HAL

archives-ouvertes.fr

A Generic Model Driven Methodology for Extending Component Models

Julien Bigot, Christian Pérez

▶ **To cite this version:**

HAL Id: inria-00401656

https://hal.inria.fr/inria-00401656

Submitted on 3 Jul 2009

INRIA

A Generic Model Driven Methodology for Extending Component Models

Julien Bigot — Christian Pérez

N° 6979

Thème NUM

Rapport de recherche

A Generic Model Driven Methodology
for Extending Component Models

Julien Bigot[*], Christian Pérez[†]

Thème NUM — Systèmes numériques
Équipe-Projet GRAAL

Abstract: Software components have interesting properties for the development of scientific applications such as easing code reuse and code coupling. In classical component models, component assemblies are however still tightly coupled with the execution resources they are targeted to. Dedicated concepts to abstract assemblies from resources and to enable high performance component implementations have thus been proposed. These concepts have not achieved widespread use, mainly because of the lack of suitable approach to extend component models. Existing approaches – based on ad-hoc modifications of component run-times or compilation chains – are complex, difficult to port from one implementation to another and prevent mixing of distinct extensions in a single model.

An interesting trend to separate application logic from the underlying execution resources exists; it is based on meta-modeling and on the manipulation of the resulting models. This report studies how a model driven approach could be applied to implement abstract concepts in component models. The proposed approach is based on a two step transformation from an abstract model to a concrete one. In the first step, all abstract concepts of the source model are rewritten using the limited set of abstract concepts of an intermediate model. In the second step, resources are taken into account to transform these intermediate concepts into concrete ones. A prototype implementation is described to evaluate the feasibility of this approach.

Key-words: software components, scientific computing, model-driven engineering, software connectors, algorithmic skeletons, genericity

* LIP/INSA
† LIP/INRIA

Une méthode générique basée sur les modèles pour étendre les modèles de composant

Résumé : Les composants logiciels ont des propriétés intéressantes pour le développement d'applications scientifiques, par exemple l'augmentation de la réutilisabilité du code ou la simplification du couplage de code. Dans les modèles de composants classiques, les assemblages de composants restent cependant fortement liés aux ressources d'exécution pour lesquelles ils ont été prévus. Des concepts destinés à abstraire ces assemblages des ressources ont donc été proposés. Ces concepts ne connaissent pas une utilisation généralisée, principalement à cause du manque d'une approche adaptée pour l'extension des modèles de composants. Les approches existantes – basées sur des modifications ad hoc des supports d'exécution ou des chaînes de compilation – sont complexes, difficiles à porter d'une mise en œuvre à l'autre et empêchent l'utilisation simultanée de plusieurs concepts distincts dans un seul modèle.

Une tendance intéressante pour séparer la logique applicative des ressources d'exécution sous-jacentes existe. Elle est basée sur la méta-modélisation et sur la manipulation des modèles résultants. Ce rapport étudie comment une approche basée sur les modèles peut être appliquée pour mettre en œuvre des concepts abstraits dans les modèles de composants. L'approche proposée s'appuie sur une transformation en deux étapes depuis un modèle abstrait vers un modèle concret. Dans la première étape, tous les concepts abstraits du modèle source sont ré-écris en s'appuyant sur l'ensemble limité de concepts abstraits d'un modèle intermédiaire. Dans la seconde étape, les ressources sont prises en compte pour transformer ces concepts en leurs équivalents concrets. Un prototype de mise en œuvre est présenté pour évaluer la faisabilité de cette approche.

Mots-clés : composants logiciels, calcul scientifique, ingénierie dirigée par les modèles, connecteurs logiciels, squelettes algorithmiques, généricité

1 Introduction

Component based software engineering (CBSE) is an approach where applications are developed as a set of independent components interacting with their environment through well defined interfaces [26, 39]. This makes it easier to reuse code and to separate concerns between distinct developer teams. These are very interesting properties for the development of scientific applications such as code coupling simulations. As a matter of fact, these applications are typically build by putting together codes developed by distinct teams of scientists, at different time, each simulating one aspect of the object.

In order to make possible the incremental development of applications, the concept of composite component whose implementation is provided by a component assembly is often added to these models. In classical component models however, the organization of component instances as well as their interactions in assemblies is tightly bound to the execution resources they are targeted to. As a result, it is difficult to reuse composite components on different hardware resources.

In non component based applications, this problem is handled by dedicated frameworks such as MPI [37, 19] or skeletons [13] for example that provide support for well known parallel structures and interaction patterns. With these frameworks, the number of process to use is usually a parameter of the application and communications amongst processes are usually handled by the framework. As a result, applications are described in a much more abstract way and thus less tightly bound to the execution resources.

Efforts to make similar features available to component based applications have been done. Due to limitations in component models, these features have not been developed as component based frameworks, but rather as extensions of the models themselves. This means that either the compilation chain, the runtime, or even both had to be modified; this is a complex task that requires knowledge both in the domain of the implemented feature and of the internal behavior of the component model. As a result, most features have no easily usable implementation and mixing components relying on features implemented separately is nearly impossible.

This paper describes and evaluates the feasibility of a new approach to support features dedicated to parallel applications in component models. This approach is based on an intermediate model with genericity and connectors in which these features can be implemented as component based frameworks. The support for this intermediate model is based on Model-Driven Engineering (MDE). The components described in this model are transformed into components of an existing model by taking into account the execution resources.

The remaining of this paper is organized as follow. Section 2 describes the context that led to this work and Section 3 focuses on related works. In Section 4, the contribution of this paper, an approach based on MDE to implement features dedicated to parallel computing in component models is described. Then, Section 5 illustrates this approach on an example to evaluate its feasibility. Finally, Section 6 concludes the paper.

2   Context

This section describes the context that led to the work described in this paper. It provides a quick overview of the concepts found in industrial component models and describes features dedicated to parallel computing found either in dedicated model or as extension of industrial models. It also provides an overview of MDE used in Section 4 to implement the intermediate model.

2.1   Component models

Industrial component models include for example the CORBA Component Model (CCM) [32] defined by the Object Management Group (OMG), the Component Object Model (COM) and .NET assemblies[1] by Microsoft, the Enterprise Java Beans [17] (EJB) defined by SUN and the Service Component Architecture [34] (SCA) defined by the Open Service Oriented Architecture (OSOA) community. The three main concepts of these models are components, ports and assemblies.

Components are defined by their external interface and their internal implementation. The external interface defines the various possible points of interaction of the component. It is usually made of a list of named ports. It can also contain configuration properties used to configure the behavior of the component. This behavior is provided by the internal implementation. This implementation can be described in another model such as an object oriented or procedural language in which case it is called a primitive implementation. Some models also support the concept of composite components whose implementation is provided by an assembly.

