Design and Implementation of an Environment for Component-Based Parallel Programming

Francisco Heron de Carvalho Junior[1], Rafael Dueire Lins[2], Ricardo Cordeiro Corrêa[1], Gisele Araújo[1], and Chanderlie Freire de Santiago[1]

[1] Departamento de Computação, Universidade Federal do Ceará
Campus do Pici, Bloco 910, Fortaleza, Brazil
{heron,correa,gisele,cfreire}@lia.ufc.br

[2] Depart. de Eletrônica e Sistemas, Universidade Federal de Pernambuco
Av. Acadêmico Hélio Ramos s/n, Recife, Brazil
rdl@ufpe.br

Abstract. Motivated by the inadequacy of current parallel programming artifacts, the # component model was proposed to meet the new complexity of high performance computing (HPC). It has solid formal foundations, layed on category theory and Petri nets. This paper presents some important design and implementation issues on the implementation of programming frameworks based on the # component model.

1 Introduction

Clusters and grids have brought the processing power of high performance computing (HPC) architectures to a wide number of academic and industrial users, bringing new challenges to computer scientists. Contemporary parallel programming techniques that can exploit the potential performance of distributed architectures, such as the message passing libraries MPI and PVM, provide poor *abstraction*, requiring a fair amount of knowledge on architectural details and parallelism strategies that go far beyond the reach of users in general. On the other hand, higher level approaches, such as parallel functional programming languages and scientific computing parallel libraries do not merge efficiency with generality. Skeletal programming has been considered a promising alternative, but several reasons have made difficult its dissemination [11]. In fact, the scientific community still looks for parallel programming paradigms that reconciles portability and efficiency with generality and high-level of abstraction [4].

In recent years, the HPC community has tried to adapt component technology, now successfully applied in business applications in dealing with software complexity and extensibility, to meet the needs of HPC applications. These efforts yielded CCA and its frameworks [2], P-COM [20], Fractal [3], et cetera [24]. Besides being a potential alternative to reconcile abstraction, portability, generality, and efficiency in parallel programming, components leverage *multi-disciplinary*, *multi-physics*, and *multi-scale* software for HPC [5], possibly targeting heterogenous execution environments that are enabled for grid, cluster, and capability computing [13].

The most important challenge to make components suitable for HPC relies on their support for parallel programming [1, 10]. Surprisingly, parallel programming based on the current approaches for supporting peer-to-peer components interaction is not suitable for performance demands of HPC software that are not embarrassingly parallel [1, 10]. Unfortunately, the presence of complex process interactions are common in modern HPC software. For this reason, HPC components models and architectures have been extended for supporting non-trivial forms of parallelism [17, 12, 22, 1, 3]. However, such approaches for parallelism do not reach generality of message-passing based parallel programming. In addition, they are influenced by the common trend of lower level parallel programming approaches to treat processes, and not only concerns, as basic units of software decomposition. We consider that this is one of the main reasons of the difficulty in adapting current software engineering techniques for the development of parallel programs. Software engineering approaches have appeared in the context of sequential software, where processes do not exist. We advocate orthogonality between processes and concerns [8]. Thus, they cannot be appropriately viewed under the same software decomposition dimension.

The # component model was primarily developed for general purpose parallel programming, taking the orthogonality between processes and concerns as a basic design premise. Unlike most of the recently proposed components approaches for HPC, it does not take inspiration in conventional component models. It has origins in the coordination model of Haskell$_\#$ [7], a parallel extension to the functional language Haskell. Most possibly, any component model may be defined in the # component model. Besides to dealing with parallel programming in a natural way, the # component model is influenced by modern ideas regarding the notion of separation of concerns [8], one of the main driving forces for recent advances in software engineering technologies [21]. Indeed, cross-cutting composition of concerns is supported. The # component model tries to achieve a higher level of abstraction by employing skeletal programming through *parameterized component types*. This paper intends to present the design of a framework based on the # component model for parallel programming targeting HPC software on top of IBM Eclipse Platform.

