Enabling Connectors in Hierarchical Component Models

Julien Bigot, Christian Pérez

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

Enabling Connectors in Hierarchical Component Models

Julien Bigot ,
Christian Pérez

Febuary 2010

Research Report N° RRLIP2010-9

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`




Enabling Connectors in Hierarchical Component Models

Julien Bigot Christian Pérez

Febuary 2010

Abstract

The continual growth of computing and storage capabilities enable 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 from Architecture Description Languages (ADL) are a promising solution to this problem. However, they have only been introduced in flat component models.

This paper describes HLCM, a model supporting both connectors and component hierarchy. This is achieved by describing potential interaction of components using the new concept of *open connections*. Complex interactions such as data sharing and parallel interactions are successfully supported by HLCM. An implementation, based on model transformation and on CCM, illustrates its feasibility and benefits.

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

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. Ils n'ont cependant été introduits que dans des modèles de composants à plat.

Ce papier décrit HLCM, un modèle de composants qui supporte à la fois les connecteurs et la hiérarchie. Ce résultat est obtenu en décrivant les interactions potentielles des composants à l'aide du concept de *connexion ouverte*. Les interactions complexes telles que le partage de données ou les interactions parallèles sont gérées avec succès par HLCM. Une mise en œuvre basée sur une transformation de modèle et sur CCM illustre sa faisabilité et ses bénéfices.

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

1 Introduction

Component based software engineering [14] is an interesting approach to simplify the development of complex applications such as scientific simulations as it improves code modularity and re-use as well as a better identification of code interactions and dependencies. In this paradigm, pieces of code are embedded into a component whose interactions with the environment are identified by a set of ports specifying both the services used and offered. Component-based applications are described by an assembly of component instances interconnected through their ports.

Scientific applications usually require a large amount of computing power and/or storage, typically delivered by complex hardware resource architectures, such as parallel and distributed infrastructures. They are therefore making use of algorithms thought to express parallel constructs. Component models only providing standard use/provide interactions are not satisfactory as they imply a strong binding between assemblies and hardware resources. An approach to solve this problem is to let these interactions be described at a higher level of abstraction, such as parallel-to-parallel component interactions, and to let their implementation be chosen when resources are known.

Supporting new kinds of interactions in component models that have not been designed for this purpose is a difficult task. An interesting concept enabling this support is brought by connectors from Architecture Description Languages (ADLs). They provide a generic mechanism to describe interactions between components. Though their introduction in software component models has been studied, it has been limited to *flat* component models.

This paper aims at studying the possibilities and the benefits in using connectors within hierarchical component models so as to efficiently support interactions between components. The difficult issue is to define a mechanism to let connectors cross composite definition. This paper proposes a generic hierarchical component model, named HLCM, which support connectors. It relies on the concept of *open connections* to specify interactions amongst components. Moreover, HLCM provides *bundle ports* and *connection transformers* to enable complex interactions. An implementation, restricted to static application, shows the feasibility of the model.

The remainder of this paper is organized as follow. Section 2 deals with the context while Section 3 presents HLCM. Examples using this model are described and discussed in Section 4. A proof-of-concept implementation is presented in Section 5. Section 6 draws some conclusions and presents some perspectives.

2 Context

This section presents an overview of related work on component models with support for hierarchy, HPC dedicated features and connectors. It discusses the advantages and limitations of each approach by focusing on two motivating examples: parallel code coupling through shared memory or (parallel) method calls. Finally, it studies the problems arising when combining hierarchy and connectors in a unique model.

Hierarchy in component models. Several component models support component hierarchy, such as FRACTAL [9] and SCA [12]. They support the concept of composite: a component whose implementation is an assembly of component instances interacting together. Ports exposed by composites are implemented by these internal instances. In SCA, this is achieved by the mean of *promotion*: ports of composites are defined as aliases of compatible internal instances ports. In FRACTAL, the concept of component *membrane* offering two views is used for this purpose. The first view describes the set of ports exposed by the composite while the second one is connected to its internal instances to provide its implementation.

