Deliverable D2.1

Programming Model Frameworks for Distributed High Performance Computing

VERSION Final revision 1.1
DATE April 20, 2011
EDITORIAL MANAGER Ronan Guivarch (Ronan.Guivarch@enseeiht.fr)
AUTHORS STAFF Julien Bigot LIP/INSA Rennes, Lyon (GRAAL team)
Frédéric Camillo IRIT/CNRS, Toulouse (GRID-TLSE project)
Ronan Guivarch IRIT/CNRS, Toulouse (APO team)
Aurélie Hurault IRIT/CNRS, Toulouse (ACADIE team)
Christian Pérez LIP/INRIA, Lyon (GRAAL team)
André Ribes EDF R&D
COPYRIGHT ANR COSINUS COOP. ANR-09-COSI-001-01
Programming Model Frameworks
for Distributed High Performance Computing

Julien Bigot, Frédéric Camillo, Ronan Guivarch, Aurélie Hurault,
Christian Pérez and André Ribes

Abstract

The present document is a report on the analysis of existing programming model
frameworks (PMF) developed by the partners of the project: GRID-TLSE (IRIT),
SALOME (EDF-CEA), ULCM and HLCM (INRIA). The main characteristics of these
PMF are presented as well as expected improvements in the framework of the COOP
project. They differ in their expressiveness but express common requirements with
respect to resource management systems.
Contents

1 Introduction 3

2 Grid-TLSE 4
   2.1 Overview of GRID-TLSE ............................................. 4
   2.2 Scenarios in GRID-TLSE ............................................. 4
      2.2.1 An example of a simple scenario: the Solve direct scenario ........ 4
      2.2.2 An example of a complex scenario: the “Decontamination” scenario .... 5
      2.2.3 Description of a scenario ....................................... 6
      2.2.4 The Weaver expert engine ....................................... 7
   2.3 Requirements of GRID-TLSE and how it could benefit from COOP ........... 7
      2.3.1 Range of parameters ........................................... 8
      2.3.2 Behavior of services ........................................... 8

3 SALOME 9
   3.1 Overview of SALOME ................................................ 9
   3.2 Creating a numerical simulation with SALOME .......................... 9
      3.2.1 The module YACS ................................................ 9
      3.2.2 SALOME component oriented programming model .................... 10
   3.3 Overview of the deployment process .................................. 11
   3.4 An example of application using the SALOME platform .................. 11
      3.4.1 Description of the problem ..................................... 11
      3.4.2 Integration in the SALOME platform ............................. 12
   3.5 Analysis .......................................................... 13
   3.6 Conclusion ....................................................... 13

4 Unified Lego Component Model 14
   4.1 Introduction ...................................................... 14
   4.2 Overview of ULCM specifications .................................... 14
   4.3 Overview of ULCMI, an ULCM implementation .......................... 15
   4.4 Conclusion ....................................................... 16

5 High Level Component Model 17
   5.1 Abstract model of HLCM ............................................ 17
      5.1.1 Structural elements of HLCM ................................... 17
      5.1.2 Specific elements of HLCM/CCM ................................ 20
      5.1.3 Behavior of HLCM Elements ..................................... 20
   5.2 HLCMI: an implementation of HLCM ................................ 21
   5.3 Deploying HLCM applications ....................................... 22
   5.4 Conclusion ....................................................... 22

6 Conclusion 23

ANR COSINUS COOP. ANR-09-COSI-001-01
1 Introduction

The COOP project aims to reconcile programming model frameworks (PMF) and resource management services (RMS) with respect to a number of tasks that they both try to handle independently.

The present document is a report on the analysis of existing PMF developed by the partners of the project:

- GRID-TLSE (IRIT),
- SALOME (EDF-CEA),
- ULCM and HLCM (INRIA).

The main characteristics of these PMF are presented as well as expected improvements in the framework of the COOP project.
2 Grid-TLSE

2.1 Overview of GRID-TLSE

The main goal of the GRID-TLSE Project (http://gridtlse.org) is to design an expert site that provides an easy access to a number of direct solvers for solving sparse linear systems, allowing their comparative analysis on user-submitted problems, as well as on matrices from collections also available on the site. The site provides for users assistance in choosing the right solver for their problems, appropriate values for the control parameters of the selected solver, or the suitable computer architecture. It is also a testbed for experts in sparse linear algebra. A computational Grid is used to deal with all the runs arising from user requests.

The expert site asks the user through a WEB interface (called WebSolve) to describe her problem as well as, optionally, the characteristics of the computers and the software that she plans to use. The expertise kernel (called Weaver) takes into account the user requirements, the internal expertise scenarios and the Grid state to build experience plans which are run using the DIET middleware [4] (http://graal.ens-lyon.fr/~diet/). The results and metrics are used to produce synthetic graphics which help the user in choosing the best tools — and the corresponding value of control parameters — or the best computer architecture for his problem (according to some metric e.g. minimizing execution time).

The GRID-TLSE project has been initially funded through the “Action Concertée Incitative” (ACI) French Research Program called “Globalisation des Ressources Informatiques et des Données” funded by the French Ministry of Research. It started in 2003. Currently, it is supported by the French research institute “Agence National de la Recherche” (ANR) through:

- the ANR-JST FP3C (Framework and Programming for Post Petascale Computing);
- the COOP project (ANR-09-COSI-001) funded by the ANR COSINUS program.

