridMultiplication {
    // Load all hosts and then filter
    val Map<String, Host> hosts = Hosts.hosts("hosts.txt").toMap[h|h.identifier]
    val 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.

2.3 Requirement 2

Specify the deployment of the BlockReduce strategy considering that every execution node must compile the sources that will run on it.

2.3.1 Redefine the custom deployments

Assuming that we can easily define an appropriate folder hierarchy in a convenient distributed file system (e.g., NFS), which can be accessed by every executing node for compiling the source code, we can take advantage of the multiple instances of a subsystem that a custom deployment can manage for meeting this requirement. For this, although it may be necessary to change some paths and subsystem dependencies, we essentially need to add the following lines to the custom deployment of the strategy.

```
1
2 //Number of execution nodes.
3 val Integer numNodes = 5
4
5 for(1:1..numNodes)
6 {
7 add(new Compile(Strategy.BLOCK_REDUCE, executionHosts.get(i)))
8 }
```

Listing 11: Defining custom deployment strategies.

3 Case Study: Distributed Processing of Large XML Files

3.1 Description

Carvajal Tecnolgía y Servicios (Carvajal, onwards) is a Colombian multinational company that offers outsourcing services for different industries in Latin America. Carvajal developed a solution for the National Agency for Overcoming Extreme Poverty (ANSPE) that allows census workers to collect and synchronize census data to families in the Colombian territory. Such solution requires daily data synchronisation from mobile devices to computing servers. This process had received several complaints due to severe delays and timeout errors at the final user device. Thus, we evaluated various distributed architectures by conducting experiments in a testing environment provided by the organization. We used Amelia to specify and deploy six architectural configurations, consisting of two variations of the Producer/Consumer design pattern and one of the Reactor design pattern, both in monolithic and distributed configuration deployments.

3.2 Deployment diagrams

The following deployment diagrams depict the high-level elements composing each of the deployment strategies for the Carvajal use-case. For each of these diagrams, please write the Amelia deployment specification and fill the evaluation form.