Ports are the points of interaction between components. Their type specifies the validity and behaviour of connections as well as the interface of use from implementations. Industrial models usually define a fixed number of port types, but these types accept parameters. For example in CCM, the `facet` (`provide`) and `receptacle` (`use`) port types are parametrized with a CORBA object interface. In object-oriented primitive implementations, ports are usually mapped on object interfaces that must be implemented or that can be used, depending on the port type. The validity of connections between ports typically depends on their types as well as their multiplicity. This multiplicity is defined by the minimum and maximum (possibly infinite) number of connections a given port can be part of.

Assemblies describe a set of named component instances as well as the connections between their ports. In some models such as CCM or SCA, assemblies can be described using dedicated languages while in others, they must be dynamically created through calls to an API. Assemblies can include additional informations, in CCM deployment constraints are used to specify that some component instances must be deployed either in the same or in distinct processes.

2.2   High performance dedicated features

While industrial component models usually support distributed computing, they do not target parallel and High Performance Computing (HPC). This lack has

---

[1]http://www.microsoft.com/com/

led to the development of dedicated features discussed in this section. A common property of these features is that they let the user describe assemblies at an abstract level. Some parameters are set only when the execution resources are known. These features have been developed either as extensions to existing models or in new component models such as the Common Component Architecture [3] (CCA) by the CCA forum or the Grid Component Model [8] (GCM) by the CoreGRID European network of excellence.

A first example of feature dedicated to HPC is the concept of parallel component used to support the single program multiple data (SPMD) paradigm. The implementation of a parallel component is instantiated multiple times in distinct processes. The number of instances is chosen when execution resources are known. Two additional features are related to parallel components: $M \times N$ method calls and collective communications.

$M \times N$ method calls are used in order to efficiently couple parallel components. In this case, each process of the caller component transmits a part of the data to processes of the callee component. The redistribution algorithm is chosen when the mapping of processes on execution resources and the distribution of data are known.

Collective communications are used to make possible efficient exchange of data between processes whose number is not known at development time. This is achieved by relying on some well known communication patterns such as the broadcast, reduce or all-to-all data exchange. The exact algorithm to use for each pattern is chosen when the number of processes and their mapping on execution resources are known.

Parallel components are a base feature of CCA while $M \times N$ method calls are provided as an extension in the SCIRun2 [40] implementation. GCM include the concepts of gathercast and scatercast port types that support $1 \times N$ and $M \times 1$ communications and when connected together, $M \times M$ communications. Parallel components have been introduced together with $M \times N$ method calls in GridCCM [35], an extension to the CCM model. Collective communications have also been introduced as an extension to CCM [9].

Another interesting feature is the concept of task farm designed to support the master/worker paradigm used in parameter sweep applications. A task farm is made of two parts: a collection of instances of the worker code and a request dispatching code. When the master code emits a request, the dispatching code is responsible for choosing one of the worker instance and routing the request to this instance, then the worker is responsible for the execution of the request. The size of the collection is chosen when the execution resources are known and the dispatching algorithm when the mapping of processes on these resources is known.

This feature has been introduced as an extension that has been applied to both CCM and CCA [11, 7]. It has also been introduced as one of the skeletons supported by STKM [1].

Another feature used in many HPC applications is the concept of Distributed Shared Memory (DSM) used in applications where various processes work on a common piece of data with irregular access patterns. A DSM is usually made of two parts: a data exchange code and a locking code. The data exchange code is responsible for offering a coherent view of the memory to all processes accessing the DSM. The locking code is responsible for offering them a locking mechanism

used to handle concurrent accesses to the memory. The algorithms used for data exchange and for locking are chosen when the mapping of processes on execution resources is known.

This feature has been introduced as an extension that has been applied to both CCM and CCA [6].

To summarize, these features can be classified in two categories. A first category describes new kinds of component implementations. It includes parallel components and task farms. A second category describes new kinds of interaction between components. It includes $M \times N$ method calls, collective communications and data sharing.

2.3   Model Driven Engineering

This section gives an overview of the Model Driven Engineering [36] (MDE) approach and sketches some MDE initiatives with the associated models and tools.

MDE is an approach that intends to increase the level of abstraction for application development by focusing on the modeling of domain specific concepts rather than on their computational aspects. This is achieved thanks to the description of applications (M1) using domain specific models (M2). These domain specific models are themselves described with a dedicated model (M3). Usually, this pattern is limited and this model is used to describe itself as shown in Figure 1.
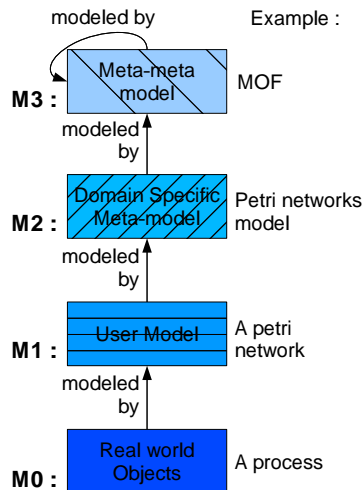

Figure 1: The four classical layers of the meta-model architecture

In order to make possible for applications described in domain specific models to be executed, an approach used in MDE is to develop transformation operations that operate at the meta-model level. These operations take as input the user-described model that conforms to the domain-specific meta-model and generate as output another model that conforms to a more general meta-model as shown in Figure 2. These transformation are themselves conforming to a

dedicated meta-model and different transformations can be applied for different target execution environment.
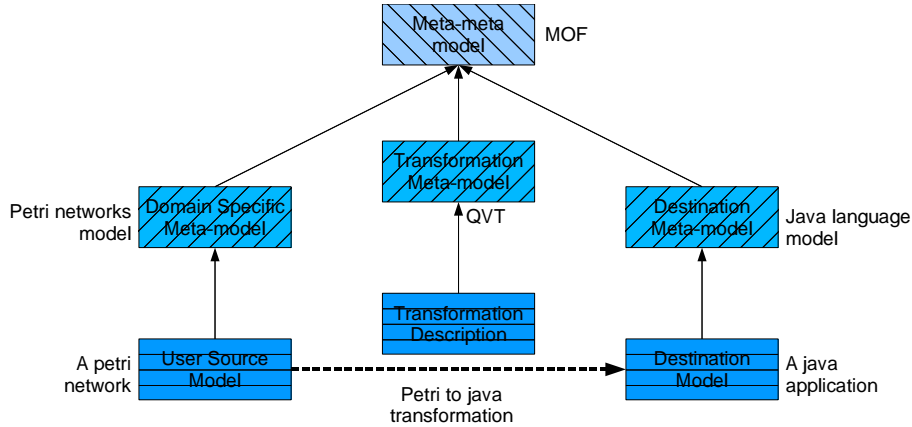

Figure 2: Architecture of model transformation operations

The best known MDE initiative is the Model Driven Architecture [29] (MDA) proposed by the OMG. In MDA, the source model is called *Platform Independent Model* (PIM) and the destination model *Platform Specific Model* (PSM). The transformation is done using the description of the platform provided in the *Platform Description Model* (PDM).