Previously, the GRID-TLSE project was part of other projects:

- the ANR LEGO project 2005-2009 (ANR-CICG05-11);
- the ReDIMSoPS project through the CNRS/JST (Japan) cooperation;
- the ANR SOLSTICE project 2007-2010 (ANR-06-CIS6-010).

2.2 Scenarios in GRID-TLSE

The expert site GRID-TLSE aims to compute responses on numerical problems (for example, what is the best solver and its parameters to solve a given linear system with the least memory consumption). The user can describe this problem following some given patterns. We call these patterns, scenarios.

Scenarios describe the sequence of operations that must be performed to achieve a request and are usually designed by expert users. The design of new scenarios is simplified by allowing a scenario to reuse — to call — existing ones. Once the user fills the scenario inputs (for example, matrix, computers, solvers), an expertise is created and can be launched.

This notion of scenarios and how they are managed is central in the reflexion of the ANR COOP.

2.2.1 An example of a simple scenario: the Solve direct scenario

Let consider the Solve direct scenario to illustrate this concept of a scenario (see Figure 1). In this scenario, for a given computer and a set of given matrices, we want to test all the available solvers.

The main steps of the scenario are the following:

**Step 1:** Get the solvers that can handle all the matrices regarding their properties (symmetry, type of its components, ...) and that are available on the chosen computer.
Step 2: Build the expertise; an expertise is a set of experiments. An experiment is a set of pairs \((\text{parameter}, \text{value})\), that can be modified during the execution of the scenario. At the beginning, it is the values of the inputs (matrix, computers, solver), that are used to generate the set of experiments.

Step 3: For each experiment, here the pair (matrix, solver), execute the treatment of the matrix for the given solver.

Step 4: Report metrics for all solvers (output of the scenario).

This scenario is static: the set of required experiments in order to perform the expertise does not depend on the results of some of others experiments.

Scenarios can also be dynamic, i.e. the results of some preliminary experiments are used to derive subsequent experiments.

2.2.2 An example of a complex scenario: the “Decontamination” scenario

What we call “decontamination” of a matrix in GRID-TLSE is the validation of the file that contains the matrix components and the computation of some algebraic and numerical characteristics. This decontamination is illustrated in Figure 2. This process is split in two phases (validation, characteristic computation); the computation phase can be express using an existing scenario called MA49BTF.

In this scenario, the MA49BTF scenario can be seen as a sub-scenario of the Decontamination scenario.
2.2.3 Description of a scenario

Figure 3: Scenarios in Grid-TLSE platform.

Scenarios are structured hierarchically in a dataflow like approach. Scenario inputs and outputs are connected to the sub-scenario inputs and outputs. A scenario can also contain internal links between sub-scenario inputs and outputs. A given scenario may then build several internal sets of experiments, execute them, and finally produce new sets depending on the results of the previous ones. Scenarios are therefore fully dynamic and may depend on the results of experiments in order to generate new experiments. In order to ensure that a scenario will stop, there must not be any internal cyclic links between sub-scenarios.

A scenario involving several steps is illustrated in Figure 4 which is the representation of the Decontamination Scenario.

Figure 4: Example of the description of an expert scenario corresponding to the Decontamination Scenario.

These steps are composed of elementary operators and sub-steps that require further decomposition:

- Step is composed of elements that are connected. They can be either simpler steps or operators. Each step has a goal and produces a set of results.
• **Operators** are the atomic elements. They perform operations on an experiment or a set of experiments. Currently, two types of operators are available: the transformation operator (OpTrans) — creation, modification, execution and filtering — and the execution operator (OpExec).

### 2.2.4 The Weaver expert engine

The Weaver expertise engine takes into account the user requirements, the internal expertise scenarios and the Grid state to build dynamic experience workflow. It proceeds in two steps: generating the workflow, and then executing it.

**Generating the workflow**  When an expertise is launched, Weaver gets the corresponding set of experiments (inputs, for example selected matrices, solvers, ...) in the database. Weaver associates this set with the scenario and produces a workflow (elementary operators, sub-steps).

**Executing the workflow**  Weaver then executes the workflow: on each flow (see Figure 5), there is a set of experiments that goes through the scenario elements:

- **OpTrans**: Weaver manipulates the flow. It can be a modification of each experiment (for example add, change or remove the value of a parameter), or a modification of the set of experiments (for example add or remove an experiment).
- **OpExec**: Weaver converts the experiment(s) into an appropriate format for an execution over the grid, and gives the information to a RMI server for an execution of the solvers via DIET.
- **Sub-scenario**: Weaver executes the sub-scenario as explained.

Weaver processes the results computed by the solvers. If it corresponds to an intermediate result, it is stored in the database and added in the experiment for the following treatments. Otherwise, they are sent back to WebSolve that reports them.

### 2.3 Requirements of GRID-TLSE and how it could benefit from COOP

The GRID-TLSE project uses a component-oriented approach. A high level semantic description of components has been defined to manipulate sparse linear algebra services (factorizations, orderings, linear solves, ...).
To provide an easy access to these services, GRID-TLSE uses the scenario concept which is a graphical description of the task workflow to perform. This description relies on a semantic description of sparse linear algebra services and tools.

The expertise engine (called Weaver) takes into account the user requirements, the internal expertise scenarios and the Grid state to build experience workflows which are run using the DIET middleware. These workflows can be dynamic since at each step we take into accounts the results of previous steps.