In what follows, Section 2 introduces the basic principles behind the # component model, comparing it to other HPC component approaches. Section 3 depicts the general design of # frameworks. Section 4 presents the design of a general purpose parallel programming framework. Section 5 concludes this paper, describing ongoing and lines for further works regarding the implementation of programming environments based on the # component model.

2 The # Component Model: Principles and Intuitions

Motivated by the success of the component technology in software industry, scientific computing community has proposed component models, architectures and frameworks for leveraging *multi-disciplinary*, *multi-physics*, and *multi-scale* HPC software, possibly targeted at HPC architectures enabled for grid, cluster, and capability computing [24]. Unfortunately, their requirements for the support of parallelism and high processing efficiency make usual forms of peer-to-peer component interaction unsuitable [10, 5]. For this reason, specific parallel programming extensions have been proposed to current component technology. For example, CCA specification includes SCMD[3] extensions for supporting SPMD style of parallel programming [1]. PARDIS[17], PADICO[12], and GridCCM[22] have also adopted a similar concept for supporting parallel objects inside components. Fractal proposes collective ports that may dispatch method calls to a set of inner parallel components [3]. In general, such extensions cover requirements of a wide range of parallel programs in certain domains of interest, but they do not provide full generality of message-passing parallel programming. It is usual to find papers on HPC components that include "support for richer forms of parallelism" in the list of lines for further works. For example, CCA attempts to move from SCMD to MCMD, a simple conceptual extension, but difficult to reach in practice. In fact, to support general purpose parallel programming is still a challenge for HPC component technology.

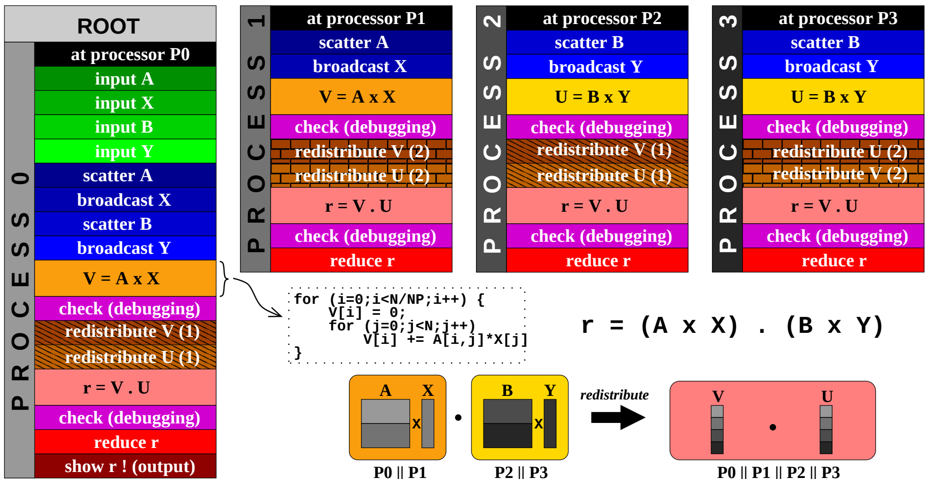
The ***inductive*** approach to augment component technology with new parallel programming extensions breaks down conceptual homogeneity of component models, making them more complex to be grasped by informal means and mathematically formalized. The # component model comes from the "opposite direction", taking a ***deductive*** generalization of channel-based parallel programming for supporting a suitable notion of component. The # component model has its origins in Haskell$_{\#}$ [7], a parallel extension to the functional language Haskell, inheriting their design premisses, including Petri nets translation [9].