MDA specifies a set of models amongst whom the most important is the Meta Object Facility [30] (MOF) that is the common model of the fourth layer used for meta-modeling. MOF is quite similar to UML in its design; it contains the concepts of meta-classes with meta-attributes, meta-associations, etc. The MOF specification describes two flavors: Essential MOF (EMOF) with only core concepts and Complete MOF (CMOF) that is the full version.

Around MOF are several other models such as MOF Object Constraint Language (MOF OCL [31]) used to further specify the validity of instances of the described model. Query / View / Transformation [33] (QVT) is the model used to describe model to model transformations. The QVT specification describes two flavors: QVT Operational (QVTO) with an imperative approach and QVT Relations (QVTR) with a declarative approach.

Another MDE initiative is the Eclipse Modeling Framework [14] (EMF) from the eclipse foundation. EMF includes a number of tools based on EMF core (ECORE) that is very similar to EMOF. EMF adopts a more pragmatic approach than the MDA initiative with a weaker focus on specification and a stronger one on implementations, often adapting MDA standards.

Another MDE initiative with a somewhat different approach is kermeta [16]. Kermeta is based on the concept of executable models: methods of the meta-classes are used to validate the assemblies, implement transformations between models and implement the behavior of model instances.

To summarize, MDE provides an interesting approach to transform applications described by the user that conform to a domain specific model into applications conforming to another model.

3 Related works

This section describes various approaches that can be used to extend models with HPC dedicated features. It starts by describing how HPC features have been implemented in component models. Then it studies the concepts of algorithmic skeletons and Architecture Description Languages (ADLs) that provide an higher level of abstraction than what is found in component models.

3.1 HPC features implementation in component models

High performance dedicated features are either found in specific models such as CCA or GCM or provided as extensions to existing models. Implementing a whole new model to support these features has the advantage of providing a consistent model where all features are well integrated. This approach does however require all components to be rewritten for this new model and thus decrease the possibility of code reuse. Another problem is that whenever a new feature is required in an application, the problem of how to add it to the model arises again. The extension of existing models with additional features has thus been attempted mainly using two distinct approaches.

A first approach is to modify the model runtime code to support new features. This is for example what has been done by SCIRun2 to support $M \times N$ communications in CCA. While this approach seems to be the more intuitive solution, it has various drawbacks. The runtime is a critical part of the component model when it comes to performance, especially in the case of high performance applications; it is thus often finely tuned and not intended for easy modification. Supporting a new feature in the runtime can thus prove to be quite difficult and requires expertise in both the domain of the added feature and of the component model runtime implementation. Adding features to the runtime implementations also make component using these features dependant on this specific runtime. In the case where two features have been implemented in two distinct runtimes, the components can not be used together anymore.

Another approach has thus been to modify the compilation chain in order to map the new features on those included in the model or possibly by others provided by additional libraries. This approach has for example been taken for the implementation of GridCCM or collective communications. It has the advantage of making possible the coexistence of components using features implemented independently in the same assembly. It does however not solve the case where a single component uses more than one feature and must thus be compiled with two distinct compilation chains. Modifying the compilation chain is also a process that requires a good understanding of the low-level aspects of the model in addition to the expertise in the domain of the feature added.

While no satisfying approach to extend component models with additional features has been found, the idea of mapping additional features on a fixed set of concepts supported by the runtime model seems the most promising approach. Doing so by modifying the compilation chain seems to be difficult; however as seen in the previous section, the model-driven approach provides tools specifically dedicated to support these kinds of transformations.

3.2 Algorithmic skeletons

Skeletons are constructs that describe the structure of typical parallel composition patterns [13]. Skeletons have for example been written to describe various kinds of patterns such as pipeline or task farms. Skeleton implementations are intended to be written by system programmers that describe efficient implementations of the modeled patterns for the target platform. On the other hand, application programmers are intended to identify parallel patterns to use, and implement application by instantiating skeletons, possibly by nesting them, and eventually providing application specific code.

In classical skeleton frameworks, the user can describe basic sequential skeletons in an existing imperative or object-oriented language such as C, C++, Fortran, etc. These sequential skeletons can then be used as the payload of provided parallel skeletons. The composition of skeletons can usually be done either using a dedicated language or with calls to an API provided by the framework.

Due to their many common properties, skeleton and component technologies have been combined to make possible the description of applications taking advantage of both worlds [2, 15, 1]. This is typically implemented by adding new keywords to the component assembly language and by replacing instances of skeletons by component that dynamically instantiates its content. This means that the assembly is not entirely known at deployment time.

Skeletons provide an interesting way to express new kind of component implementations for parallel applications. Skeleton models do however typically provide only a limited and fixed set of constructs and the addition of new skeletons poses the same kind of problems as the addition of new features in component models. The instantiation of the content of skeletons is also typically the responsibility of the skeleton implementation which makes it difficult to make the right planning choices at deployment.

In a previous work [10], we described an approach to introduce genericity in component models: we have shown that it is possible to implement skeletons by relying on generic components (similar to C++ templates or Java generics) only. In this case, skeletons are mere composites with the type of some components in the assembly being a parameter.

With this approach, new skeletons can be implemented as simple new components and deployment tools can make planning choices in function of their content. Skeletons do however focus on component implementations only, they do not offer any solution to introduce new type of interactions between components.

3.3 Architecture Description Languages

ADLs [28, 12] are languages such as ACME [18], C2 [27], Darwin [23], Rapide [21, 22] or Wright [4] intended to make possible the description of application software architecture. The concepts found in ADLs are very similar to those of components assembly languages: applications are described as a set of component instances interacting together through well defined interfaces. Unlike component models however, ADLs do not focus on reuse of existing components but rather on the top down description of architectures mainly for the purpose of communication between parties involved in the development.

The architecture is usually described at a high abstraction level and then refined to eventually obtain the set of components that must be implemented. As a result of this different focus, ADLs often include tools to formally describe the behavior of components and connections so that assemblies validity and refinements compatibility can be checked. For example Darwin can use the finite state processes algebra [24] (FSP) and Wright the communicating sequential processes algebra [20, 4] (CSP).

In order to describe application architectures at a high abstraction level many ADLs (including ACME, C2, and Wright) include the concept of connector as a first class entity. A connector is a type whose instances are connections between component ports. Letting the user describe new connectors makes it possible to describe new types of interaction between components or to provide different behaviors for a connection depending on the connector used.

Some work has been made to bring the advantages of connectors to software component models [38, 25]. These models are however limited to supporting new kind of interaction between primitive components; they have no support for HPC dedicated features such as new kind of component implementations.

3.4   Conclusion

To summarize, existing extensions to component models targeting HPC have been implemented at a very low abstraction level, modifying either their runtime or compilation chain. On the other hand, skeletons and ADLs provide interesting higher level concepts, but these have not been combined in component models yet. The next section describes an approach based on MDE to introduce these concepts in component models in order to ease the introduction of HPC dedicated features.