### 2.3.1 Range of parameters

The GRID-TLSE expert-site allows to test the impact of parameters values, for a given run, on a set of metrics (computational time, flops, ...). In this context, we can take advantage of a cooperation to adapt the scenario depending the available resources: for real parameters (such as threshold for example), we can adapt the variation interval and the number of values.

In the same scenario, if we can study the impact of many parameters, we can decide which ones are more important. Depending on the resources, we can choose to study all the parameters or only a restriction.

### 2.3.2 Behavior of services

Some services (solvers) available on GRID-TLSE expert site have an Out of Core (OOC) mode (independently from sequential or parallel mode). It means that they can consider data (matrices) that cannot be in random access memory but also that allows restarting, i.e. the possibility to stop computations at selected moments by saving data for a further restart.

We can give the example of the computation of the diagonal of the inverse of a $n \times n$ matrix, that consists, once the L.U factorization is performed, to the solution of $n$ independent linear systems. These $n$ solutions can be stopped and restarted if the resources are not sufficient. This kind of problem is an example of an astrophysics applications.

The expertise engine, when it runs the scenario, can after a cooperation, activates the Out of Core mode of one or many workflows and this, depending on available resources (time that we consider insufficient to compute in a non Out of Core mode, computer with not enough memory). The scenario execution will adapt, in this case, to the available resources obtained after cooperation.
3 SALOME

3.1 Overview of SALOME

SALOME [11] is a free software (distributed under the terms of the GNU LGPL) developed by EDF and CEA. It provides a generic platform for pre- and post-processing in the context of numerical simulation. It is based on a kernel and a set of standard modules:

- **GUI**: homogeneous user interface for SALOME, each module can integrate its own interface.
- **Geometry**: basic functionalities to create, import and correct any CAD models (IGES, STEP, BRep).
- **Mesh**: provides capabilities to mesh a CAD model using a standard meshing algorithm or any external mesher (as plugin).
- **YACS**: a way to describe a computational schema involving multi-solver coupling.
- **Post-processor**: viewer dedicated to the analysis of the results produced by solver computations (scalar, vectorial, 3D).

This platform can be extended by creating additional modules such as: meshing algorithms, standard or specific solvers, pre-processing module, etc. The goal is to be able to create specific platforms dedicated to a given field of research, such as nuclear computations, mechanics, etc. Figure 6 shows the main picture of the SALOME platform.

![Figure 6: SALOME platform overview.](image)

3.2 Creating a numerical simulation with SALOME

One goal of SALOME is to permit the design of numerical simulations. The platform provides a specific module named YACS for creating a calculation schema, and a parallel and distributed component oriented programming model for embedding user’s scientific codes.

3.2.1 The module YACS

The YACS [10] (dYnamic pArallel Coupling Supervisor) module is a tool to define and control execution of complex interconnected scientific applications on computer networks and clusters.
Interconnected scientific applications can be seen as a collection of computational tasks that are executed in a known order. Such a kind of application is described by a calculation schema. A calculation schema is mainly a graph of nodes that refers to computational tasks or control structures.

Nodes have control and data ports that are used to specify control and data flow. Nodes are connected by control and data links. A control link specifies that a node must be executed before another one. A data link specifies a data transfer between two ports with compatible types. Supported data types are basic ones (int, float, string, bool), sequence of basic types and references on remote objects (SALOME objects and CORBA objects). A node makes a treatment using incoming data provided by its input data ports and provides data to other nodes through its output data ports. There are two kinds of data ports: dataflow ports that are used only at the beginning and the end of the calculation, and datastream ports that can be used by a node during its execution.

Computational tasks are performed by two kinds of elementary nodes: inline nodes and service nodes. Inline nodes execute a Python script or a function directly in the YACS process. Service nodes execute a SALOME service in a remote process. Dataflow ports of inline nodes are mapped to input and output parameters of its function.

Composed nodes are used to modularize the calculation schema or to introduce control structures such as loop or switch:

• a block node can be used to define a sub-schema which control flow is a directed acyclic graph with parallel branches;
• a for loop node executes its internal node a given number of times;
• a while loop node executes its internal node while a condition is true;
• a switch node executes one of its internal nodes depending on the value of its selector port;
• a foreach loop is a special node that performs, in parallel, a given number of execution of its internal node. It has a special port that accepts a data list. Each item of the list is connected to one of the executing nodes. On output, data are gathered in special outgoing ports. This node has been designed mainly for parametric calculations.

A calculation schema can be built by using the Python interface directly, or by writing a file with XML syntax or interactively by using the module editor. When a schema is built, it can be executed in batch mode or in step by step mode by the YACS executor. The monitoring tool allows to inspect a running schema: state of nodes, for example, or to stop it (pausing to examine the current state) and then go step by step.

### 3.2.2 SALOME component oriented programming model

SALOME component model is composed of two layers. The first layer is an extension of the CORBA [8] distributed object model. It adds two entities: SALOME containers and SALOME objects.

- A SALOME object is a CORBA object with a new interface aimed at handling its life cycle and its integration into the platform.
- A SALOME container is a server of SALOME objects.

The framework uses the container’s interface to load, create or delete instances of SALOME objects. SALOME permits to create sequential or parallel components that are executed on sequential or parallel containers.