2.1 From Processes to Concerns

The basic principles behind the # component model come from message passing programming, notably represented by PVM and MPI. They have successfully exploited peak performance of parallel architectures, reconciling generality and portability, but with hard convergence with software engineering disciplines for supporting productive software development. The following paragraphs introduce fundamental principles behind the # component model: *the separation of concerns through process slicing*; and *orthogonality between processes and concerns as units of software decomposition.* Familiarity of readers with parallel programming is needed to understand the # component model from the intuition behind their underlying basic principles. Induction from examples must be avoided. Readers may concentrate on fundamental ideas and try to build their own examples from their experience and interests. Figures 1 and 2 complementarily present a simple parallel program that is used for exemplifying the idea of slicing processes by concerns. Let **A** and **B** be $n \times n$ matrixes and X and Y be vectors. The parallel program computes $(\mathbf{A} \times X^T) \bullet (\mathbf{B} \times Y^T)$.

We have searched for the fundamental reasons that make software engineering disciplines too hard to be applied for parallel programming, concluding that

[3] Single Component Multiple Data



```
for (i=0;i<N/NP;i++) {
   V[i] = 0;
   for (j=0;j<N;j++)
      V[i] += A[i,j]*X[j]
}
```

Fig. 1. Slicing a Simple Parallel Program by Concerns

they reside on the tendency to mix *processes* and *concerns* in the same dimension of software decomposition, due to the traditional process-centric perspective of parallel programming practice. Software engineering disciplines assume concerns as basic units of software decomposition [21]. We advocate that processes and concerns are orthogonal concepts. Without loss of generality, aiming at to clarify intuitions behind the enunciated orthogonality hypothesis, let $\mathcal{P}$ be an arbitrary parallel program formed by a set $\{p_1, p_2, \ldots, p_n\}$ of processes that synchronize through message-passing. By looking at each process individually, it may be split in a set of slices, each one addressing a concern. Figure 1 shows an example of process slicing in a simple parallel program. Examples of typical concerns are: (a) a piece of code that represents some meaningful calculation, for example, a local matrix-vector multiplication; (b) a collective synchronization operation, which may be represented by a sequence of send/recv operations; (c) a set of non-contiguous pieces of code including debugging code of the process; (d) the identity of the processing unit where the process executes; (e) the location of a process in a given process topology. The reader may be convinced that there is a hierarchical dependency between process slices. For instance: (a) the slice representing collective synchronization operation is formed by a set of slices representing send/recv point-to-point operations; (b) a local matrix-vector multiplication slice may include a slice that represent the local calculation performed by the process and another one representing the collective synchronization operation that follows it. If we take all the processes into consideration, it is easy to see existence of concerns that cross-cuts processes. For example: (a) the concern of parallel matrix-vector multiplication includes all slices, from individual processes, related to local matrix-vector multiplication; (2) the concern of process-to-processor allocation is formed by the set of slices that define the identities of processors where each process executes. It is easy to see that, from the overall perspective of processes, most of slices inside individual processes does not make sense when observed in isolation. Individually, they do not define concerns in the overall parallel program. The cross-cutting nature of decompositions based on concerns and processes strongly enforces the orthogonality hypothesis.

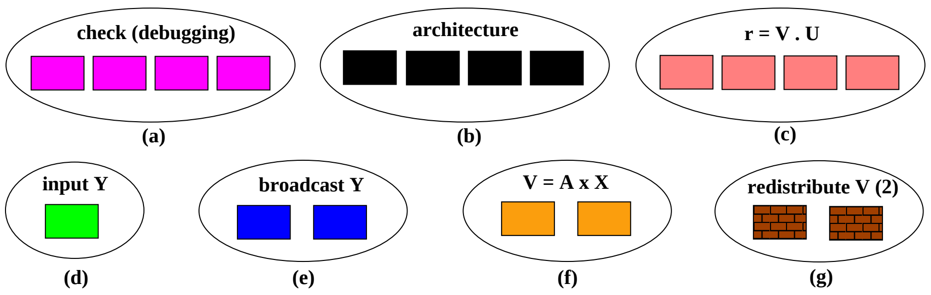
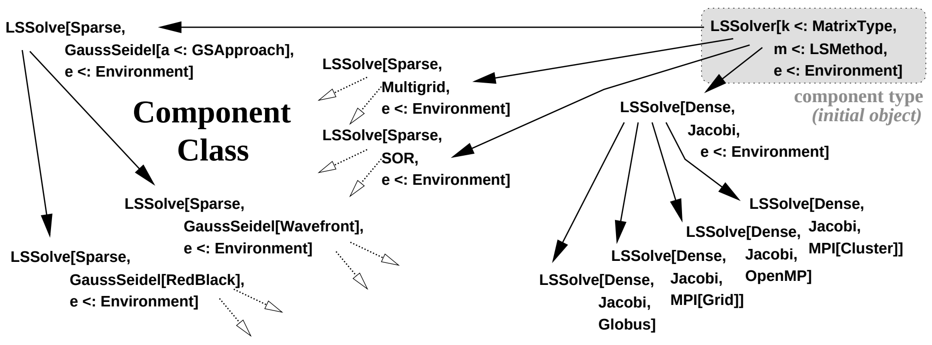


