Enabling Connectors in Hierarchical Component Models

Julien Bigot, Christian Pérez

HAL Id: inria-00456608

https://hal.inria.fr/inria-00456608v3

Submitted on 25 Aug 2010

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Un modèle de composants hiérarchiques avec connecteurs

Julien Bigot — Christian Pérez

N° 7204 — version 3

version initiale Février 2010 — version révisée Août 2010

Thème NUM

R apport de recherche

Un modèle de composants hiérarchiques avec connecteurs

Julien Bigot , Christian Pérez

Thème NUM — Systèmes numériques
Équipe-Projet GRAAL

Résumé : La croissance continue des capacités de calcul et de stockage permet aux applications numériques d'intégrer un nombre croissant de phénomènes dans leurs calculs au prix d'une complexité accrue. Les modèles de composants hiérarchiques apparaissent comme une approche intéressante pour gérer cette complexité. Cependant, définir et implémenter des interactions efficaces entre composants hiérarchiques est une tâche difficile, d'autant plus dans le cas d'applications parallèles et distribuées. Les connecteurs issus des langages de description d'architecture (ADL) offrent une solution prometteuse à ce problème. Il y a cependant des cas où une simple combinaison de la hiérarchie et des connecteurs dans un modèle de composants unique oblige l'utilisateur à faire un choix entre des mises en œuvres efficaces pour les composants et leur comportement « boîte noire ».

Ce papier décrit HLCM, un modèle avec connecteurs et hiérarchie qui fournit le concept de *connexions ouvertes* pour decrire les interfaces de composants. Ce méchanisme améliore l'encapsulation et facilite le remplacement des mises en œuvre de composant tout en permettant des interactions efficaces. Des interactions complexes telles que le partage de données ou les invocations de méthodes parallèles sont gérées avec succès par HLCM. Une mise en œuvre basée sur une transformation de modèle et sur CCM est utilisée pour illustrer sa faisabilité et ses bénéfices.

Mots-clés : Composants logiciels, Connecteurs, Hiérarchie, Calcul parallèle et distribué, Ingénierie dirigée par les modèles

Enabling Connectors in Hierarchical Component Models

Abstract: The continual growth of computing and storage capabilities enables scientific numerical applications to integrate more and more phenomena in their computations at the price of increased complexity. Hierarchical component models appear as an interesting approach to handle such complexity. However defining and implementing efficient interactions between hierarchical components is a difficult task, especially in the case of parallel and distributed applications. Connectors originating from Architecture Description Languages (ADL) offer a promising solution to this problem. There are however some cases where a simple combination of hierarchy and connectors in a single component model forces users to choose between an efficient implementation of components and their black box behavior.

This paper describes HLCM, a model with connectors and hierarchy that provides *open connections* as a mechanism to describe component interface that enhances encapsulation and eases component implementation replacement while supporting efficient interactions. Complex interactions such as data sharing and parallel method calls are successfully supported by HLCM. An implementation, based on model transformation and on CCM, illustrates its feasibility and benefits.

Key-words: Software Components, Connectors, Hierarchy, Parallel/Distributed Computing, Model-Driven Engineering

1 Introduction

Scientific numerical simulations offer interesting challenges from the software engineering point of view. Their software architecture is becoming more and more complex as the result of the coupling of multiple codes developed at different times by different teams. Moreover these codes make use of parallel constructs such as collective communications or parallel method calls. These applications also require computing power provided by complex hardware architectures such as highly parallel supercomputers or grids federating multiple clusters. With respect to the amount of computation to perform and the cost of the machine, efficiency is a very important criteria.

Component based software engineering (CBSE) [23] is an interesting approach to simplify the development of such complex applications. In this paradigm, pieces of code are embedded into components whose interactions with their environment are clearly identified, usually by a set of ports specifying both the services used and offered. Assembly of component instances connected through these ports are used to build larger grain components (composites) and to describe a whole application. These clear identifications of code interactions and dependencies improve code modularity and ease re-use.

The high efficiency required by scientific simulations does of course mean that programming models should introduce as few overhead as possible. This has been the goal of some specialized high performance component models such as the Common Component Architecture (CCA) [2]. In order to achieve high performance on a variety of hardware architecture it is however also highly important to use the best suited version of computation algorithms and communication patterns. That is why, porting an application to a new architecture —usually every few years— is still a very expensive and long task,

The expression of communication requirements at a higher level of abstraction such as with collective communications in MPI [15] makes it possible for the implementation to optimize their implementation depending on the available hardware resources. Propositions to introduce a set of higher level form of interactions between components have been made for example in the Grid Component Model (GCM) [7] and as extensions to other component models [8]. They include event passing, parallel method calls, master-worker relationships, collective communications among components, workflow-type interactions or data sharing between components for example.