The second layer adds the notion of ports to SALOME objects which become a SALOME component. A port is a CORBA interface that a component uses or provides, which leads us to define two kinds of ports: uses ports and provides ports. A provides port implements the port’s CORBA interface and a uses port will use port’s CORBA interface implementation. This layer is named the Dynamic Software Component (DSC) extension.
Each method of a SALOME component (which is an object) can be exported into a YACS service. This mapping is described into a specific file that describes the services provided by a component. An instance of a SALOME component created by a SALOME container can provide one or many services in the same application.

3.3 Overview of the deployment process

The installation of SALOME is mainly a manual operation. The SALOME’s modules as well as the application components have to be installed on the machines and registered into a SALOME catalog. Machines that can be used by an application also need to be registered into a catalog — it includes information such as OS, memory, CPU, remote process creation (e.g. ssh), etc.

Currently SALOME can easily execute an application on interactive machines. The execution is done in two steps. First, the SALOME platform has to be launched on a local/front-end resource. Second, when starting the application, the YACS module launches SALOME containers on remote machines with the help of the container manager and the resource manager. Then, components are loaded into containers and connections are established. Let’s detail these three entities:

The resource manager selects a resource amongst those registered into its catalog. The set of resources can be filtered to match some characteristics of OS, CPU, memory, etc. and/or of SALOME components.

The container manager handles the localization or the launching of sequential or parallel container and the halting of containers. To fulfill these operations, it makes use of the resource manager. To start a new container, the container manager builds and submits a command line. Batch systems are not yet directly supported.

The supervisor (the YACS module) interprets a schema that contains information about computing nodes, abstract containers, and connections between computing elements. It is in charge of computing a mapping of the components onto containers and of handling control and data flows between computing elements. Therefore, it needs to determine how many containers it needs and on which nodes to start them. It makes use of the container manager to obtain containers compatible with the component type to instantiate.

In the case of an execution on a cluster managed by a batch manager (like PBS or LSF), SALOME provides a service to launch a YACS schema into a batch job. Inside the job, a new SALOME session is created and the YACS schema is launched with a resource catalog file provided by the job. Inside the job, these machines appears as interactive resources.

3.4 An example of application using the SALOME platform

In the framework of the 2006 French law on sustainable management of radioactive materials and waste, an evaluation of the industrial perspectives of minor actinides transmutation advantages and drawbacks in GENERATION IV fast spectrum reactors systems is requested for 2012. Recycling minor actinides would reduce the size of the final radioactive waste geological repository. This application use a 3D repository model in order to associate a canister and its residual power with the optimal area needed for its disposal. This model uses the thermal code SYRTHES [12] embedded in the SALOME platform, both codes being open source and being developed by EDF R&D.

3.4.1 Description of the problem

HLW (High Level Waste) disposal cells are dead-end, horizontal boreholes with an excavated diameter of 0.7 meter. A metallic sleeve supports the argillite and enables canisters handling for their introduction and positioning, and also for a possible retrieval, as shown in Figure 7. Their length is about 40 meters, but only the last 30 meters are used for package disposal, the first ten meters of the tunnel between the access gallery and the waste packages being used for sealing the cell by a clay plug held mechanically by concrete plug. Each disposal cell contains a single row of
3 to 18 waste packages (shown in Figure 8), depending on the repository size optimization under thermal constraints. Dummy packages are inserted between waste packages, in order to dilute the impact of waste canisters energy release on the Clay Layer temperature.

3.4.2 Integration in the SALOME platform
For a given HLW package, thus a given residual thermal power, the problem we face is an optimization under constraint problem. The function to minimize is the final storage area occupied by the package, which we will call $E$; this package area is the useful disposal cell area, depending of the distance separating the axis of two adjacent disposal cells, called $P_x$, divided by the number of canisters one disposal cell contains, called $N_c$. $E$ is given as a function of $P_x$ and $N_c$.

The pre-processing of this problem (geometry and meshing) can be summarized in a Python function whose arguments are the two parameters of optimization, $P_x$ and $N_c$. This script is inserted into a YACS schema that performs the optimization (See the results of one meshing in Figure 9). A part of the schema is showed in Figure 10. The different boxes are Python nodes that communicates with variables symbolized by the links between the nodes.

The final calculation scheme, which, for a bunch of different HLW canisters, with their number and thermal power, associated to a given scenario, gives back the total surface area needed for their disposal, with a distributed calculation on several CPUs.

Two parallel ForEach loops are used, one inside the other. The first ForEach loop works on the list of the different HLW canisters. For each type of canisters, several parallel computations are done to optimize the area under the constraints. Currently, the number of parallel branches is fixed by the user. It would be interesting to have dynamic information of how many machines are available to change dynamically the number of branches.
3.5 Analysis
Currently, the management of resources are only one way, from YACS to the resource manager. It would be interesting to understand what kind of information the resource manager could give to YACS and to the container manager, to have a better dynamic choice of resource. Furthermore, YACS interprets the schema without any look-ahead mechanism. Hence, it is very dynamic and it has no a priori knowledge of what it will need in the future. It means that all component mapping decisions are independent, which is not very efficient on heterogeneous machines. Moreover, information concerning communications between components is not available to the container manager.