Fig. 2. #-Components From The Example in Figure 1

Above, some examples of #-components extracted from slicing of the parallel program in Figure 1. Some of them are non-functional concerns: (a) debugging code and (b) process-to-processor mapping. The #-components $V = A{\times}X$ and $r = V{\bullet}U$ addresses functional concerns: parallel matrix-vector multiplication and parallel dot product, respectively. The #-components "broadcast Y" and "redistribute V (2)" address data distribution concerns, acting as synchronization protocols. The #-component "input Y" is a local concern of a process (root) that is responsible to read vector Y.

The # component model moves parallel programming from the process-based perspective to a concern-oriented one. In fact, through a *Front-End*, # programmers may build applications through composition of concerns. Then, a *Back-End* may synthesize the process topology of the intended parallel program. A ***#-component*** is a software entity that encapsulates a concern. Such definition covers usual notions of components, because concerns are elementary units of software decomposition. The *units* of a #-component correspond to the slices (of processes) that constitutes its addressed concern. A #-component may be inductively built from other #-components through unification of their units, forming units of the resultant #-component. Thus, units also form a hierarchical structure, attempting to resemble hierarchical structure of process slices, where units may be formed by other units (unit slices). Sharing between components is supported through *fusion of unit slices* in unification. Sharing of data structures is a fundamental feature for ensuring high performance in scientific software. Another component model that supports sharing between components is Fractal [6]. The protocol of a unit is specified by a labelled Petri net whose labels are identifiers of their slices. It determines a Petri net formal language which dictates the possible activation traces for slices. Intuitively, it defines the order in which processes execute their functional slices. Petri nets allows for analysis of formal properties and performance evaluation of parallel programs.



LSSolver[k <: MatrixType, m <: LSMethod, e <: Environment]
component type (initial object)
Component Class
LSSolve[Sparse, GaussSeidel[a <: GSApproach], e <: Environment]
LSSolve[Sparse, Multigrid, e <: Environment]
LSSolve[Sparse, SOR, e <: Environment]
LSSolve[Sparse, GaussSeidel[Wavefront], e <: Environment]
LSSolve[Sparse, GaussSeidel[RedBlack], e <: Environment]
LSSolve[Dense, Jacobi, e <: Environment]
LSSolve[Dense, Jacobi, MPI[Cluster]]
LSSolve[Dense, Jacobi, OpenMP]
LSSolve[Dense, Jacobi, MPI[Grid]]
LSSolve[Dense, Jacobi, Globus]

Fig. 3. Component Class for LSSolver

A component class for LSSolver components, represented by the *component type* inside the gray box. Lowercase identifiers are formal parameters. The notation $x <: C$ says that x may be replaced by any sub-type of component type C. Arrows indicate instantiations, which replace a formal parameter by an actual component. For example, there are four components that implement solutions for *dense linear systems* using the Jacobi iterative method. They target different architectures. The component type MPI is parameterized by the intended architecture (MPI[a <: Architecture]).