4   A model transformation approach for the extension of component models

This section describes an approach to implement features dedicated to HPC in component models. The remaining of the section characterizes the models that appear at the various steps of the transformation and describes the transformations between these models.

4.1   Overview

In order to implement the studied concepts through a model transformation, the source Model S and the destination Model D must be identified. In the scope of this paper, both models are component models.

- D is typically an existing component model close to the execution. It models a set of concrete concepts dc. These concepts are typically those described in Section 2.1: components, ports and assemblies.

- S typically models the same concepts dc but also additional abstract concepts sc; it is therefore a super-set of D. The sc concepts are typically

those found in scientific models as described in Section 2.2: parallel components with $M \times N$ method calls, collective communications *à la* MPI, data sharing ports, task farms, etc.

The idea is to let applications described by the user in S be transformed into semantically equivalent applications of D. As seen in Section 2.2 however, implementations of sc concepts are typically dependant on the execution resources; these must be taken into account by the transformation. As an additional constraint, these resources are typically known when the application is deployed and the transformation should therefore occur at that time. As a classical property of primitive component implementations is their binary distributability as a black box, their modification at deployment time should be avoided. Thus the model transformation described in this section transforms the assembly model rather than the whole component model.

4.2   A two step transformation

A first approach would be to implement the transformation as a single monolithic step. Such an approach does however lead to similar drawbacks as those of existing approaches. The introduction of a new concept in the S model or the modification of the D model are likely to require the rewrite of the whole transformation. The transformations of all sc concepts are tightly bound together and can hardly be provided by different actors.

We therefore advocate for the introduction of an intermediate Model I that borrows its additional concepts ic from skeletons and ADLs: genericity and connectors as first class entities. With this approach, the transformation from the S to the D model is split into two parts: from S to I, and then from I to D. In the transformation from S to I, the behavior of sc concepts is described using ic concepts. In the transformation from I to D, resources are taken into account to transform a limited set of well known abstract concepts (ic) into their concrete counterpart. As a result, the proposed transformation chain is described in Figure 3.
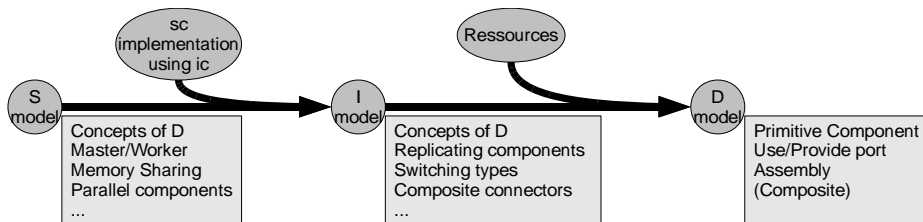

Figure 3: The S to D transformation chain.

Model I

Similarly to Model S, Model I is a super-set of D. In order to support the introduction of new kind of interactions between components, connectors are first class entities in the model and new port types can be described using bundle and collection port types. In order to support the introduction of new component implementations, genericity as described in [10] is introduced in the

```
bundle BundlePortType : SubPortType1(p1) {
    SubPortType1 p1;
    SubPortType2 p2;
}
```

Figure 4: Example of a bundle port type (`BundlePortType`) that contains two sub-ports (`p1` and `p2`) and that inherits `SubPortType1` through its sub-port `p1`.

model with support for replicating and switching component implementations. In order to let execution resources be taken into account, data-value parameters of generic components and connectors can be left free (no argument is provided for the parameter) to let their value be chosen when execution resources are known.

To summarize, six concepts are added to Model D to create Model I: genericity, bundle and collection port types, replicating and switching implementations and first class connectors. Let's describe them.

- A **generic component or connector** as defined in [10] is a type that accepts parameters, similarly to C++ templates or JAVA generics. These parameters can carry either a data-value or a type (data-type, component, port type or connector). The value of the parameters can then be used in the implementation of the component or connector. While [10] has introduced the concept of explicit specialization, this is not used in this paper; instead, the two main applications of this concept: replicating and switching implementations are introduced directly in the model.

- A **bundle port type** is defined by a set of sub-ports where each sub-port is named and typed by an existing port type. This is similar to the concept of structure used to define new data types in languages such as `C`. A bundle port type can inherit existing port types. This is done by attaching a sub-port to each inherited port type. The notation used to describe bundle port types in the remaining of this report is described in Figure 4.

- A **collection port type** is defined by two properties: an existing port type (`PT`) and an integer (N). The collection port type contains N ordered sub-ports of type `PT`. This is similar to the concept of array used to define new data types in most imperative languages. In the remaining of the report, the notation used to describe such a port type is `collection<PT,`N`>`.

- A **replicating component implementation** describes the content of a component as a set of instances of the same component. It is defined by two properties: the type of the internal component instances (`IC`) and the number of instances (N). In the remaining of the report, the notation used to describe instances of such a component is `replicating<IC,`N`>`. For each port `p` of `IC` whose type is `PT`, the replicating component has a port `p` of type `collection<PT,`N`>`. An example of replicating component instantiation is described in Figure 5.

- A **switching component implementation** describes a component whose implementation can be chosen amongst a list of possible implementations.
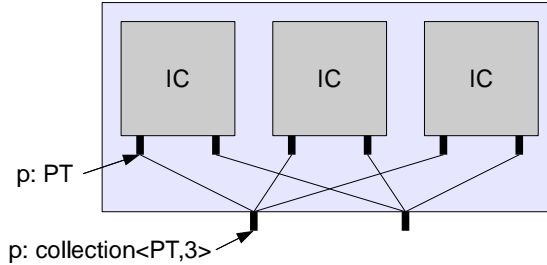
Figure 5: Content of an instance of the component `replicating<IC,3>`

It is defined by three properties: a list of component types (`CL`), an integer (N) that identifies the rank of the type to use in that list, and the mappings of the ports of the switching component to the ports of the types in `CL`. Each port of the switching component must be mapped to one port of the same type for each type of `CL`. The notation used to describe switching components in the remaining of the report is described in Figure 6. The notation used to describe instances of such a component is `SwitchingComponent<`N`>`.

```
component SwitchingComponent [
    T1 p1;
    T2 p2;
] switching {
    // 1st possibility: component C1 is used
    // in this case
    // port p1 is provided by port cp1 of C1
    // port p2 is provided by port cp2 of C1
    C1 [ p1 = cp1, p2 = cp2 ];

    // 2nd possibility: component C2 is used ...
    C2 [ ... ] ;
}
```

Figure 6: A switching component with two ports p1 and p2.