3.6 Conclusion
SALOME is a generic open source platform that permits to perform numerical simulations. The platform is actually used to perform industrial numerical simulations. To achieve this, SALOME provides a parallel and distributed component oriented programming model and a tool named YACS that permits to create calculation schemes. Currently, the management of computational resources is working well for static applications and static resources. The next step is now to handle dynamic applications with dynamic resources.
4 Unified Lego Component Model

4.1 Introduction

Unified Lego Component Model (ULCM) [1, 13] is a component model that has been designed during the French ANR LEGO (2005-2009). ULCM is an abstract component model that extends classical component models with some novel features. By classical component models, we mean a composite-enhanced component model such as Fractal [3]. Abstract means that ULCM is not tied to a particular primitive component implementation so as to be able to study new relationships between components. To leverage this possibility, ULCM is based on a simple component syntax. The novel component model features studied within the ANR LEGO project were composition models based on data sharing, on the master-worker paradigm and on temporal dependencies.

After an overview of the specifications of ULCM, this section also presents ULCM, an implementation of ULCM which in particular is able to make use of Adage, an automatic deployment tool.

4.2 Overview of ULCM specifications

ULCM is a hierarchical component: a component can be either a primitive component or a composite component. A primitive component is implemented in another model such as Java, C++ or CCM. This part is not specified by ULCM— as it is abstract — and it depends on the implementations of ULCM. A composite component is implemented by an assembly of components. An originality of ULCM is that assembly can contain spatial and temporal relationships. Spatial relationships are expressed through spatial interactions — client/server and data-sharing ports (see below) — while temporal interactions are expressed with temporal ports as well as with workflow constructs that can be used in an assembly (see below).

```component aComponent {
  ports {
    server name=a_server_port type="interface1"
    client name=a_client_port type="interface2"
    server name=another_server_port type="interface3" multiple
    client name=another_client_port type="interface4" multiple
  }
  content { ... }
}
```

**Figure 11:** Example of client and server ports in ULCM.

The interactions of a component with other components are done through ports. A port contains a name, a kind of port and its parameter (usually a type). There are several kinds of ports in ULCM: client/server, attributes, data sharing and temporal ports. Client/Server ports enable interactions based on remote method invocation: a server port provides an object-like interface that a client port can use. Client and server ports can be multiple. Figure 11 shows an example of a component with client and server ports. Attributes are a specialized version of server port that provides getter and setter operation with respect to a data type. Note that attribute ports can not be connected: therefore, they are targeted to set up initial configurations.

Data-sharing ports enable a component to expose a data that it wants to share or to access a shared data provided by another component. From a programming point of view, these ports give access to the data through an interface that contains synchronization operations (acquire/release). It is up to an ULCM implementation to use an adequate data-sharing middleware to actually share the data between remote components.

Temporal ports can either be input port or output ports. These ports are associated to a data type and can be used in a workflow embedded into an assembly. ULCM’s workflow is an adaption to component models of AGWL [6]. The language is made of component oriented
instructions (component instance creation/removal, port connection/disconnection, attribute setting
and component execution) and control flow instructions (sequence, parallel section, condition, for,
and parallel for). Control flow instructions define blocks. Each block implicitly is a component
which may have ports. However, such ports are restricted to input and output ports. Blocks are not
declared as explicit component so as to avoid a verbose language. A graphical example is provided
in Figure 12.

ULCM also supports the concept of collection so as to easily implement master-worker rela-
tionships. A collection is a special kind of composite component: instead of containing a concrete
assembly, it contains an abstract assembly. The actual number of instantiation of its assembly is
left to the runtime — so as to implement adaptation policy — as well as the request transport
mechanism that determines to which instance invocations on server ports are transmitted.

4.3 Overview of ULCM, an ULCM implementation

ULCM is a proof-of-concept implementation of ULCM. It is based on JAVA. Figure 13 provides
an overview of ULCM. ULCM takes an ULCM program as input. An internal abstract representation
of it is built — based on ANTLR. Then, a home-made centralized workflow engine executes it.

The workflow engine first builds the initial state of the application, which is an assembly of
components. It instantiates them and then executes the workflow represented by the temporal
connection of ULCM. In the current version, each new component instantiation is handled before
the connection are analyzed. Therefore, advanced placement strategies are not yet supported.

ULCM supports five types of backends: simulation, JAVA, C++, CCM, and CHARM++. Simulation does not actually instantiates any component. It mainly targets to check the correctness
of an ULCM program. The JAVA and C++ are both local implementations of ULCM. Components are
instantiated within the same process as ULCM. Java Native Interface is used for C++ components.
The CCM and CHARM++ backends can handle distributed executions.

The CCM backend relies on ADAGE to deploy CCM components. Two distinct cases have to be
distinguished. The initial deployment is provided to ADAGE as a CCM CAD file. Therefore,
ADAGE could be able to take into account communication between components when computing
a placement. Dynamic component instantiations are given to ADAGE without any connection
information. Moreover, ADAGE is invoked for each component instantiation, even though one
ADAGE invocation could be done when instantiating a new composite as all initial sub components are known.

ADAGE is not very well optimized to work with batch systems. For example, with respect to OAR, the user has to provide ADAGE with a reservation ID as ADAGE is not able to directly perform a submission. However, CORDAGE [5], a upper layer abstraction layer towards resource management is able to handle resource management.

The last backend supported by ULCMi is CHARM++. ULCMi interacts with a CHARM++ application thanks to the Converse Client-Server API [9] of CHARM++. ULCMi makes use of the client-side API to send requests — such as component creation or connection — to a CHARM++ application. The CHARM++ application implements the server side operation needed by ULCMi. However, ULCMi does not launch the CHARM++ application; it is up to the user.