The diversity of these extensions does however support the idea that there is probably not a fixed set of interactions that fits the needs of all applications; thus leading to the need to define an infinite of models. The concept of connectors as a first class entity originating from Architecture Description Languages (ADL) makes it possible to introduce new forms of interactions in a way similar to new components without needing new models. Connectors have been introduced in component models with their implementation either generated or chosen between various assembly of components [17]. As will be further explained in Section 2 however, in certain cases, to prevent performance penalties the interface of composites has to expose their content thus reducing the black box aspect of components and preventing their exchangeability.

This paper introduces a component model named HLCM that supports connectors, genericity and hierarchy. It avoids the limitations of existing models with connectors and hierarchy by expressing component interfaces with the

novel concept of *open connections*. This enhance encapsulation and thus ease the replacement of component implementation while supporting efficient implementations of components and interactions between them. Moreover, HLCM provides *bundle ports* and *connection transformers* that support polymorphism for open connections further increasing exchangeability of implementations. A prototype implementation as well as the analysis of the model on two use cases show the feasibility of the model.

The remainder of this paper is organized as follow. Section 2 introduces a synthetic application example that motivates our work and it analyzes the related work in light of this example. HLCM is described in Section 3. Section 4 describes the implementation of the motivating example using HLCM and uses it to evaluate HLCM. Section 5 draws some conclusions and presents some future works.

2   Context

This section presents two variations of a synthetic code coupling application that motivates this work that will be used through the paper. It is in particular used in this section to discuss the advantages and limitations of related work : component models with hierarchy, dedicated HPC interactions and connectors.

2.1   Motivating examples

The synthetic application described in this section is inspired by tight code coupling applications [22] found for example in image rendering [4], hydrology [14], quantum molecular dynamics [11], etc.

These applications are constituted by the coupling of multiple codes developed by different teams with different domain of expertise. The coupled codes may be sequential but they are often SPMD parallel codes. It is even possible for a single code to have multiple versions, some sequential, other parallel with distinct properties when it comes to memory, processing power or Input/Output requirements for example. In this case, the choice of the best suited version of each code for a given execution depends on available hardware resources.

To mirror these properties, the synthetic example application is the result of the coupling of two codes with a parallel and a sequential version each.

The interactions used to couple the codes of such an application can take various forms. We focus on two kinds of interactions that will lead to two variations of the synthetic example application.

The first example is a *coupling by method calls* between the codes. This is typical of applications [14, 11] where the codes alternate between a computing phase where a step of time is simulated and a communication phase where the modifications of the global state of the simulated space are exchanged. This information exchange is achieved through method calls between the parts of the application that describe change of the state in their parameters. In the case where some codes are parallel, this requires parallel to parallel method calls known as $M \times N$ method calls.

The second example is a *coupling by shared memory* between the codes. This is typical of applications [4] with irregular interaction patterns where the various codes access and modify portions of a single global state shared amongst them.

This can be achieved by providing a logically shared global memory with proper lock mechanism to ensure data integrity.

The most common approach used to develop these code (especially parallel codes) is to take advantage of programming environments such as MPI [15]. These environments are however usually targeted at specific kind of interactions (message passing in the case of MPI) that might not be what the other codes use. In addition these environments are not well suited to code coupling as they tend to favor hard coded and deeply hidden interactions between application parts that are very complex to modify.

2.2 Related work

The remaining of this section analyzes the answers provided to these challenges by various models that build on each other : component models, component models with hierarchy, with dedicated interactions and with connectors.

Component models The main element in component models are components with an external interface and an internal implementation. Component implementations are developed in an external programming models usually object-oriented or imperative. Component interfaces describe their interactions with the environment as a set of named ports that describe both the services they provide and use. These services take the form of an object interface that can be connected point to point to provide a method call semantic. Applications are build as sets of components interconnected through these ports. Examples of such models include the CCA [2], a process-local model designed for high performance applications and the CORBA Component Model (CCM)[19] based on CORBA to support distributed computing.

In the case of our motivating examples, each of the two coupled codes could be described as a component. By clearly identifying their interaction points, this ease the coupling of these components as well as their replacement by a different versions. The limitations of interactions to one-to-one use/provide interactions does however force the components to expose the way the higher level interactions (shared memory and parallel use/provide) are implemented, thus making complex the replacement of components.

Hierarchy Hierarchical assembling has been introduced in component models such as FRACTAL [13] or SCA [20]. This is achieved thanks to the concept of composites : components whose implementations are assemblies of interconnected component instances exposing some of their ports as ports of the composite. Hierarchy makes it possible to use components at multiple level of granularity.

In the case of our motivating examples hierarchy makes it possible to use components to describe both the whole application and the parallel parts of the application. At the whole application level, the application is seen as two components instances connected together. Each of these components can then be implemented as a composite containing a set of interconnected instances of a sequential part of the implementations.