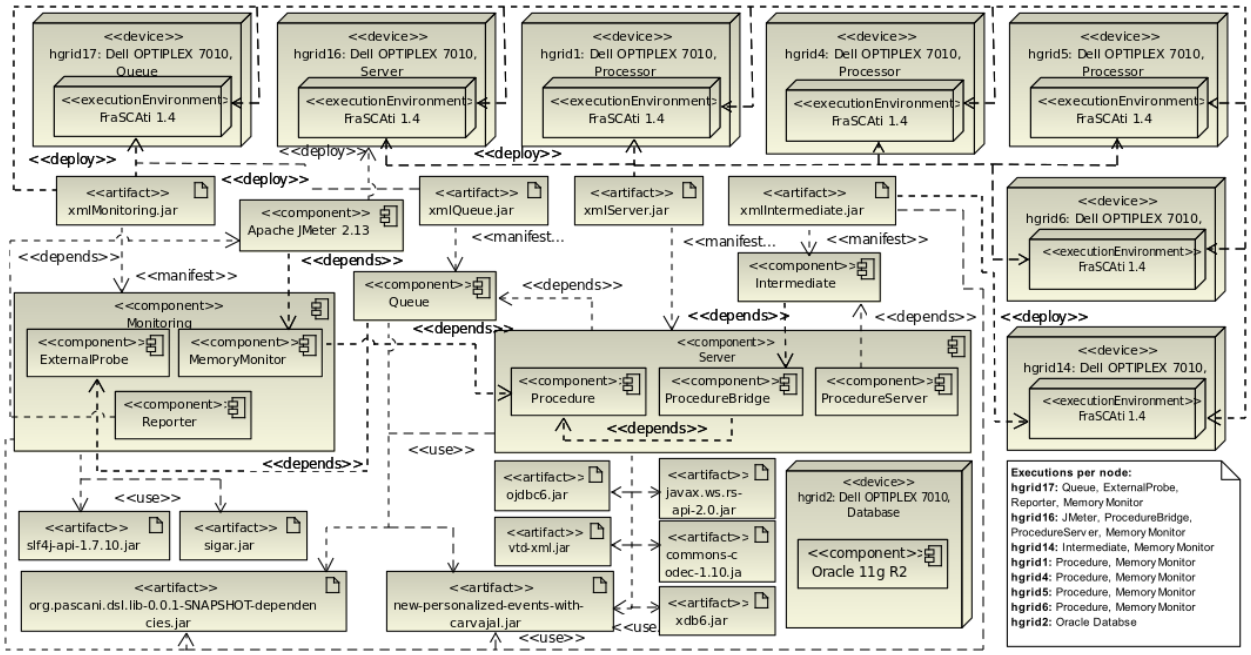

Figure 4: Deployment Diagram for the Producer/Consumer Configuration Strategy

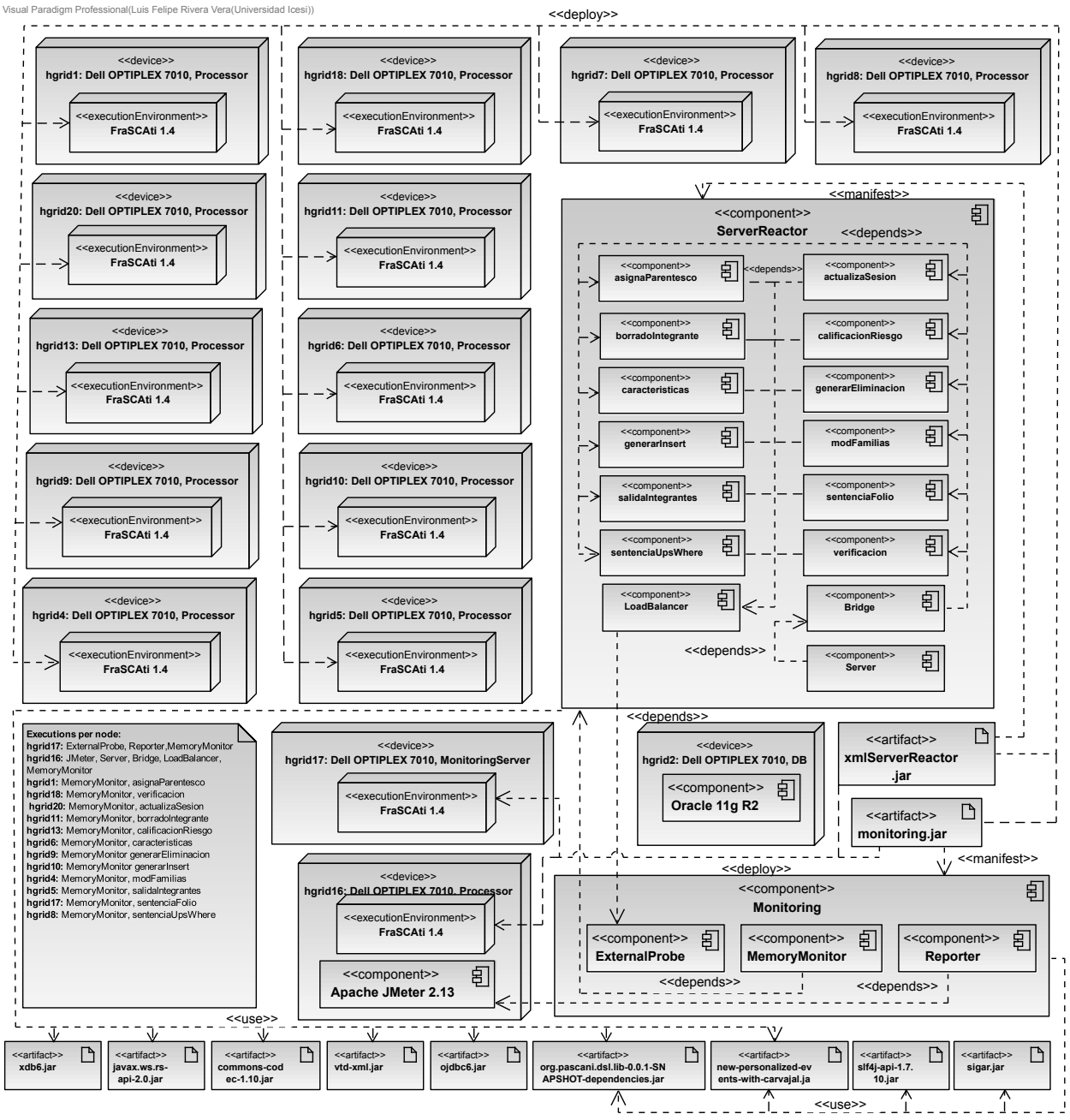

Figure 5: Deployment Diagram for the Reactor Configuration Strategy

4 User Questionnaire for AMELIA Evaluation

For the case study illustrated in Section 3, please fill out the questionnaire below.