4.4 Conclusion

ULCM aims at increasing component model abstraction level for high performance computing by combining component, workflow, data sharing and skeleton concepts. ULCMi embeds an ULCM interpreter and the adequate runtime systems. It currently supports primitive components written in JAVA, C++, OMG CORBA component and CHARM++. With respect to deployment, JAVA and C++ components are deployed locally and supports multithreading, while CCM components are deployed thanks to the use of ADAGE and CHARM++ components are deployed by directly interacting with a deployed application. Hence, a user has to know in advance the needed resource to reserve them before launching its application.
5 High Level Component Model

5.1 Abstract model of HLCM

HLCM is an abstract component model that supports hierarchy, genericity and connectors. It provides a new way of describing component interface that enhances encapsulation and thus eases the replacement of their implementation while supporting efficient component implementations and interactions.

HLCM is abstract: it does not specify the primitive elements of the model (primitive component implementations, generators and port types which are introduced hereafter); primitive elements are instead specified by specializations of HLCM. This makes it possible to take advantage of HLCM using various underlying execution models or backends.

5.1.1 Structural elements of HLCM

The basis of HLCM is a generic hierarchical component model with connectors. The main elements of HLCM are components, connectors, port types, bundles and connection adaptors. Components and connectors are implemented respectively by component implementations and generators. The specificities of HLCM are open connections used to specify component interfaces and connection adaptors that support connection polymorphism.

The meta-model of HLCM is described in the Ecore language of the Eclipse Modeling Platform (EMF). As for instances of any Ecore meta-models, HLCM applications can be described in the OMG XML Metadata Interchange (XMI) dialect. This syntax is however not human-friendly; examples in this section are described in a dedicated HLCM textual syntax as well as in an informal graphical syntax.

Genericity has been introduced in HLCM using the approach described in [2]. All types of the model are generic (i.e. accept other types as parameter). An implementation of such a type can be for the whole generic type or it can be restricted to a given set of generic parameters. HLCM supports meta-programming with constructs such as static conditionals and loops evaluated at compilation time.

Components As usual in component models, a component in HLCM is a black-box, locus of computation. It exposes a set of named interaction points and has one (or more) implementation(s). Unlike in other component models however, these points of interactions are not ports but open connections that will be further discussed. There are two kinds of component implementations: primitive implementations and composite implementations. Primitive implementations are specific to each HLCM specialization; those of HLCM/CCM will be presented in Section 5.1.2. Composite component implementations are assemblies of component instances.

Connectors As in other models supporting connectors as first class entities, a connector in HLCM represents a kind of interactions. It exposes a set of named roles and has one (or more) implementation(s). Following the nomenclature defined in [7], connector instances are called connections and connector implementations are called generators. An example of connector is shown in Figure 14.

```java
class UseProvide<role user, role provider> {  
  // Connector implementation
}
```

*Figure 14: Declaration of a connector UseProvide with two roles — user and provider — to support Use/Provide interactions.*

Port types Ports in HLCM are instances of port types that are internal to primitive component implementations and which fulfill roles of connections. Port types are primitive and thus specific to each HLCM specialization; those of HLCM/CCM will be presented in 5.1.2.
component AComponent exposes {
    UseProvide<provider={Facet<A>}, user={}> ocA;
}

Figure 15: Example of a component exposing a connection ocA of type UseProvide whose role provider is fulfilled by a single port and whose role user is not fulfilled.

i) Textual representation

composite AComposite
    implements AComponent {
        AnotherComponent c1;
        AThirdComponent c2;
        merge (c1.ocB, c2.ocC);
        merge (this.ocA, c1.ocA);
    }

ii) Expanded representation (connections are not merged)

iii) Compact representation (connections are merged)

Figure 16: Three representations of a composite implementation of the component AComponent of Figure 15. It contains two internal component instances c1 and c2, that interact by merging the two open connections c1.ocB and c2.ocC, and exposes the open connection c1.ocA as its own ocA.

Connections  Interactions between component instances in assemblies are described by connections. A connection is an instance of a connector having each of its roles fulfilled by a set of ports. The types of these ports are implicit generic arguments of the generator implementing the connection.

Potential interactions of components are described by exposing (partially fulfilled) connections. An example of such exposition is illustrated in Figure 15.

Interactions in assemblies are described by merging two or more connections of the same type (connector). The result of a merge is a new connection. Each role of this new connection is fulfilled by the union of the sets of ports fulfilling this same role in the merged connections. An example of merge is illustrated by the composite component implementation described in Figure 16.

There are two kinds of connections in HLCM: closed and open connections. Closed connections are connections that can not be merged anymore: i.e. connections internal to an assembly. In order for a closed connection to be valid, there should exist at least one generator that can be used to implement it. Open connections are connections that are or can be further merged: i.e. connections exposed in the interface of components. In order for an open connection to be valid, it should be possible to construct another connection which it can be merged with to form a valid closed connection.

Generators A generator is an implementation of a connector. Multiple generators can implement the same connector. Generator can impose constraints on the generic parameters of the connector it implements (i.e. the type of the ports) as any implementation of a generic type. In addition, it can also impose constraints amongst a set specific to the specialization. This can for example be used to impose locality constraints on the component instances exposing the ports.

There are two kinds of generators: primitive generators and composite generators. Primitive generators are specific to each HLCM specialization; those of HLCM/CCM will be presented in Section 5.1.2.