Hierarchy is required to use component at multiple level of granularity. For example, it enables the description of a parallel component as an assembly of sequential components whose communications are handled by the model and whose placement can be handled by a dedicated mechanism. The efficient mapping of logical interactions onto physical ones is however highly dependent on

the placement of components on hardware resources. Therefore, such a mapping should not be embedded into an application assembly by application developper.

Dedicated HPC interactions. To deal with high performance computing, some interactions have thus been proposed as extensions to component models. $M \times N$ method calls from, to and between parallel (SPMD) components are for example available in CCA [2], GCM [5] (based on FRACTAL), and GRIDCCM [13] (an extension to CCM). Data sharing between components has also been proposed as an extension to CCA and CCM [3], and MPI-like collective communications as an extension to CCM [6]. Finally, some interactions are supported as part of more generic extensions such as the master/workers paradigm [8], and more generally (parallel) algorithmic skeletons [1].

The existence of these extensions demonstrates a clear need for HPC dedicated interactions, whose number is not known. Moreoever, their implementation is complex and typically results in incompatible component models. This is due to the fact that component models were not designed with the support for additional types of interactions. Models based on FRACTAL such as GCM can partially support this by intercepting connections in the membrane to modify their behavior. This is however limited to the local mapping of new interactions on existing ones preventing optimized implementations relying on a global knowledge of the participants to the connection.

Component models with connectors. The concept of connector originates from ADLs. Connectors are first class entities similarly to components used to describe their interactions [11]. They have already been introduced in component models, for example in the SOFA component model [4] or in [10]. In these models, connectors contain roles (or plugs) fulfilled by ports of component instances to form a connection. 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.

Connectors make it possible to efficiently support complex interactions such as $M \times N$ method calls as there is a global knowledge of the participants when generating their implementation. Connectors have however only been introduced in flat component models until now. As explained before, hierarchy is a strong requirement to support multiple levels of granularity. A model supporting both features would be valuable.

Analysis. In order to understand the implications of the interactions between hierarchy and connectors, let us study the implementation of two motivating examples in an hypothetical model with both features. As discussed earlier, the parallel components would be composites containing instances of sequential components. Interactions between those instances could easily be supported by connectors providing MPI-like interactions for example. A first example is the connection of two parallel components with shared memory. It might require access to the shared memory by all the internal instances. The composite can either expose the memory sharing ports of its internal instances as a set of independent ports or group them in an internal connection. While the first solution fails to express the fact that the ports are part of a single interaction, the second one prevents interactions with instances outside the composite.

Similarly for the second example, in the case of a connection by method call, the composite can either expose the ports of its internal instances or group them so as to expose a single port making a sequential call. The fisrt solution fails again to express that the ports are part of a single interaction and the second one implies a bottleneck in the case of parallel to parallel connection. Additionally, it should be possible for parallel components to be connected to sequential components. Relying on a distinct connector implementation for each case implies a quadratic number of implementations.

To summarize, a component model supporting both connectors and hierarchy should make it possible for connections to logically cross composite definitions. In addition, it should be possible to define a new type of ports without having to implement too many new connectors.

```
connector UP {
  role user;
  role provider;
}
```

Figure 1: Example of declaration of a connector `UP` that supports Use/Provide interactions.

```
component MyComponent exposes {
  UP { user PT; } aC;
} ...
```

Figure 2: Example of a component exposing a connection named `aC` whose role `user` is filled by a port of type `PT` in the component implementation (not shown). The role `provider` is not filled.

```
component MyCcmComponent {
  provides A a;
  uses B b;
}
```

```
component MyHlcmPrimitive exposes {
  UP { provider CcmFacet<A>; } a;
  UP { user CcmReceptacle<B>;} b;
} ccm (''MyCcmComponent'')
```

Figure 3: Example of a CCM component described in CORBA IDL3 (left) and its corresponding HLCM component (right). `CcmFacet` and `CcmReceptacle` are two generic port types natively supported in HLCM/CCM.