However it does not make possible for interactions to be described at a higher level of abstraction ; it rather makes the problem worse as new kind of interactions might be required inside composites such as for example MPI-like interactions between the sequential parts of the parallel components.

Dedicated HPC interactions Dedicated HPC interactions have been introduced in some component models such as the *scattercast* and *gathercast* in GCM [7] (based on Fractal). Other interactions have been introduces as extensions to existing component models. For example, $M \times N$ (parallel-to-parallel) method calls from, to and between parallel components have been introduced as extensions to CCA in Scirun2 [2] and to CCM in GridCCM [21]. MPI-like collective communications have been introduced as an extension to CCM [9]. Data sharing between components has been proposed as an extension to CCA and CCM [5]. Finally, some interactions are supported as part of more generic extensions such as the master/workers paradigm [12], and more generally (parallel) algorithmic skeletons [1].

These models and extensions support the various interactions required in our motivating examples. The support of $M \times N$ method calls by GridCCM makes it possible to replace a sequential version of a component by a parallel version without change of interface. Similarly the extensions providing shared data between components works whatever the number of components taking part in the interaction is. There is not however a single component model that supports all these interactions. Some more application-specific interactions might not be available in any model at all.

Component models with connectors The concept of connector originates from ADLs and have been introduced in component models such as the Sofa component model [6] and in [17]. Connectors are first class entities similarly to components that contain roles (or plugs) fulfilled by ports of component instances so as to describe their interactions [18]. Unlike components, connectors are intrinsically generic and their implementation can vary in function of the quantity, type and locality of the ports taking part in the connection.

In the case of our motivating examples, connectors make it possible to add support for the various interactions required in the same way as new components. A connector with two roles, `user` and `provider` could support $M \times N$ method calls and connectors with a single role each could support shared memory and MPI-like communications.

In the case of the parallel version of the codes however some problems arise. For example, in the case of the coupling by shared memory, two approaches for the implementation of the parallel version of the codes can be taken, none of which is satisfactory. The first approach consists in letting the composite to expose the ports of their inner instances in their interface as shown in Figure 1. This solution breaks the black box behavior of the components and prevents their easy replacement as it leads to a distinct interface for the parallel and sequential versions of components.

The second approach prevents this problem by inserting a component responsible for interface adaptation inside the composite as shown in Figure 2. This second option can however lead to severe performance degradation as it implies that interactions between the two parallel components have to go through a single process that can present a bottleneck.

2.3 Discussion

Software component models provide an interesting approach for the development of code coupling application thanks to the black box behavior of com-
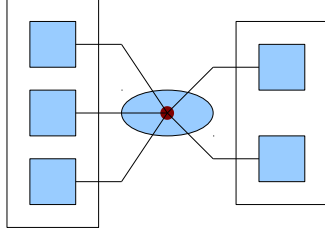
FIGURE 1 – A first approach for the coupling of two parallel codes in models with connectors.
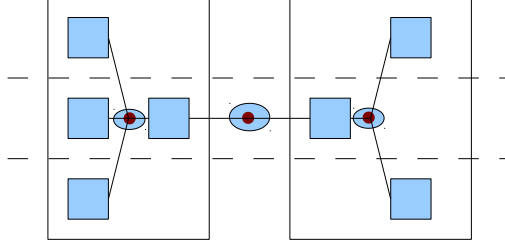

FIGURE 2 – A second approach for the coupling of two parallel codes in models with connectors.

ponents that eases their replacement. Hierarchy is interesting as it makes it possible to describe reusable composition of component instances such as parallel components. Dedicated interactions provided either as a fixed set or more interestingly through connectors additionally allow to change the implementation of not only components but also interaction implementations allowing for better performance on various hardware resources.

There are however some cases where a simple combination of hierarchy and connectors in a single component model forces users to choose between efficient implementation of component and the black box behavior that makes exchangeability possible. The next section will focus on the description of a component model supporting both aspects thanks to a novel way of expressing component interface.

3   HLCM : a High Level Component Model

This section introduces HLCM, an abstract component model that supports hierarchy, genericity and connectors. HLCM 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*.

This paper takes the example of HLCM/CCM, a specialization that uses the elements of the CORBA Component Model (CCM) as its primitive elements. Other specializations for JAVA, C++ and charm++ [16] exist but are not described in this paper.

This section first describes the structural elements of HLCM and then the elements specific to the HLCM/CCM specialization. The behavior of HLCM applications is specified through an equivalence with applications of the underlying execution model.

```
connector UseProvide<role user, role provider>;
```

FIGURE 3 – Declaration of a connector `UseProvide` with two roles —`user` and `provider`— to support Use/Provide interactions.

3.1   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 has been 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 [10]. All types of the model are generic (*i.e.* accept other types as parameter). The implementations of these types can either implement the whole generic type or 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 3.2. **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 [17], connector instances are called connections and connector implementations are called generators. An examples of connector is shown in Figure 3.