A composite generator implements a connector with an assembly as shown in Figure 17. An assembly contains a set of component instances and connection merges. In addition, a composite generator can use ports fulfilling the roles of the connection it implements as fulfillments to roles of internal connections.
**Figure 17:** Example of a generator implementing the `UseProvide` connector when its role are fulfilled by CCM ports by inserting a proxy component for logging purposes. Ports fulfilling the roles of the `UseProvide` connections are used to fulfill roles of the `client` and `server` exposed connections using the `+=` operator.

```java
generator LoggingUP<UI,PI> implements 
    UseProvide<provider={Facet<PI>}, 
    user={Receptacle<UI>}> 
when ( UI super PI ) { 
    LoggerComponent<UI> proxy; 
    proxy.client.user += this.user[0]; 
    proxy.server.provider += this.provider[0]; 
}
```

**Figure 18:** Example of a bundle type. The `CcmPeer` bundle type contains two open connections: `pc` and `uc`. This makes it possible to implement a kind of peer-to-peer connection where each peer is both a provider and a potential user of a service.

```java
bundletype CcmPeer<I> { 
    UseProvide<provider={Facet<I>},user={}> pc; 
    UseProvide<user={Receptacle<I>},provider={}> uc; 
}
```

**Bundle types** Bundles are instances of bundle types that can be used similarly to ports to fulfill the roles of a connection. A bundle type is a set of named open connections that specifies the types of the connections of a bundle as shown in Figure 18. Bundles are instantiated in assemblies to logically group multiple connections of the underlying execution model that can not be independently connected. An example of a bundle instantiated in an assembly is illustrated in Figure 19.

A generator implementing a connection with a bundle port taking part in role fulfillments can use this ports as fulfillment of a role of some inner connections as with primitive ports. It can also explode the bundle and it can use its internal open connections like any other open connection.

**Connection Adaptors** Connection adaptors enable (open) connection polymorphism. A connection adaptor can adapt an (open) connection exposed by a component whose actual type does not match the type declared in the component interface. The definition of a connection adaptor is an assembly that uses the available connection and exposes a new connection of the expected type. An example is given in Figure 20.

The exposition by a composite of a connection whose actual type does not match the type declared in the component interface is only valid if there is an adaptor that supports it. The adaptor might however not be used in the case where a generator implements the connection without adaptation.

```java
composite Example implements AComponent { 
    AnotherComponent cmp; 
    this.peerConn.peer += CcmPeer<AnInterface> { 
        pc = cmp.provide; 
        us = cmp.use; 
    } 
}
```

**Figure 19:** The `CcmPeer` bundle type of Figure 18 is instantiated to fill the `peer` role of the `peerConn` connection exposed by the composite. Its internal open connections are set to those exposed by the `cmp` component instance.
adapter PushPull supports
UseProvide<user={Receptacle<Push>},
  provider=()} as
UseProvide<user=(),
  provider={Facet<Pull>}> 
{
  BufferComponent buffer;
  merge(buffer.pushSide, supported);
  merge(this, buffer.pullSide);
}

Figure 20: Example of connection adaptor describing how to adapt a UseProvide connection (supported) whose user role is filled by a Receptacle<Push> port as a UseProvide connection (this) whose provider role is filled with a Facet<Pull> port.

//OMG IDL3 annotated for HLCM/CCM
//@implements AComponent
component MyCcmImplementation {
  //@fulfills ocA.provider
  provides A a_port;
}

Figure 21: Example in extended OMG IDL of a (primitive) Ccm implementation of the component AComponent of Figure 15. The port a_port fulfills the role provider of the open connection ocA.

5.1.2 Specific elements of HLCM/CCM

Each specialization of HLCM has to specify the three primitive elements of the model: primitive component implementations, generators and port types. These elements can be defined by an equivalent element of the backend such as for components in HLCM/CCM. Otherwise, a finite set of primitive elements can be defined such as for connectors and port types in HLCM/CCM.

**Primitive Component Implementations**  HLCM/CCM primitive component implementations are Ccm components whose ports are used to fulfill the roles of the connections exposed by the HLCM component. These primitives component implementations can be for example defined in the OMG Interface Definition Language of Ccm annotated to specify the component implemented and the roles fulfilled by ports as shown in Figure 21.

**Primitive Port Type**  HLCM/CCM primitive port types are Ccm port types. Port types are however not a first class entity in Ccm; there is a finite set of types that can not be extended: facets, receptacles, event publishers, emitters and sinks. Therefore, there is only a fixed set of primitive port types in HLCM/CCM that match these types and that can not be extended by the user. These types are however generic and can be parametrized by CORBA object interfaces or event types.

**Primitive Generators**  HLCM/CCM primitive generators support the allowed connections between Ccm ports: Use/Provide interactions between facets and receptacles and event passing between event publishers or emitters and sinks. Similarly with port types there is only a fixed set of primitive generators in HLCM/CCM. These connectors simply model connections directly supported by the backend that do not require any information to be implemented.

5.1.3 Behavior of HLCM Elements

As previously explained, the specification of HLCM is based on a MDE approach and the behavior of HLCM applications is based on the specification of a model transformation that puts HLCM
applications in equivalence with applications of the underlying execution model. The behavior of an
HLCM application is defined as being that of the equivalent application of the underlying model.