3 HLCM: a High Level Component Model

This section introduces HLCM, a generic component model with support for hierarchy and connectors. HLCM relies on an underlying execution model for the definition of some of its concepts (*i.e.* primitive components and connectors). For example HLCM/CCM uses CCM as its underlying execution model; its implementation is described in Section 5. First, the structural elements of HLCM are described and then its behavior is illustrated with an algorithm mapping an HLCM application to a primitive one.

3.1 Structural Elements of HLCM

The basis of HLCM is a standard hierarchical component model. Components expose a set of named interaction points and have an implementation. This implementation can be either primitive or composite. Primitive implementations are provided by the underlying execution model. Composite implementations are provided by an assembly of component instances and connections. HLCM supports genericity [7]; examples make use of a notation similar to JAVA generics.

As in other component models supporting connectors, interactions between components are described by connections that are instances of connectors. Connectors are first class entities that define a type of interaction. They contain a set of roles. Roles are named and have a multiplicity: either single (default) or unbounded. An examples of connector is shown in Figure 1. Roles in connections are filled by ports, roles of unbounded multiplicity can be filled by multiple ports.

A specificity of HLCM is that the interaction points of components are not ports but connections. These connections have some of their roles internally fulfilled by the implementation of the component but not necessarily all. Some roles will be fulfilled externally when connecting the component as will be explained hereafter. Connections allowing external role fulfillment are called *open connections*. An example of component exposing an open connection is shown in Figure 2. Ports can be either primitive ports or bundle ports. Bundle ports contain a set of named open connections.

Primitive Components. The definition of primitive component implementations depends on the targeted model. In HLCM/CCM, primitive components are implemented by CCM components. CCM components expose ports whereas their HLCM/CCM counterparts expose connections. Ports of CCM components are thus wrapped in HLCM connections of the same name as shown in Figure 3.

Composite Components. A composite component is described by an assembly of component instances and connections. A composite exposes a connection by making an alias to one of its

```
component MyComposite exposes {
  UP { provider CcmFacet<A>; } a;
} composite {
// Two internal component instances
   MyHlcmPrimitive c1;
   MyHlcmPrimitive c2;
   ...
   ...
// Exposition of c1.a as a
A: this.a = c1.a;
// Interaction between c1.b and c2.a
X: UP cnab;
Y: cnab |= c1.b; cnab |= c2.a;
}
```

Figure 4: Example of a composite implementation containing two internal component instances `c1` and `c2`. It exposes the connection `c1.a` as `a` using the alias operator `=`. It lets `c1` and `c2` interact through a connection `cnab` using the merge operator `|=`.

```
generator  UPLog<interface UI, interface PI> with {
   UI super PI;           // constraints
} implements UP {
   provider CcmFacet<PI>;
   user CcmReceptacle<UI>;
} {
  LoggerProxy<UI> proxy
  UP up1; up1.user += this.user; up1 |= proxy.clientSide;
  UP up2; up2.provider += this.provider; up2 |= proxy.serverSide;
}
```

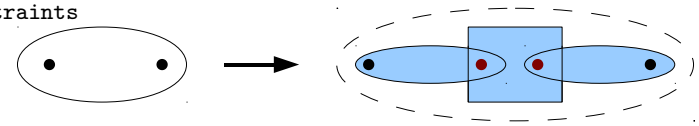

Figure 5: Example of composite generator inserting a proxy component, which constraints the user type to be a parent of the provider interface type. The `+=` operator fulfills a role with a port. For example, the role `user` of the connection `up1` is fulfilled with the port `user` of the considered connection.

internal connections (line A in Figure 4). An interactions between component instances is set by creating a connection (line X in Figure 4) and by binding it to the open connections by which these instance are to be connected (Lines Y in Figure 4). This creates a logical connection whose roles fulfillment are the union of those of the bound open connections.

Generators. Generators are implementations of connectors. A connector can be implemented by several generators. A generator implements a specialization of a connector, that is to say a connector with constraints on the number, the type and the locality of the ports fulfilling its roles.