- A **connector** is a first class entity similar to a component with roles instead of ports. Instances of connectors that have their roles fulfilled by ports of component instances are called connections. Unlike components however, connector roles are not typed. The validity is a global property of the connection and not local to each role. Native connector implementations are used to model the type of connections supported in Model D. Composite, replicating and switching connector implementations, similar to their component counterpart are also available. In these two cases, the roles of the connector can either directly expose roles of internal connections or in the case where bundle or collection ports are connected to this role, a different role can be exposed for each sub-port.

When a primitive component exposes a port whose type is either a bundle or a collection port type, only the sub-ports (recursively until native port types are

reached) are seen from the implementation. This approach ensure that primitive implementations only deal with native port types. Therefore, implementations from Model I are also valid implementations in Model D.

Both the number of instances in replicating components and the implementation to choose in switching components are type parameters specified at instantiation. Leaving these parameters free is interesting to describe applications that can adapt themselves to the execution resources.

Because of the introduction of new meta-classes to describe these additions, the modeling of assemblies is slightly modified in I. Assemblies still describe instances of components but they also describe connections. These connections are instances of connectors and they contain fulfillments that reference the ports fulfilling each role of the connection. Two kind of fulfillment are possible: either the whole port of a component instance is used, or in the case where it is a collection, the fulfillment can reference the sub-ports of the collection and the connection is replicated.

4.3 Transformation implementation

This section details the two steps of the transformation: from S to I and then from I to D.

4.3.1 S to I transformation

The S to I transformation is based on the implementation of sc concepts using ic concepts. This implementation is very dependant on the exact concepts to implement – two examples are described in more details in Section 5. As these implementations are independent of each other, it is possible for them to be developed by specialists of the domain they handle (message passing, distributed memory sharing, etc.). In addition, since Model I is rather independent from Model D except for primitive component implementations, it should be easy to port the implementation of a feature from a destination model to another by providing implementations of primitive components in the new destination model.

The role of the transformation code itself is to map concepts of S into their equivalent in I, which has two aspects. The first aspect covers the transformation of concepts with different modeling in S and I. A typical example is where S has no concept of connector and only supports direct connections from a use port to a provide port; hence, a connection has to be mapped to an instance of a connector of Model I that supports an equivalent use/provide semantics. The second aspect deals with the mapping of native types of S that do not exist in I into elements of I. As it will be shown through an example in Section 5, this rewriting is quite straightforward.

4.3.2 I to D transformation

In the I to D transformation, all the ic concepts described earlier have to be transformed into their equivalent in D. This transformation depends on the execution resources for the choice of the value of type parameters that were left to be specified at deployment time (free parameters). This part of the transformation is however completely independent of the sc concepts that have

been implemented. It is thus made of two steps: the choice of values for free parameters and the transformation of *ic* concepts into *dc* concepts.

When specifying free parameters, it is possible that the transformation of a single type leads to multiple results with different values for these parameters depending on the context in which the type was used. In order to simplify this case, an intermediate Model I_i that models instances rather than types is introduced. The transformation from I to I_i consists in the (recursive) instantiation of a given component. In the case where the instantiated component is a composite, the instantiation produces a tree of components whose leaves are primitive components. When a type with free parameters is instantiated, their value are chosen accordingly to the available resources.

The second step of the transformation is applied on Model I_i. Replicating component instances are replaced by the number specified as parameter of component instances. Switching component instances and connections are replaced by instances of the type chosen by their parameter. Composite connections are replaced by a their content. Bundle and collection ports are replaced by their sub-ports. This is applied until no more transformation is possible.

The result of this transformation is an instance of Model I_i with no more free parameters, replicating or switching components, bundle or collection port and with native connectors only. The transformation to the D model is then quite straight forward, for each component instance its type is rebuild from its content and native connector instances are transformed into direct port connections.

5   Implementation and evaluation

The approach described in the previous section has been applied to a concrete case to study its feasibility. It has been used to extend the SCA component model with two additional features: data sharing ports and task farms. This extended SCA model is referred to as *exSCA*. As described in the previous section, the transformation is made of two steps: a first one that depends of the features added to the source model but not of the execution resources and a second one that depends of the execution resources but not of the added features. The intermediate model (*i.e.* the Model I applied to the case of SCA) is referred to as *iSCA*. This section starts by describing the first step and especially how the two concepts have been manually described in *iSCA*. Then, it focuses on the second step and especially on a JAVA-based prototype that implements the transformation engine.

5.1   *exSCA* to *iSCA* transformation implementation step

The first transformation step takes as input a set of components (primitive or composite) described in *exSCA* by the user and transforms them into *iSCA* components. The instances of *exSCA* are described with an XML syntax very similar to the syntax of plain SCA with a few additional notations for task farms and data sharing ports. An example is given in Figure 7. The syntax for *iSCA*, on the other hand is the one presented in the previous sections. It is not linked with the SCA syntax which makes the transformation engine for the intermediate model easier to port for another component model. Figure 8 displays the description of the example in *iSCA*.

```
Sharer.componentType:                 Accessor.composite:
<componentType>                       <composite name="Accessor">
  <shares name="shared1"/>              <accesses name="data1"/>
  ...                                   ...
</componentType>                      </composite>

Appli.composite:
<composite name="Appli">
  <component name="sh">
    <implementation.cpp header="Sharer.h" library="Sharer.so"/>
  </component>
  <component name="acc">
    <implementation.composite name="Accessor"/>
    <accesses name="data1">a/shared1</accesses>
  </component>
</composite>
```

Figure 7: `Appli` description in *exSCA*

```
Sharer.component:                     Appli.component:
component Sharer <                    component Appli [
  // type parameter                  ] composite {
  String lib                            // components instances
> [                                     Sharer<‘‘Sharer.so’’> sh;
  // ports                              Accessor acc;
  SharePort shared1;
  ...                                    // connections
] cpp { // C++ implem.                  SharedMem sh_shared1 {
  library=lib;                            // roles fulfillments
  header=‘‘Sharer.h’’;                    sharer = sh.shared1;
}                                         accessor = acc.data1;
                                        }
Accessor.component:                   }
component Accessor [
  AccessPort data1;
] composite {
  ... // composite implem.
}
```

Figure 8: `Appli` description in *iSCA*.

This step of the transformation relies on the implementation of four types directly in *iSCA*:

1. the `AccessPort` port type,

2. the `SharePort` port type,

3. the `SharedMem` connector, and
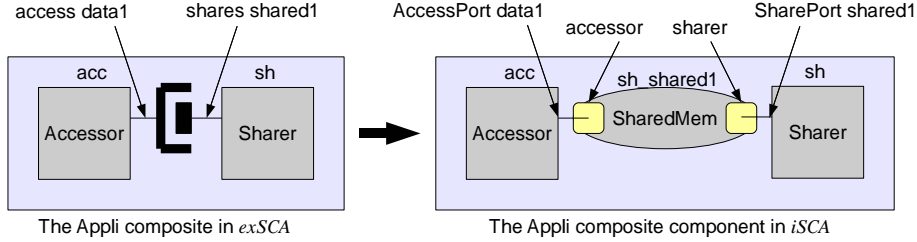
4. the `TaskFarm` component.