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 3.2.

```
component MyHlcmComponent exposes {
  UseProvide<provider={Facet<A>}, user={}> ocA;
}
```

FIGURE 4 – 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.

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 4.

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 5.

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 with which it can be merged 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 3.2.

A **composite generator** implements a connector with an assembly as shown in Figure 6. An assembly contains a set of component instance 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.

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 7. Bundle are instantiated in assemblies to logically group multiple
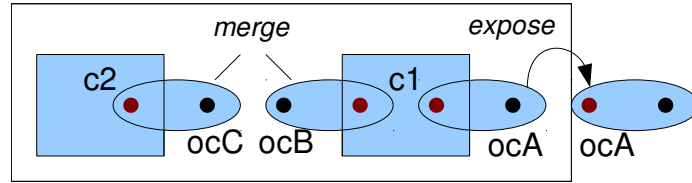
i) Textual representation

```
composite MyCompositeImplementation
    implements MyHlcmComponent {
  AnotherComponent c1;
  AThirdComponent c2;
  merge ( c1.ocB, c2.ocC );
  merge ( this.ocA, c1.ocA );
}
```

ii) Expanded representation (connections have not been merged)

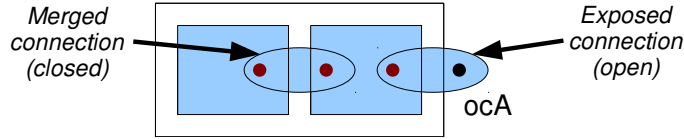*iii) Compact representation (connections have been merged)*

FIGURE 5 – Three representations of a composite implementation of the component `MyHlcmComponent` of Figure 4. It contains two internal component instances `c1` and `c2`, that interacts by merging the two open connections `c1.ocB` and `c2.ocC` and exposes the open connection `c1.ocA` as its own `ocA`.

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 8.

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, for example by merging it.

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 9.

The exposition by a composite of an 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.

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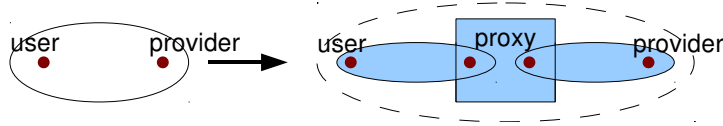

FIGURE 6 – Example of a generator implementing the `UseProvide` connector when its role are fulfilled by CCM ports by inserting a proxy component for logging purposes. Ports fulfilling the roles of the `UseProvide` connections are used to fulfill roles of the `clientSide` and `serverSide` exposed connections using the $+=$ operator.

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

FIGURE 7 – 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.

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

FIGURE 8 – The `CcmPeer` bundle type of Figure 7 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.

3.2 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

```
adaptor PushPull supports
UseProvide<user={Receptacle<Push>},provider={}>
as UseProvide<user={}, provider={Facet<Pull>}>
{
  BufferComponent buffer;
  merge(buffer.pushSide, supported);
  merge(this, buffer.pullSide);
}
```
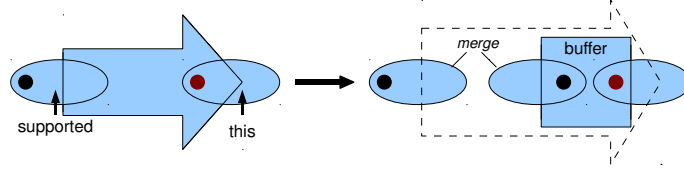

FIGURE 9 – Example of connection adaptor describing how to adapt a `UseProvide` connection (`supported`) whose `user` role is filled by a `Receptacle<Push>` port as a `UseProvide` connection (`this`) whose `provider` role is filled with a `Facet<Pull>` port.

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

FIGURE 10 – Example in extended OMG IDL of a (primitive) CCM implementation of the component `MyHlcmComponent` of Figure 4. The port `a_port` fulfills the role `provider` of the open connection `ocA`.

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 10.

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. There is therefore 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 parameterized 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 therefore only a fixed set of primitive generators in HLCM/CCM. These connectors simply model connections directly supported by the backend that do not require anymore information to be implemented.

3.3   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 [10] 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 define 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 choice 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.

3.4   Discussion

This section has presented HLCM, an abstract component model, and HLCM/CCM, a specialization of HLCM that uses CCM as a backend. HLCM 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 behavior of HLCM applications has been defined by equivalence with applications of the underlying execution model obtained with a straightforward transformation.

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 dataflows and workflows.

The concept of open connection makes it possible for connections to cross the frontier of composites. It is possible in HLCM for a single connection to have multiple roles fulfilled inside a given composite and other roles fulfilled in another composite. The next section will show how this and open connection polymorphism make it possible to elegantly support the motivating examples described in Section 2.1 easing the replacement of their implementation while supporting efficient implementations and interactions.