There are two kinds of generators: primitive and composite. Primitive generators specify the interactions directly supported by the underlying execution model. Composite generators generate an assembly in which the ports fulfilling the roles of the connector are made part of the connections as illustrated in Figure 5. This assembly can be parametrized by the number, type and locality of the ports fulfilling the roles of the connector.

Connection Transformers Connection transformers provide a functionality similar to inheritance in object oriented models. They make it possible to use a connection of a given type where another type was expected. The definition of a connection transformer is an assembly that uses the available connection and exposes a connection of the expected type instead as illustrated in Figure 6.

3.2 Behavior of HLCM Elements

The behavior of an HLCM application is defined through an equivalence with a primitive application, *i.e.* an application described in the underlying execution model. This means that it is fully defined by the combination of the definition of the behavior of applications in the underlying execution model and a mapping algorithm. Let us now further discuss this mapping algorithm.

An HLCM application is defined by the set of HLCM elements it contains: components, connectors, generators, port types and connection transformers and by the component used as the

Algorithm 1 Transforming an abstract HLCM application into a concrete one.

Input:
- An HLCM application

Output:
- A primitive application or an error

while composite component instances or unimplemented connections remain **do**
  Replace the composite component instances by the content of their assembly and merge their exposed connections to those they are bound to;
  Choose a set of connection transformers and generators whose constraints can be fulfilled to implement the connections or rollback or return an error;
  Replace the composite connections by the content of their assembly;
end while

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 primtive connections.

Such a transformation can be achieved by applying Algorithm 1 that replaces composite instances by the content of their assembly and chooses the generators to use for the implementation of connections. It is non deterministic as it does not specify how the choice of connection implementations is made. If no valid choice can be made at a given point, either a rollback is done or an error is returned. Any assembly obtained by applying this algorithm is defined as providing a valid behavior of the application.

The difficult part when implementing this algorithm lies in the choice of connection implementations. The identification of the valid combinations of connection transformers and generators that might be used to implement a given connection is a complex problem. As the amount of generators and connection transformers applicable to a given connector is expected to remain rather small, a naive implementation trying all combinations seems however acceptable.

It must be noted however that the choice of the implementation is not a self contained problem. Locality constraints introduce dependencies between these choices. For example in a situation where two component instances are connected by two distinct connections, their locality constraints must be compatible. In the general case, this is expected to be NP-hard as most planning problems.

4 Evaluation of HLCM to Support HPC Interactions

This section evaluates the use of HLCM/CCM to implement the two motivating examples introduced in Section 2: interactions through shared memory and (parallel) method calls between parallel components.

```
transformer PushPull
supports UP { user CcmReceptacle<DataPush>; } input
as       UP { provider CcmFacet<DataPull>;  } output
{
  CacheComponent c;
  UP cnx;
  cnx |= input;
  cnx |= c.pushSide;
  output = c.pullSide;
}
```

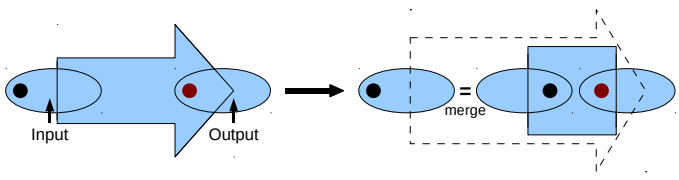

Figure 6: Example of connection transformer describing how a `UP` connection (`input`) whose `user` role is filled with a `CcmReceptacle<DataPush>` port can be seen as a `UP` connection (`output`) whose `provider` role is filled with a `CcmFacet<DataPull>` port. It does so by inserting a component instance acting as a cache.

```
connector SharedMem {
  role access[];
}
```

Figure 7: The `SharedMem` connector declaration with an unbounded role `access`.

```
interface CDataAccess {
  CPointer get_data();
  long get_size();
  ...
}
```

Figure 8: IDL declaration of the CDataAccess interface.

```
generator LocalSharedMem<Integer N> implements SharedMem {
  for (Integer i in [1..N]) { access[i] CcmFacet<CDataAccess>; }
} with { // locality constraints
  for (Integer i in [1..N-1]){access[i].process == access[i+1].process;}
} composite {
  LocalMemoryStore store;
  for (Integer i in [1..N]) {
    UP cnx[i]; cnx[i].user += access[i]; cnx[i] |= store.access;
} }
```

Figure 9: 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.