Figure 9: Transformation of the `Appli` composite from *exSCA* to *iSCA*.

The implementation of these types are presented in the following subsections. It applies the following algorithm to transform *exSCA* components into *iSCA* components:

- composites of *exSCA* are transformed into *iSCA* components with the same ports and a composite implementation containing the same component instances,

- primitive *exSCA* components are transformed into *iSCA* components with the same ports, a primitive implementation and type parameters for things (such as the library file to use) specified at instantiation in *exSCA*,

- references and services of *exSCA* are transformed into native `Use` and `Provide` ports of *iSCA* with their interface specified as a type parameter,

- shares and accesses ports of *exSCA* are transformed into instances of the `SharePort` and `AccessPort` port types,

- connections of a reference to a service in *exSCA* are transformed into instances of the native `MethodCall` connector,

- connections of a accesses to a shares in *exSCA* are transformed into instances of the `SharedMem` connector,

- farms of component `C` in *exSCA* are transformed into instances of the TaskFarm component with C as a type parameter in *iSCA*.

An example of transformation is shown in Figure 9 with the description of the associated descriptors in Figure 7 and 8. The remaining of this section describes in more details the data sharing and task farm description in *iSCA*.

5.1.1 Data sharing implementation

The data sharing implementation has been designed to provide the same behavior as found in [6]. In the referenced paper, the two port types are seen as use ports with a dedicated interface from primitive implementations. To provide a similar semantic, the port types are described as bundle ports containing a single use port as shown in Figure 10 and 11.

The `SharedMem` connector — defined in Figure 12 — used to connect these ports has two roles: `sharer` that must be fulfilled by one `SharePort` port and `accessor` that can be fulfilled by any number of `AccessPort` ports. The most

```
// C++ class definition            // Bundle definition
class Access {                     bundle AccessPort {
   virtual void* get_pointer()=0;     Use<Access> prim;
   virtual long  get_size()=0;     }
   virtual void  acquire()=0;
   virtual void  acquire_read()=0;
   virtual void  release()=0;
};
```

Figure 10: `AccessPort` C++ interface (left) and bundle port type (right).

```
// C++ class definition
class Share : public Access {
   virtual void associate(void* data, long size)=0;
   virtual void disassociate()=0;
};

// Bundle definition
bundle SharePort {
   Use<Share> prim;
}
```

Figure 11: `SharePort` C++ interface and bundle port type.

```
connector SharedMem [
   // roles and validity rules
   role sharer;
   role accessor;
   constraint (sharer.size == 1)
   and (sharer.each(type == SharePort))
   and (accessor.each(type == AccessPort));
] switching {
   LocalSharedMem [
      sharer = localSharer,
      accessor = localAccessor ];
   JuxmemSharedMem [
      sharer = juxmemSharer,
      accessor = juxmemAccessor ];
}
```

Figure 12: `SharedMem` switching connector implementation.

efficient implementation of this connector is however very dependant on the execution resources. Two implementations of this connector are described in Figure 12. The first one, `LocalSharedMem` is a simple implementation dedicated to the case where all interacting component instances are located in the same process. The second one, `JuxmemSharedMem` is dedicated to the case where instances are distributed on a computing grid and is based on JuxMem [5]. To support the choice between these multiple implementations, the `SharedMem`

connector is implemented as a switching type that let the choice of the actual implementation be a parameter as shown in Figure 12.
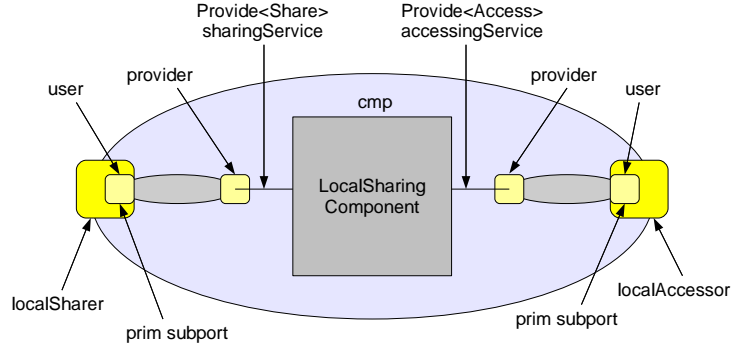


```
connector LocalSharedMem [
    role localSharer;
    role localAccessor;
    constraint localSharer.size == 1
    and localSharer.each(type == SharePort)
    and localAccessor.each(type == AccessPort);
] composite {
    LocalSharingComponent cmp;
    MethodCall sh_mc {
        provider = cmp.sharingService;
        user = localSharer.prim;
    };
    // a MethodCall connection for each accessor
    MethodCall acc_mc {
        provider = cmp.accessingService;
        user = localAccessor.prim.each;
    };
}
```

Figure 13: Graphical and textual representation of the `LocalSharedMem` composite connector.

The local implementation of the connector, described in Figure 13, is composite and contains a single component instance with two provide ports, one for data sharing and the other for memory accessing. This component simply maintains a reference (pointer) to the memory that has been declared as shared through the sharing port and provides it when requests are made on the accessing port. Two MethodCall connections are part of this assembly: one for each role of the connection (accessor and sharer). Each one has its provider role fulfilled by the corresponding port of the component instance while its use role is exposed as the sub-role of one of the connector roles.

The JuxMem based implementation shown in Figure 14 is similar, but instead of containing one single component instance, it contains two instances. The `sh` instance provides the sharing service by redirecting calls to JuxMem. The `acc` one similarly provides the accessing service. Having a single instance
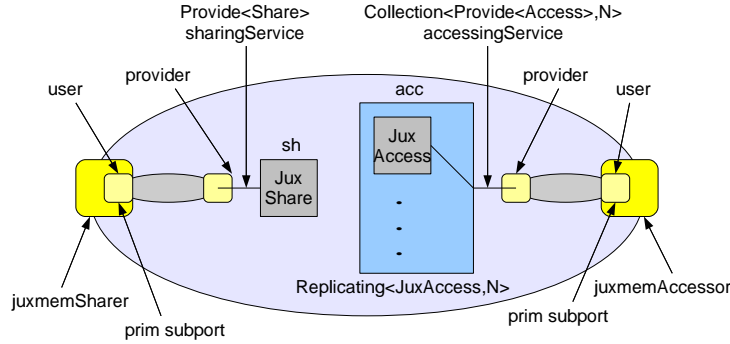
```
connector JuxmemSharedMem [
    role juxmemSharer;
    role juxmemAccessor;
    constraint juxmemSharer.size == 1
    and juxmemSharer.each(type == SharePort)
    and juxmemAccessor.each(type == AccessPort);
] composite {
    JuxShare sh;
    // N is left as a free parameter
    Replicating<JuxAccess,N> acc;
    MethodCall sh_mc {
        provider = sh.sharingService;
        user = juxmemSharer.prim;
    };
    MethodCall acc_mc {
        provider = acc.accessingService.each;
        user = juxmemAccessor.prim.each;
    };
}
```

Figure 14: Graphical and textual representation of the `JuxmemSharedMem` composite connector.

of this component while multiple components can be connected to the accessor role might however lead to performance penalties. This instance is thus a replicating component with the number of replicas left as a free parameter.