An HLCM application is defined by the set of HLCM elements it contains: components, connectors, generators, port types and connection adaptors and by the component used as the
root of the application. To map it into a primitive application, it should be transformed into an
assembly which only contains primitive components, primitive ports, and primitive connections. In
the case of HLCM/CCM for example, this means that HLCM/CCM applications are mapped to
plain CCM applications.

As a first step of the transformation, the transformation required to support genericity and the
approach described in [2] is applied. The rest of the transformation is straight forward:

• the implementations of the various component instances and connectors are chosen amongst
  the available choices;
• in the case of composite implementations, their content is exposed so as to form a flat
  assembly.

The process is repeated until all elements are implemented. Since composite implementations are
opened and only their content is used, all elements have primitive implementations and thus form
an application of the underlying execution model.

This transformation is however non deterministic as it does not specify how the choice of
connection implementations is made. There can therefore be multiple distinct applications of the
underlying execution model in equivalence with a single HLCM application. In this case, each of
these primitive applications defines a valid behavior of the HLCM application.

The difficult part when implementing this algorithm lies in the choice of implementations for
components and connections. It is possible for a choice made at one step of the transformation to
lead to a state where no valid implementation is available for a given instance. This does not mean
however that different choices could not have lead to a valid application as a result.

Various answers can be given to this problem, such as enforcing stronger constraints on the
validity of implementations or supporting a rollback mechanism during the transformation for
examples. Until now this has not been a real problem in the examples we have worked with; however
it is an issue that we are working on.

5.2 HLCMi: an implementation of HLCM

HLCMi is a proof-of-concept implementations of HLCM that supports several primitive component
models. It currently supports component in three low level component models — where a component
is very close to a class: LLCMj for JAVA, LLCMCpp for C++ and LLCMCharm++ for CHARM++ —
and in CORBA Component Model (CCM).

HLCMi is written in JAVA. It relies on the tools provided as part of the Eclipse Modeling
Framework (EMF). It is built around two models described in the ECore using the EMPATIC
syntax: the HLCM Platform Independent Model (PIM) models HLCM assemblies as described
by the user and the HLCM Platform Specific Model (PSM) models concrete assemblies where all
choices have been made with only primitive components or connectors remaining. Specializations
that introduce primitive elements that can be described by the user such as primitive components
in HLCM/CCM must extend these models.

These implementations are themselves built as assemblies of the Low Level Component Model
JAVA (LLCMj), a plain JAVA backend for HLCM. In order to solve the bootstrap problem of
HLCMi, these assemblies are hard-coded in JAVA and do not take advantage of the HLCM assembly
language.

The architecture of a typical HLCMi specialization is described in Figure 22. A parser stage
takes as input the user-provided files and generates an instance of the PIM. The transformer
stage takes this instance as input and generates an instance of the PSM according to the algorithm
described in Section 5.1.3. This transformer also relies on a chooser to make the choices required
when multiple implementations of an element are available. Finally the PSM instance is used by a
backend that can either generate files or directly execute the application.
5.3 Deploying HLCM applications

HLCM is only able to deploy static applications. All application transformations have to be made before the application is launched. The transformation process relies on one or several choosers to actually make choices. However, some of these choices typically depend on resources. For example, a chooser may have to decide between a 32 bits or 64 bits processor architecture; another chooser may have to decide the number of instances of a replicated component, which typically depends on the number of nodes.

Three strategies can be envisioned for deploying HLCM applications. A first strategy consists in giving a priori resources to HLCM. This can be useful for machines with predefined queues of job. Therefore, if the selected queue is for 32 nodes, HLCM can be fed with this information. A second strategy is to let HLCM request resources from a resource management system based on criteria specified by the user. A third strategy is to let HLCM interact with resource management systems to be able to take into account resource availability and application specific characteristics.

Currently, HLCM supports only the first strategy.

5.4 Conclusion

HLCM is a component model that enables connectors in hierarchical component models. It introduces a novel way of expressing component interfaces based on the concept of open connections. It also introduces bundles and connection adaptors that support polymorphism of open connections.

The approach based on an abstract model and a transformation to an underlying execution model makes it possible to easily support multiple backends. As a matter of fact, only the primitive elements of the backend have to be described; the transformation is backend independent and the execution is directly handled by the backend.

A limitation of the choice of a model transformation however is that it limits HLCM to the expression of static assemblies that do not evolve during their execution. This does for example prevent the use of HLCM in its actual state for the implementations of workflows. However, an (efficient) implementation supporting it remains to be done. Centralizing the transformation phase will probably not be compatible with the very large scale required by upcoming exascale applications.

With respect to deployment, the transformation phase needs a close interaction with resource management systems in order to optimize application structure to available resources and to the user’s goals.
6 Conclusion

The report has presented four programming models — Grid-TLSE, Salome, ULCM and HLCM — and their associated frameworks. They differ in their expressiveness but express common requirements with respect to resource management systems.

GRID-TLSE needs to interact with resources in order to take account resource availability to optimize its execution of scenarios. SALOME also needs such information for static optimizations such as computing a good placement of components or to adapt the branching parameter of ForEach loop and for dynamic optimizations when dealing with its workflows. ULCM has requirements quite similar to SALOME as it also needs to optimize its initial placement as well as its workflow. HLCM currently only enables static choices but can have dramatic impact on an application as they concern component and connector implementations in addition to initial placement optimizations.
References