2.2 Skeletal Programming and Parameterized Component Types

The simpler form of abstraction in # programming is to hide lower level operations in higher level ones encapsulated in components. For example, a programmer that makes use of a component LSSolver for solving a linear system $A.x = B$ does not need to be aware about synchronization operations inside the component, resembling linear algebra parallelized libraries. Partitioning of parallel programs by concerns suggests richer abstraction mechanisms, such as skeletal programming through component types, representing a class of components that address the same concern through distinct implementations, each one appropriate to a specific execution environment. For example, there may exist several possible implementations for LSSolver, adapted to specific parallel architectures, process topologies, and density properties of matrix A. Such parameters are known by programmers before execution. Figure 3 exemplifies a component class for dealing with implementations of LSSolver. The idea of *parameterized* component types have appeared due to the formalization of the # component model using Theory of Institutions [15], firstly intended to study its formal properties and to formalize the notions of component types and their *recursive composition*. Because Theory of Institutions have been used to formalize logical independence in algebraic software specification, some ideas from this context have been brought to # programming, including parameterized programming [14], which gives rise to polymorphic component types.

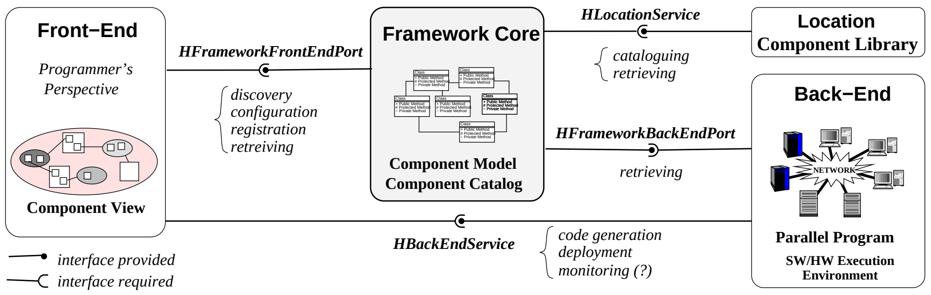
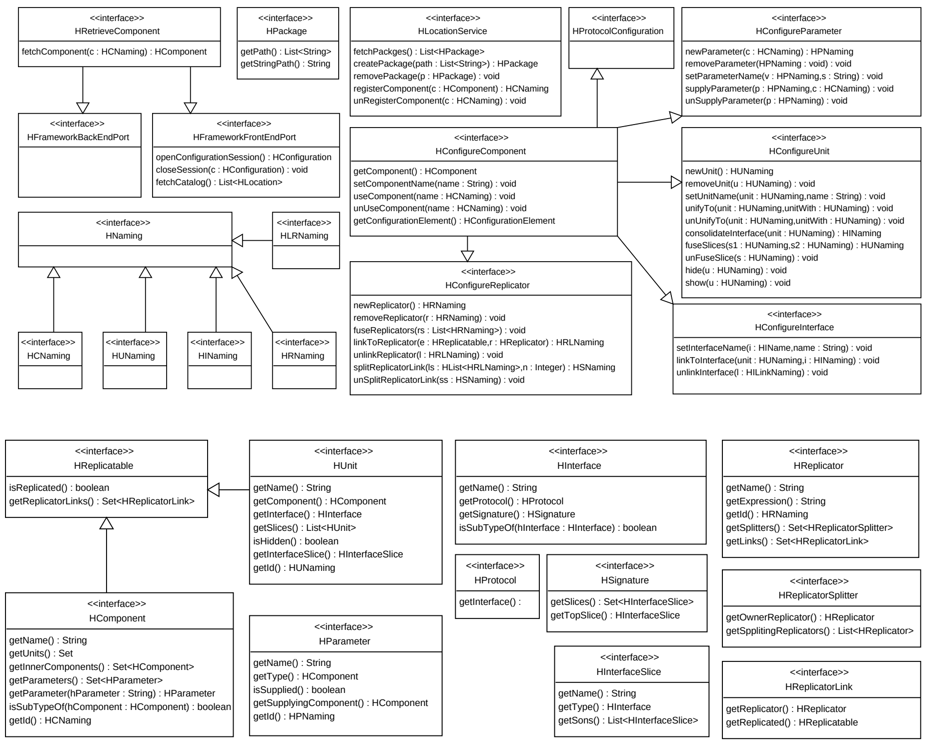