5.1.2   Task farm implementation

The task farms implementation has been designed to provide the same behavior as found in [11]. The `TaskFarm` component is a composite as shown in Figure 15. It contains a replicating component instance that replicates the `Worker` component provided as a parameter and a `TransportPattern` instance that embeds the logic for transmitting the requests to one of the worker instances.

Many implementations with different properties are possible for this transport pattern component. It is thus implemented with a switching type. A simple

```
component TaskFarm<
    component Worker
> [
    Provide<Service> service;
] composite {
    TransportPattern pattern;
    Replicating<Worker,N> workers;

    service = pattern.service;
    MethodCall mc {
        user =
            pattern.to_workers;
        provider =
            workers.service.each;
    };
}
```
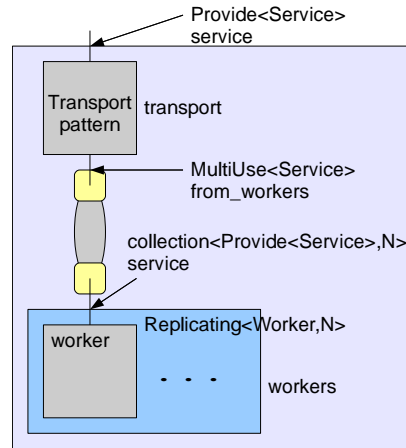

Figure 15: The `TaskFarm` composite

implementation is the `PrimitiveRandom` component that makes a centralized random choice of the worker to use: it is implemented as a primitive component.

```
component MultiLevelRandom [
    Provide<Service> service;
    MultiUse<Service> to_workers;
] composite {
    PrimitiveRandom this_level;
    Replicating<HierarchicRandom,N>
        sub_levels;

    service = this_level.service;
    to_workers =
        sub_levels.to_workers;
    MethodCall mc {
        user =
            this_level.to_workers;
        provider =
            sub_levels.service.each;
    };
}
```
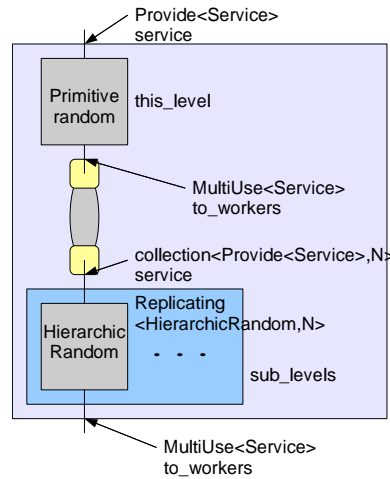

Figure 16: The `MultiLevelRandom` transport pattern

A more interesting implementation is the `HierarchicRandom` component. In order to let the number of levels in the hierarchy be chosen at deployment time, this implementation is recursive. This means that it is a switching component with two possible implementations: either the `PrimitiveRandom` component for a single level hierarchy or the `MultiLevelRandom` shown in Figure 16 for more levels of hierarchy. This `MultiLevelRandom` is a composite that contains one

instance of `PrimitiveRandom` that deals directly with requests and transmits them to one of the `HierarchicRandom` instances in the sub-levels replicating.

When an instance of the `HierarchicRandom` is transformed, it can either lead to a single level of hierarchy if the `PrimitiveRandom` implementation is chosen. In the case where the `MultiLevelRandom` implementation is chosen, it contains at least two levels of hierarchy. The instances of the bottom level are however again `HierarchicRandom` instances and the same transformation can recursively be applied leading to arbitrary deep hierarchies.

5.2   *iSCA* to SCA transformation implementation step

Once the application described in model *exSCA* has been transformed into an *iSCA* application, the second phase of the transformation is executed to transform it into a plain SCA application. While the first step of the transformation was done by hand; this transformation is done by a prototype transformation engine. This section describes the behavior of this prototype.

As described in Section 4, four distinct models appear in this phase: *iSCA*, plain SCA, and the models of their instances. For each kind of model (types and instances), the plain SCA part is common between the two models; the major part of these models is even not specific to SCA and could be reused for most component model. The modeling has thus been written as three layers: a generic meta-framework for component meta-modeling, a SCA specific part common to all models of this example and the third layer specific to *iSCA*.

These models were first written using kermeta, but this tool was still not complete at that time, they have thus been rewritten as with plain Java classes. These meta-classes contain only attributes and accessors, the logic of the transformation is implemented in external classes to make the implementation of other transformations possible. Factoring non SCA specific parts of the modeling in a component model agnostic meta-framework seems to have been a good decision as most part of the meta-classes can be described independently of the component model. The meta-framework contains 30 meta-classes, while the SCA specific part contains only 8 additional meta-classes and the *iSCA* one 16 additional meta-classes.

The transformation is implemented by following the steps described in Section 4.3.2. First, the *iSCA* application is instantiated: instances of Model *iSCA* are transformed into instances of Model $iSCA_{instances}$. In order to apply this transformation, the component that represents the whole application and that will thus be instantiated must be specified by the user; this component can not accept any parameter with no default value. The transformation then executes the following algorithm:

- if the instantiated element has free parameters, an external plugin is called that chooses the value for this parameter, currently the only plugin implemented requires the user to interactively provide a value,

- if the instantiated element is a primitive component or a native connector, instantiate it,

- if the instantiated element is composite, instantiate it and instantiate all its inner components instances and connections; if any of these element has free parameters, their value are chosen,

- if the instantiated element is a switching type, instantiate the type chosen by the switching parameter value,

- if the instantiated element is a replicating type, instantiate as many instances of the replicated type as specified by the replication parameter value.

The next next step is applied on Model $iSCA_{instances}$. Bundle and collection ports are replaced by their sub-ports and composite connections are replaced by their content. This is executed until the resulting assembly contains only native connections and sub-ports. In the last step, the types of each instance is regenerated and dumped into SCA XML.

5.2.1 Discussion

A transformation engine that transforms $iSCA$ applications using the concepts presented in this report into valid SCA applications has been implemented. The implementation of these transformation in this engine requires about 400 lines of JAVA code with about 150 lines for the first step, 100 lines for the second one and 150 lines for the last one.

The implementation of the `AccessPort` and `SharePort` port types, of the `SharedMem` connector and of the `TaskFarm` component made it possible to handle applications relying on both task farms and data sharing. A simple test application using both concepts has been written and successfully executed. As the transformation from $exSCA$ to $iSCA$ is still manual, this application has however been written directly in Model $iSCA$. As described in this section, the implementation of the task farm and data sharing have been done independently of each other.