4   Evaluation

This section evaluates HLCM by using HLCM/CCM to implement the two motivating examples introduced in Section 2. It starts by presenting $HLCM_i$, a framework supporting the implementation of HLCM specializations and $HLCM_i$/CCM, a prototype implementation of HLCM/CCM based on this framework. Then it describes the implementation in HLCM/CCM of our two motivating examples : the coupling of parallel codes through shared memory and through parallel method calls. Finally it analyzes these two implementations and compares their behavior to implementations that do not take advantage of HLCM.

4.1 $HLCM_i$/CCM : an HLCM/CCM implementation

In order to evaluate HLCM and its various specializations, a set of proof-of-concept implementations have been developed. These implementations are themselves build as assemblies of the Low Level Component Model JAVA ($LLCM_j$), a plain JAVA backend for HLCM. In order to solve the bootstrap problem, these assemblies are hard-coded in JAVA and do not take advantage of the HLCM assembly language. The components shared by the various implementations form a framework known as $HLCM_i$.

$HLCM_i$ relies on the tools provided as part of the Eclipse Modeling Framework (EMF). It is build around two models described in the ECORE using the EMFATIC syntax. The HLCM Platform Independent Model (PIM) models HLCM assemblies as described by the user ; it contains 59 classes described in
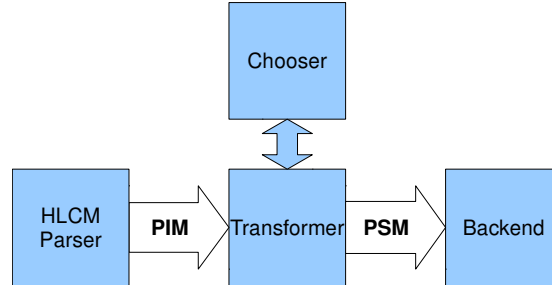
FIGURE 11 – Architecture of typical HLCM$_i$ based compiler.

around 300 lines in EMFATIC. The HLCM Platform Specific Model (PSM) models concrete assemblies where all choice have been made with only primitive components or connectors remaining ; it contains 33 classes described in around 150 lines in EMFATIC. Specializations that introduce primitive elements that can be described by the user such as primitive components in HLCM/CCM must extend these models ; the modeling of HLCM/CCM primitive components requires 6 classes and 30 lines of EMFATIC.

The architecture of a typical HLCM$_i$ specialization is described in Figure 11. 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 3.3. 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.

In HLCM$_i$/CCM, the `parser` stage is made of two components implemented using the XTEXT framework. The first component parses the HLCM assembly files ; the second —specific to HLCM$_i$/CCM— parses the extended CCM IDL.

The `transformer` stage is not specific to HLCM$_i$/CCM ; it is implemented in pure JAVA and requires around 2000 lines of code. Languages dedicated to model transformations are available in EMF ; however they were not mature enough when this code has been written.

For the `chooser` stage, various kinds of heuristics can be implemented. For the experiences of this paper however, a very simple chooser that is limited to making choice dictated by the user or random choice otherwise has been used.

The `backend` stage is made of an HLCM$_i$/CCM specific component that dumps the HLCM/CCM PSM instance as a CCM Component Assembly Descriptor (CAD) file.

This implementation makes it possible to easily experiment with HLCM/CCM. It can be seen as a black box that takes as input a set of CCM components and HLCM files and generates a CCM CAD as output. As of now this implementation does not automatically take resources into account. It does however make it very easy to change the implementation chosen for one of the HLCM components to obtain a completely different CCM CAD as a result as will be seen in the remaining of this section.

```
connector SharedMem<role access>;
```

FIGURE 12 – Declaration of the `SharedMem` connector with a single role `access`.

```
interface DataAccess {
  DataPointer get_data();
  long get_size();
  ...
}
```

FIGURE 13 – Declaration in OMG IDL of the DataAccess interface.

```
component MemoryAccessor exposes {
  SharedMem<access={LocalReceptacle<DataAccess>}>
      memory;
}
```

FIGURE 14 – Declaration of a component `MemoryAccessor` exposing a connection `memory` that can be used to access the shared memory.

4.2 Implementation of the motivating examples in HLCM/CCM

In order to evaluate HLCM, the motivating examples described in Section 2.1 has been implemented in HLCM/CCM. This section focusses on the implementations of the two kinds of interaction used in this examples : interaction by shared memory and by (parallel) method call.

Shared memory interaction The shared memory interaction has been implemented using an approach inspired by [5]. It is supported by `SharedMem`, a connector with a single role : `access` described in Figure 12.

An object interface `DataAccess` whose IDL specification is shown in Figure 13 is dedicated to the access to the shared memory. This interface is not aimed to be remotely accessed as the opaque valuetype `DataPointer` is used to manipulate a pointer to the actual data. As a result this interface can correctly be used only by components located in the same address space (process).