Fig. 4. The # Framework and #-Components Life Cycle

3 An Architecture for # Programming Frameworks

Figure 4 depicts an architecture proposal for # programming frameworks. Like CCA and Fractal, # compliant frameworks be built from instantiation of a set of interfaces that define the *# component model architecture*, whose interfaces are depicted in the UML diagram of Figure 5. Frameworks control life cycle of components by means of the interfaces that components must provide. CCA targets simplicity, by adopting a lightweight interface for components to interact with the framework, including only the method *setServices*, where programmers register their *uses ports* and *provides ports* for dynamic binding. Fractal compliant components also support dynamic bindings, also targeting hierarchical composition from primitive components. As already shown, the # component model supports recursive composition, but their "bindings" are static, which, at a first glance, appears to restrict the application domain of #-components. For this reason, the # component model is not yet proposed for general distributed applications, but only for applications in high performance computing domain. In fact, most of parallel programs are static, avoiding performance overheads of run-time control. However, #-frameworks can still deal with dynamic execution scenarios needed by HPC applications. In fact, static configuration does not imply static execution. A #-framework may encapsulate predictable dynamic adaptations of programs, supported by some underlying programming artifact, as concerns of #-components.

Unlike CCA and Fractal, # programmers does not glue components to frameworks by direct implementation of architecture interfaces for programming modules, but using an architecture description language (ADL). This is motivated by the requirement to place coordination and computation concerns at separate programming layers, and to facilitate support for overlapping composition, since current usual programming artifacts does not support to overlap implementation of modules. As depicted in Figure 4, the *Front-End* of a #-framework deals with component views of a component model, managed by the *Framework-Core* and accessed by the Front-End through the interface *HFrameworkFrontEndPort*. The *Framework-Core* is also responsible to manage a library of #-components placed at registered locations. In Fractal and CCA, programmers directly manipulates the *component model.* The # component model delegates to #-frameworks to define one or more appropriate ADL's, managed by distinct *Front-End*'s. ADL's may be *graphical*, using visual metaphors, or *textual.* A textual ADL may be XML-based, making possible interoperability at the level of component views. Indeed, general framework interoperability can be achieved at the level of component models. In visual programming, the MVC (Model-View-Controller) pattern is a good design pattern for *interpreting component views* as *component models.*