1. Please indicate the time you spent (in minutes) specifying the deployment in AMELIA: _____

2. Please describe the difficulties you experienced developing the AMELIA deployment specifications.

_____

_____

_____

_____

_____

3. How much time have you been working as a professional Software Engineer? _____

4. Please describe how much experience you have deploying software.

_____

_____

_____

_____

_____

5. What type of software do you work on (e.g., monolithic software, distributed software, software in cloud, component-based software)?

_____

_____

_____

_____

_____

Please indicate your agreement or disagreement with the following statements by selecting only one square. The left-most square indicates that you **strongly agree**, while the right-most square indicates that you **strongly disagree**.

Functional Suitability

6. In the software deployment domain, all concepts and building-blocks to solve the addressed problem can be expressed in AMELIA. □—□—□—□—□

7. AMELIA is an appropriate and useful tool for deploying software. □—□—□—□—□

Usability

8. The language elements are understandable (*e.g.*, language elements can be understood after reading their descriptions). □—□—□—□—□

9. The concepts and symbols of the language resemble the terminology of the deployment domain, are learnable and rememberable (*i.e.*, learning easiness, easiness for developing deployment specifications). □—□—□—□—□

10. AMELIA helps users achieve their deployment tasks in acceptable development times. □—□—□—□—□

11. AMELIA is appropriate for the deployment of the type of software you work on. □—□—□—□—□

12. AMELIA has useful language elements to control the actual deployment operations (*e.g.*, language elements can be selected and put into practice easily, actions are undoable, error messages that explain recovery methods are available for controlling the deployment operations). □—□—□—□—□

13. AMELIA has a concise syntax that allows expressing deployment operations in short specification files. □—□—□—□—□

Reliability

14. AMELIA prevents making errors in deployment specifications. The language constructs helps the user to avoid and identify mistakes. □—□—□—□—□

15. AMELIA includes the right elements and correct relationships between them (it prevents unexpected interactions between its elements). □—□—□—□—□

Productivity

16. The development time of writing software deployment specifications is improved with AMELIA. □—□—□—□—□

17. AMELIA allows to compose and reuse language elements, which increases developer efficiency when creating new deployment specifications. □—□—□—□—□

18. AMELIA helps to improve the productivity of system deployment specifications. □—□—□—□—□

Expressiveness

19. A deployment strategy can be mapped into a AMELIA specification easily. □—□—□—□—□

20. AMELIA provides one and only one good way to express every concept of interest. □—□—□—□—□

21. Each AMELIA construct is used to represent exactly one distinct concept in the application domain. □—□—□—□—□

22. The language constructs correspond to significant application domain concepts. AMELIA does not include domain concepts that are not important. □—□—□—□—□

23. AMELIA does not contain conflicting or ambiguous elements. □—□—□—□—□

24. AMELIA is at the right abstraction level for writing deployment specifications, such that it is not more complex or more detailed than necessary. □—□—□—□—□

Integrability

25. AMELIA can be integrated with other languages used in the software development process, such as using already developed libraries. (*e.g.*, language integrability with other languages). □—□—□—□—□

References

[1] Alain Abran, James W Moore, Pierre Bourque, Robert Dupuis, and Leonard L Tripp. *Guide to the software engineering body of knowledge: 2004 version SWEBOK*. IEEE Computer Society.

[2] Antonio Carzaniga, Alfonso Fuggetta, Richard S Hall, Dennis Heimbigner, Andre Van Der Hoek, and Alexander L Wolf. A characterization framework for software deployment technologies. Technical report, DTIC Document.

[3] Alan Dearle. Software deployment, past, present and future. In *2007 Future of Software Engineering*, FOSE '07, pages 269–284. IEEE Computer Society.

[4] 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.

[5] Deployment OMG. Configuration of component-based distributed applications specification—version 4.0.