The HLCM/CCM backend has been extended with two additional primitive ports : `LocalReceptacle` and `LocalFacet` to support this use case. These two port types behave exactly like the usual CCM `Receptacle` and `Facet` except that they impose a process collocation constraint between the involved component instances in the resulting CAD.

A component can therefore safely access the shared memory through a port of type `LocalReceptacle<DataAccess>` used to fulfill the `access` role of a `SharedMem` connection as shown in Figure 14.

A composite component implementation containing multiple instances of the `MemoryAccessor` component as shown in Figure 15 can both specify that i) its internal instances access the same shared memory and ii) external members to the composite can access the same shared memory by exposing the connection.

i) Textual representation

```
composite CompositeAccessorImpl
    implements MultiAccessor {
  MemoryAccessor c1;
  MemoryAccessor c2;
  merge(this.memory, c1.memory, c2.memory);
}
```
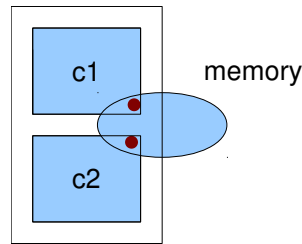
ii) Graphical representation

FIGURE 15 – The `CompositeAccessorImpl` composite merges the `c1.memory` and `c2.memory` into a single connection exposed as `this.memory`.

```
generator LocalSharedMem<Integer N>
    implements SharedMem<access=each(i:[1..N]){
        LocalReceptacle<DataAccess>}>
{
  LocalMemoryStore<N> store;
  each(i:[1..N]){store.access[i].user+=access[i];}
}
```

FIGURE 16 – Definition of the `LocalSharedMem` generator supporting local `SharedMem` connections. Its implementation relies on an instance of a `LocalMemoryStore` component that embeds the data accessed by all components.

Two generators implementing this connector have been developed. A first generator simply inserts an internal component in charge of the memory is described in Figure 16. This implementation imposes a process collocation constraint between each component instance accessing the connector and the `store` instance, effectively requiring all accessors to be in the same process.

A second generator supports distributed shared memory by inserting for each accessor an instance of a component delegating the calls to JuxMem [3], a distributed shared memory implementation. It is not shown by lack of space but it does not contain any originality.

Parallel method call interaction Unlike shared memory, the support for parallel method calls [21] does not require the introduction of a new connector : the `UseProvide` connector already supports method calls. Two new bundles are however introduced : `ParallelFacet` whose definition is presented in Figure 17

```
bundle ParallelFacet<Integer N, interface I> {
  each(i:[1..N]){
    UseProvide<provider={Facet<I>}> part[i];}
}
```

FIGURE 17 – Definition of the `ParallelFacet` bundle port type. It contains N `UseProvide` connections called `part` whose `provider` role if fulfilled by a `Facet<I>` port.
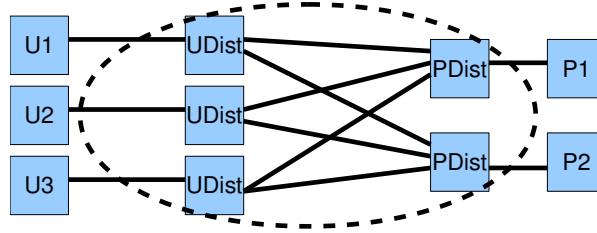

FIGURE 18 – A parallel `UseProvide` connection implemented by the `MxN` generator. A proxy instance is inserted for each participant. Each proxy instance is connected to all those of the opposite side.

and the symmetrical `ParallelReceptacle`. We make the choice that supporting parallel method calls means supporting the cases where these bundles fulfill the `user` or `provider` roles of a `UseProvide` connection.

A `MxN` generator that implements `UseProvide` connections whose roles are fulfilled by a `ParallelFacet` and a `ParallelReceptacle` has been implemented. An example of connection implemented by this generator is presented in Figure 18. This enables an efficient support of $M \times N$ connections — as shown in Section 4.3— with data redistribution on the user side, the provider side or even both.

The support for `UseProvide` connections with only one of the role filled by a parallel port is supported by two connection adaptors. The `Scatter` transformer whose definition is presented in Figure 19 supports a connection whose user role is filled by a `ParallelReceptacle` as if they was filled by a sequential `Receptacle`. It contains a component in charge of distributing the data connected to all the `part` sub-connections of the bundle and exposing an open connection with a sequential `Receptacle` used as result of the transformer. A `Gather` transformer supports the symmetrical case.

4.3   Analysis

Both connectors have been used to compile versions of the motivating application with a degree of parallelism varying from 1 to 1000 components. The compilation time varies between three and seven seconds. This is completely acceptable when compared with the 50 seconds required to compile the primitive components on the same machine or the deployment time of a distributed CCM application that is usually in tens of seconds.

```
adaptor Scatter<Integer N> supports
UseProvide<user={ParallelReceptacle<N, MatrixPart>}>
as UseProvide<user={Receptacle<Matrix>}> {
  Distributor<N> dist;
  each(i:[1..N]){
      merge(dist.in[i], supported.user.part[i]);}
  merge(this, dist.out);
}
```

FIGURE 19 – Definition of the `Scatter` transformer.