However, by using this model, we ran into some limitations when connector are used together with composite components. As a matter of fact, only ports can be exposed by components and it is thus for example not possible to interact through shared memory between primitive components spread in multiple composites.

6 Conclusion

Component models appear very interesting for handling the increasing complexity of both scientific applications and resources. However, existing component models are still too close to resources, and it turns out to be quite difficult to experiment/implement extensions of component models.

This report has studied the usability of the MDE approach in order to be able to quite straightforwardly implement concept extension to component models. The studied methodology is based on the use of an intermediate model between the extensions and the concrete component models so as to be able to compose extensions and to have a clear separation between concept implementations and resource dependent issues. This intermediate model is derived from the concrete model with the addition of five concepts (bundle port type, collection port type, replicating component implementation, switching component implementation and connector). For many published extensions to component models, Model I seems to be powerful enough.

This report has studied two extensions to the SCA component model: data sharing and task farms. For both extensions, the transformation from *exSCA* to *iSCA* has been hand-coded: it is mainly string substitution. However, the transformation from *iSCA* to SCA has been implemented in JAVA with the limitation that all choices have been made manually.

Thus, the approach has proved to be usable for the development of advanced component models with concepts abstracted from the execution resources. Its main advantages relies in the possibility to develop extensions independently — there is no conflict between the memory sharing extension and the task farm extension — and to only focus on their interactions in a second phase. It also makes it possible to introduce such extensions without having to deeply modify any existing component runtime or deployment tool.

It does however still present some drawbacks. The dynamic modification of the assembly is not actually possible as the deployed assembly is different from the one described by the user. Another problem is that a new primitive components must be written for each value of its interface even if its behavior is dependant of that interface such as in the case of the transport pattern component. Moreover, algorithms for selecting free parameters have to be developed.

On the other hand, The implementation of the transformation engine has shown a real need for language dedicated to MDE. A first step would have been to use languages such as ECORE to describe the meta-models that appear at the various steps of the transformation. A second step would have been to use languages such as kermeta or QVT to describe the transformation themselves. The relative youth of the tools supporting these languages does however make it still rather difficult to debug transformations written by using them.

References

[1] M. Aldinucci, H. Bouziane, M. Danelutto, and C. Pérez. Towards software component assembly language enhanced with workflows and skeletons. In *Joint Workshop on Component-Based High Performance Computing and Component-Based Software Engineering and Software Architecture (CBHPC/COMPARCH 2008)*, 14-17 October 2008.

[2] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Kilpatrick, P. Dazzi, D. Laforenza, and N. Tonellotto. Behavioural skeletons in gcm: autonomic management of grid components. In *Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing*, Toulouse, France, Feb. 2008. IEEE. To appear.

[3] B. A. Allan, R. Armstrong, D. E. Bernholdt, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. McInnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and S. Zhou. A component architecture for high-performance scientific computing. *International Journal of High Performance Computing Applications*, 20(2):163–202, 2006.

[4] R. J. Allen. *A formal approach to software architecture.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1997. Chair-Garlan,, David.

[5] 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. Also available as an INRIA Research Report 4917: http://www.inria.fr/rrrt/rr-4917.html.

[6] 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.

[7] G. Antoniu, H. L. Bouziane, M. Jan, C. Pérez, and T. Priol. Combining data sharing with the master–worker paradigm in the common component architecture. *Cluster Computing*, 10(3):265 – 276, 2007.

[8] 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, 2009.

[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. Technical Report RR-6941, INRIA, 2009.

[11] H. L. Bouziane, C. Pérez, and T. Priol. Modeling and executing master-worker applications in component models. In *11th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, Rhodes Island, Greece, Apr. 2006.

[12] P. C. Clements. A survey of architecture description languages. In *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design*, page 16, Washington, DC, USA, 1996. IEEE Computer Society.

[13] M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, 2004.

[14] F. Budinsky D. Steinberg, M. Paternostro, and E. Merks. *EMF Eclipse Modeling Framework.* Eclipse Series. Addison-Wesley Professional., 2nd edition, Dec. 2008.

[15] M. Danelutto and G. Zoppi. Behavioural skeletons meeting services. In *Proc. of ICCS: Intl. Conference on Computational Science, Workshop on Practical Aspects of High-level Parallel Programming*, volume 5101 of *lncs*, pages 146–153, Krakow, Poland, Jun. 2008. springer.

[16] Z. Drey, c. Faucher, F. Fleurey, and D. Vojtisek. *Kermeta language reference manual*, 2006.

[17] EJB 3.0 Expert Group. *JSR 220 : Enterprise JavaBeans Version 3.0*, Dec. 2005.

[18] D. Garlan, R. Monroe, and D. Wile. Acme: an architecture description interchange language. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 7. IBM Press, 1997.

[19] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *The MPI-2 Extensions*, volume 2 of *MPI: The Complete Reference*. The MIT Press, Cambridge, MA, USA, Sep. 1998.

[20] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[21] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, W. Mann, and D. Bryan. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.

[22] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[23] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Fifth European Software Engineering Conference, ESEC '95 , Barcelona*, 1995.

[24] J. Magee, J. Kramer, and D. Giannakopoulou. Software architecture directed behaviour analysis. In *IWSSD '98: Proceedings of the 9th international workshop on Software specification and design*, page 144, Washington, DC, USA, 1998. IEEE Computer Society.

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

[26] D. McIlroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 138–150, 1968.

[27] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the c2 style. *SIGSOFT Softw. Eng. Notes*, 21(6):24–32, 1996.

[28] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[29] Object Management Group. *MDA Guide Version 1.0.1*, Jun. 2003.

[30] Object Management Group. *Meta Object Facility (MOF) Core Specification Version 2.0*, Jan. 2006.

[31] Object Management Group. *Object Constraint Language Version 2.0*, May. 2006.

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

[33] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.0*, Apr. 2008.

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

[35] C. Pérez, T. Priol, and A. Ribes. A parallel corba component model for numerical code coupling. In M. Parashar, editor, *GRID '02: Proceedings of the Third 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.

[36] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[37] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *The MPI Core*, volume 1 of *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 2 edition, Sep. 1998.

[38] B. Spitznagel and D. Garlan. A compositional approach for constructing connectors. In *Proceedings of the Working IEEE/IFIP Conference on software Architecture (WICSA 2001)*, pages 148–157, 2001.

[39] C. Szyperski, D. Gruntz, and S. Murer. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 2 edition, 2002.

[40] K. Zhang, K. Damevski, V. Venkatachalapathy, and S. G. Parker. Scirun2: A cca framework for high performance computing. *High-Level Programming Models and Supportive Environments, International Workshop on*, 0:72–79, 2004.