Shared memory interaction. In order to support memory sharing inspired by [3], let us define a `SharedMem` connector whose declaration is given in Figure 7. It contains a single role `access` of unbounded multiplicity.

From the point of view of primitive components, the access to a `SharedMem` connection is done through a `CDataAccess` interface whose IDL description is given in Figure 8. `CPointer` is a valuetype holding a native reference to the actual data. It can then be only used between instances located in a same process.

Figure 9 presents a generator for `SharedMem` connections based on a local centralized implementation. The `N` ports fulfilling its `access` role are of type `CcmFacet<CDataAccess>`.

Figure 10 describes another generator for `SharedMem` connections based on a distributed implementation. For each accessor, it instantiates a local `DsmNodeComponent` component which is interconnected with all other (distributed) `DsmNodeComponent` instances. Each `DsmNodeComponent` is constrained to be colocated to the same process as its associated accessor, because of the use of

```
generator DistributedSharedMem<Integer N> implements SharedMem {
  for (Integer i in [1..N]) { access[i] CcmFacet<CDataAccess>; }
} composite {
  for (Integer i in [1..N]) {
    DsmNodeComponent node[i];
    LocalUP cnx[i]; cnx[i].user += access[i]; cnx[i] |= node[i].access;
  }
  for (Integer i in [1..N]) { for (Integer j in [1..N]) {
    UP in[i,j]; in[i,j] |= node[i].from; in[i,j] |= node[j].to;
  } }
}
```

Figure 10: Definition of the `DistributedSharedMem` generator supporting `SharedMem` amongst distributed component instances. It is made of a set of `DsmNodeComponent` instances, one for each accessor. Each instances is connected to all of them through two dedicated `UP` connections, one in each direction.

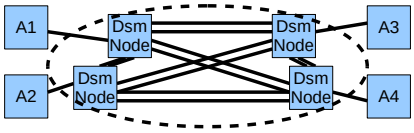

Figure 11: A `SharedMem` connection with four accessors (`A1` to `A4`) implemented by the `DistributedSharedMem` generator.

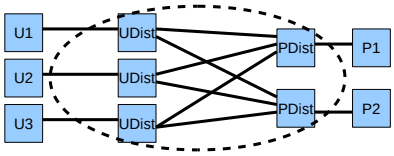

Figure 12: A parallel `UP` connection implemented by the `MxN` generator. A proxy instance is inserted for each participant. Each proxy instance is connected to all those of the opposite side.

```
bundle ParallelCcmFacet<Integer N, interface I> {
  for (Integer i in [1..N]) { UP { provider CcmFacet<I>; } part[i]; }
}
```

Figure 13: Definition of the `ParallelCcmFacet` bundle port type. It contains `N UP` connections called `part` whose `provider` role if fulfilled by a `CcmFacet<I>` port.

the `LocalUP` connector. An example of a connection implemented by this generator is presented in Figure 11.

Parallel method call interaction. Unlike shared memory, the support for parallel method calls [13] does not require the introduction of a new connector, the `UP` connector already supports method calls. It only requires the support of new type of ports fulfilling its roles: the `ParallelCcmFacet` whose definition is presented in Figure 13 and the symmetrical `ParallelCcmReceptacle`.

A `MxN` generator needs to implement `UP` connections whose roles are fulfilled by these two ports. It is quite similar to the `DistributedSharedMem` generator. An example of connection implemented by the `MxN` generator is presented in Figure 12. This enables an efficient support of $M \times N$ connections with data redistribution on the user side, the provider side or even both.

The support for `UP` connections with only one of the role filled by a parallel port is implemented thanks to two transformers. The `Scatter` transformer whose definition is presented in Figure 14 supports a connection whose user role is filled by a `ParallelCcmReceptacle` as if they was filled by a sequential `CcmReceptacle`. 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 `CcmReceptacle` used as result of the transformer. A `Gather` transformer supports the symmetrical case.