In the case of shared memory interactions, we have seen that HLCM makes it possible to expose a single open connection whether the component is sequential or a composite containing multiple instances participating in the connection as shown in Figure 15.

Similarly, for parallel method call interactions, a component exposing a `UseProvide<user={Receptacle<Matrix>}>` open connection can be sequential. Thanks to the mechanism of connection adaptors, the component can also have a parallel implementation exposing an open connection whose actual type is `UseProvide<user={ParallelReceptacle<N, MatrixPart>}>`. In this case, a component sequentializing the interaction is automatically inserted if required by the adaptor.

This means that the implementations of the components in HLCM/CCM presents the same characteristics as the second version of the motivating example presented in Figure 2 in terms of implementation exchangeability. This is an advantage over the first version that does not support the replacement of components.

In terms of performance however, the HLCM/CCM versions of these examples contain a single connection. Hence, it is possible to choose the best implementation of the connection with no bottleneck introduced similarly with the first version of the motivating example presented in Figure 1. In the parallel method calls for examples, if both components are parallel, the `MxN` generator is used rather than using the two connection adaptors that would sequentialize the interactions

This is an important advantage over the second version of the motivating example in term of performance as can be seen in Figure 20. This figure shows the time required to execute a $M \times N$ interaction with a degree of parallelism of three on the client side and four on the server side in function of the size of the parameters. This experiment has been conducted on a cluster of the Grid5000 french experimental platform. As can be seen, the HLCM version is very close to the PACO++ version —a specialized $M \times N$ environment— while the version that sequentializes communications is between two and six times slower depending on the size of the data.

5   Conclusion

Component models appear very interesting for complex numerical scientific applications targeted to be run on complex parallel and distributed infra-
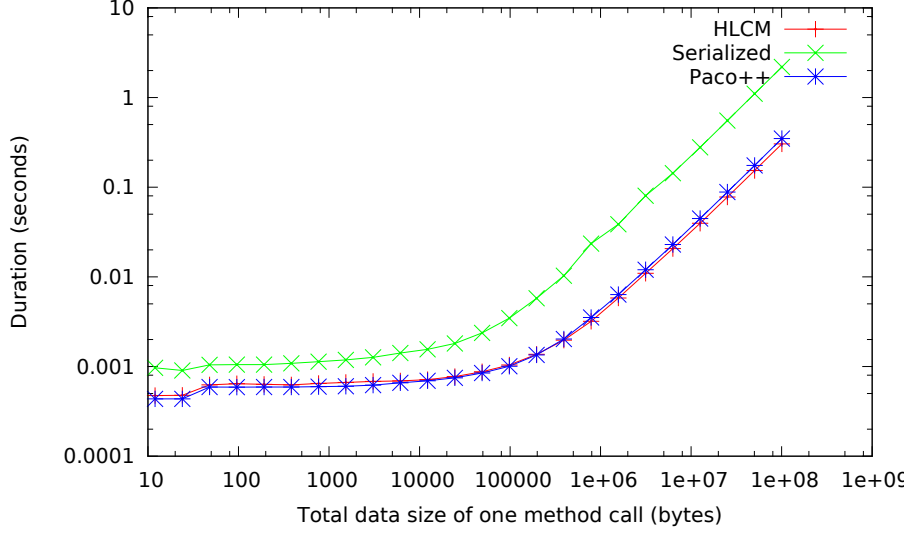
FIGURE 20 – Comparison of three implementations of the parallel method call interactions : the version described in this section (HLCM), the version that sequentializes interactions to support implementation exchangeability of Figure 2 (Serialized) and a version of a dedicated environment (PACO++).

structures. While advanced component models have been proposed to ease the description of applications, their implementation and the possibility to support many interaction kinds and to optimize them to a particular situation was still difficult tasks.

This paper has studied the feasibility and the benefit of using connectors in hierarchical component models. It first shows that it is feasible based on the definition of HLCM, that proposed the concept of open connections, and on a proof-of-concept implementation based on model transformation. Moreover, it shows that simple and efficient implementations of parallel interactions (shared data and parallel method calls) can be defined. Moreover, multiple implementations are supported without impacting the component interface while enabling optimized implementations.

There are two main future works. First, the transformation phase needs further research. The issue seems to find a right tradeoff between the expressiveness of HLCM and the complexity of computing a transformation (the problem is probably NP-hard with high expressiveness). Second, as workflows and dataflows are important interactions in scientific applications, HLCM has to support them. Though HLCM may support dynamicity, 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.

Références

[1] M. Aldinucci, H. L. Bouziane, M. Danelutto, and C. Pérez. STKM on SCA : a unified framework with components, workflows and algorthmic skeletons. In *15th Intl European Conference on Parallel and Distributed Computing (Euro-Par 2009)*, volume 5704 of *LNCS*, pages 678 – 690, Delft, Netherlands, August 2009. Springer.

[2] B. A. Allan et al. A component architecture for high-performance scientific computing. *International. Journal of High Performance Computing Applications*, 20(2) :163–202, 2006.

[3] G. Antoniu, L. Bougé, and M. Jan. Juxmem : An adaptive supportive platform for data sharing on the grid. *Scalable Computing : Practice and Experience*, 6 :45–55, Nov. 2005.

[4] G. Antoniu, H. Bouziane, M. Jan, C. Pérez, and T. Priol. Combining data sharing with the master-worker paradigm in the common component architecture. *Cluster Computing*, 10 :265 – 276, 2007.

[5] G. Antoniu, H. L. Bouziane, L. Breuil, M. Jan, and C. Pérez. Enabling transparent data sharing in component models. In *6th IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 430–433, Singapore, May 2006.

[6] D. Bálek and F. Plasil. Software connectors and their role in component deployment. In *Proceedings of the IFIP TC6 / WG6.1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems*, pages 69–84, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V.

[7] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Pérez. Gcm : A grid extension to fractal for autonomous distributed components. *Special Issue of Annals of Telecommunications : Software Components – The Fractal Initiative*, 64(1) :5–24, 2009.

[8] J. Bigot, H. L. Bouziane, C. Pérez, and T. Priol. On abstractions of software component models for scientific applications. In *Euro-Par 2008 Workshops - Parallel Processing*, pages 438–449, Berlin, Heidelberg, apr 2009. Springer-Verlag.

[9] J. Bigot and C. Pérez. Enabling collective communications between components. In *CompFrame '07 : Proceedings of the 2007 Symposium on Component and Framework Technology in High-Performance and Scientific Computing*, pages 121–130, New York, NY, USA, 2007. ACM Press.

[10] J. Bigot and C. Pérez. Increasing reuse in component models through genericity. In *Proceedings of the 11th International Conference on Software Reuse, ICSR '09*, LNCS, pages 21–30, Berlin, Heidelberg, oct 2009. Springer-Verlag.

[11] E. Bohm, A. Bhatele, L. V. Kale, M. E. Tuckerman, S. Kumar, J. A. Gunnels, and G. J. Martyna. Fine grained parallelization of the car-parrinello ab initio md method on blue gene/l. *IBM Journal of Research and Development*, 52(1/2), 2007.

[12] H. L. Bouziane, C. Pérez, and T. Priol. Extending software component models with the master-worker paradigm. *Parallel Comput.*, 36(2-3) :86–103, 2010.

[13] E. Bruneton, T. Coupaye, and J-B. Stefani. *The Fractal Component Model, version 2.0.3 draft*. The ObjectWeb Consortium, Feb. 2004.

[14] É. Canot, C. de Dieuleveult, and J. Erhel. A parallel software for a saltwater intrusion problem. In G. Joubert, W. Nagel, F. Peters, O. Plata, P. Tirado, and E. Zapata, editors, *Parallel Computing : Current and Future Issues of High-End Computing*, volume 33 of *NIC*, pages 399–406, 2006.

[15] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI : The Complete Reference – The MPI-2 Extensions*, volume 2. The MIT Press, 2 edition, September 1998. ISBN 0-262-57123-4.

[16] L. V. Kale and S. Krishnan. *Parallel Programming using C++*, chapter Charm++ : Parallel Programming with Message-Driven Objects, by Gregory V. Wilson and Paul Lu, pages 175–213. MIT Press, 1996.

[17] S. Matougui and A. Beugnard. Two ways of implementing software connections among distributed components. In *OTM Conferences (2)*, pages 997–1014, 2005.

[18] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd international conference on Software engineering*, pages 178–187, New York, NY, USA, 2000. ACM.

[19] Object Management Group. *Common Object Request Broker Architecture Specification, Version 3.1, Part 3 : CORBA Component Model*, Jan. 2008.

[20] Open Service Oriented Architecture. *SCA Service Component Architecture : Assembly Model Specification Version 1.00*, Mar. 2007.

[21] C. Pérez, T. Priol, and A. Ribes. A parallel corba component model for numerical code coupling. In M. Parashar, editor, *Proc. 3rd International Workshop on Grid Computing*, volume 17 of *Lect. Notes in Comp. Science*, pages 88–99, Baltimore, Maryland, Nov. 2002. Springer-Verlag. Special issue Best Applications Papers from the 3rd Intl. Workshop on Grid Computing.

[22] C. Pérez, T. Priol, and A. Ribes. A parallel corba component model for numerical code coupling. *The International Journal of High Performance Computing Applications*, 17 :417–429, 2003.

[23] C. Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.