Discussion. As can be seen with these two examples, HLCM easily and efficiently supports the implementation of both shared memory and parallel method calls as connectors without having to modify either the model or its implementation. Efficiency is obtained because generated concrete

```
transformer Scatter<Integer N>
supports UP { user ParallelCcmReceptacle<N, MatrixPart> } input
as       UP { user CcmReceptacle<Matrix>;               } output
{
  Distributor<N> dist;
  for ( Integer i in [ 1 .. N ] )
  { UP cnx[i]; cnx[i] |= input.user.part[i]; cnx[i] |= dist.in[i]; }
  output = dist.out;
}
```

Figure 14: Definition of the `Scatter` transformer.

applications are the same as with the dedicated shared data and parallel extensions. In addition, the concept of open connections makes it possible for connections to logically cross the definition of composites, thus enabling support for hierarchy.

Another interesting point are transformers that make it possible for parallel ports to be used as their sequential counterpart. This enables the support of future implementations such as a load balanced facet designed to support the master/worker paradigm for example.

A current limitation of HLCM/CCM is the lack of genericity in CCM itself. As a result, primitive components and generators using them are limited to specific data types. Using a generic model as backend would solve this.

It would be also interesting for components to be allowed to have multiple implementations, similarly to generators for connectors. This would however make the transformation algorithm more complex as the choices of implementations for components and connectors would be completely dependent of each other.

5 A Proof-of-Concept Implementation of HLCM

We have developed a proof-of-concept implementation of HLCM/CCM based on a Model Driven Engineering (MDE) approach. It transforms an HLCM/CCM application into a plain CCM assembly in three steps. First the HLCM/CCM files are parsed to create a model instance, then this model instance is transformed according to Algorithm 1, and finally the result of this transformation is dumped into a CCM CAD file. This implementation relies on the tools provided as part of the Eclipse Modeling Framework (EMF).

A meta-model of HLCM/CCM has been written in the ECORE language. It contains about 100 meta-classes amongst which about 10 are specific to CCM. A parser creating instances of this model from HLCM/CCM files has been implemented based on the Xtext framework.

The implementation of Algorithm 1 required around 1200 lines in JAVA. It works on instances of a second model describing instantiated HLCM/CCM assemblies adding 15 ECORE classes to the first model. After the transformation, the assembly contains only primitive component instances and connections and can be dumped to its CCM CAD counterpart in 100 lines of JAVA.

This proof of concept implementation has been successfully used to transform the two examples described in Section 4. A typical transformation takes less than five seconds on a standard laptop amongst which more than three seconds are spent in initialization and parsing.

The choice of connection implementations is still a random choice amongst the set of generator requiring the minimal number of connection transformations. Smarter choices would require performance information on components and connectors as well as heuristics to take them into account.

Locality constraints expressible in generators are currently limited to process collocation. This simplifies the problem of placement since these constraints can not lead to any contradictions. Moreover, they can be expressed in the output CCM CAD file. Fully supporting locality constraints would require a resource model as well as the use of a constraint solver in the transformation algorithm.

Another limitation of this implementation is that it is restricted to static applications. The choice of a compilation prevents the support of dynamic modifications of the assembly. Such support would require a deeper integration between the transformation algorithm and the target component model.

6 Conclusion

Component models appear very interesting for complex numerical scientific applications targeted to be run on complex parallel and distributed infrastructures. While advanced component models are proposed to ease the description of applications, the implementation of such models as well as the possibility to optimize an application to a particular infrastructure are 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 as well as 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.

There are two main perspectives. First, though HLCM supports dynamicity, an efficient implementation supporting it remains to be done. Second, the optimization of an HLCM application with respect to available resources can be improved, with the support of both multiple component implementations and performance information on primitive components and connections.

References

[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, 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] H. L. Bouziane, C. Pérez, and T. Priol. Extending software component models with the master-worker paradigm. *Parallel Computing*, In Press, 2010. DOI: 10.1016/j.parco.2009.12.012.

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

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

[11] 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.

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

[13] 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.

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