<<interface>> HRetrieveComponent
fetchComponent(c : HCNaming) : HComponent
<<interface>> HPackage
getPath() : List<String>
getStringPath() : String
<<interface>> HLocationService
fetchPackges() : List<HPackage>
createPackage(path : List<String>) : HPackage
removePackage(p : HPackage) : void
registerComponent(c : HComponent) : HCNaming
unRegisterComponent(c : HCNaming) : void
<<interface>> HProtocolConfiguration
<<interface>> HConfigureParameter
newParameter(c : HCNaming) : HPNaming
removeParameter(HPNaming : void) : void
setParameterName(v : HPNaming,s : String) : void
supplyParameter(p : HPNaming,c : HCNaming) : void
unSupplyParameter(p : HPNaming) : void
<<interface>> HFrameworkBackEndPort
<<interface>> HFrameworkFrontEndPort
openConfigurationSession() : HConfiguration
closeSession(c : HConfiguration) : void
fetchCatalog() : List<HLocation>
<<interface>> HConfigureComponent
getComponent() : HComponent
setComponentName(name : String) : void
useComponent(name : HCNaming) : void
unUseComponent(name : HCNaming) : void
getConfigurationElement() : HConfigurationElement
<<interface>> HConfigureUnit
newUnit() : HUNaming
removeUnit(u : HUNaming) : void
setUnitName(unit : HUNaming,name : String) : void
unifyTo(unit : HUNaming,unitWith : HUNaming) : void
unUnifyTo(unit : HUNaming,unitWith : HUNaming) : void
consolidateInterface(unit : HUNaming) : HINaming
fuseSlices(s1 : HUNaming,s2 : HUNaming) : HUNaming
unFuseSlice(s : HUNaming) : void
hide(u : HUNaming) : void
show(u : HUNaming) : void
<<interface>> HNaming
<<interface>> HLRNaming
<<interface>> HConfigureReplicator
newReplicator() : HRNaming
removeReplicator(r : HRNaming) : void
fuseReplicators(rs : List<HRNaming>) : void
linkToReplicator(e : HReplicatable,r : HReplicator) : HRLNaming
unlinkReplicator(l : HRLNaming) : void
splitReplicatorLink(ls : HList<HRLNaming>,n : Integer) : HSNaming
unSplitReplicatorLink(ss : HSNaming) : void
<<interface>> HConfigureInterface
setInterfaceName(i : HIName,name : String) : void
linkToInterface(unit : HUNaming,i : HINaming) : void
unlinkInterface(l : HILinkNaming) : void
<<interface>> HCNaming
<<interface>> HUNaming
<<interface>> HINaming
<<interface>> HRNaming
<<interface>> HReplicatable
isReplicated() : boolean
getReplicatorLinks() : Set<HReplicatorLink>
<<interface>> HUnit
getName() : String
getComponent() : HComponent
getInterface() : HInterface
getSlices() : List<HUnit>
isHidden() : boolean
getInterfaceSlice() : HInterfaceSlice
getId() : HUNaming
<<interface>> HInterface
getName() : String
getProtocol() : HProtocol
getSignature() : HSignature
isSubTypeOf(hInterface : HInterface) : boolean
<<interface>> HReplicator
getName() : String
getExpression() : String
getId() : HRNaming
getSplitters() : Set<HReplicatorSplitter>
getLinks() : Set<HReplicatorLink>
<<interface>> HProtocol
getInterface() :
<<interface>> HSignature
getSlices() : Set<HInterfaceSlice>
getTopSlice() : HInterfaceSlice
<<interface>> HReplicatorSplitter
getOwnerReplicator() : HReplicator
getSpplitingReplicators() : List<HReplicator>
<<interface>> HComponent
getName() : String
getUnits() : Set
getInnerComponents() : Set<HComponent>
getParameters() : Set<HParameter>
getParameter(hParameter : String) : HParameter
isSubTypeOf(hComponent : HComponent) : boolean
getId() : HCNaming
<<interface>> HParameter
getName() : String
getType() : HComponent
isSupplied() : boolean
getSupplyingComponent() : HComponent
getId() : HPNaming
<<interface>> HInterfaceSlice
getName() : String
getType() : HInterface
getSons() : List<HInterfaceSlice>
<<interface>> HReplicatorLink
getReplicator() : HReplicator
getReplicated() : HReplicatable

Fig. 5. The Framework Core and the # Component Model Architecture

The *Back-End* of #-frameworks synthesizes a parallel program from the *component model*, targeting their supported execution environments. This is needed due to orthogonality between #-components and processes, a fundamental distinction from CCA and Fractal, where component models are the units of programming and deployment. For this reason, CCA and Fractal do not need a *Front-End* and a *Back-End* (Figure 4). In fact, #-frameworks act as bridges between *component views* and parallel programming artifacts. The # component model does not intend to be "yet another parallel programming technology", but to be a components-based layer on top of which existing ones can take advantage of software engineering disciplines. It is conjectured that any programming technology may be defined in terms of the # component model, including CCA and Fractal frameworks. The interoperability hypothesis has been verified by experimental evaluation with #-frameworks. The *Back-End* of #-frameworks may perform optimization steps for reducing synchronization costs in the resultant parallel program. For example, if all slices of a #-process are programmed in the same language, they can be fused (inlined) in a single procedure, avoiding unnecessary costs of procedure calls and improving cache behavior. It is intended that the synthesized parallel program be similar or better than programmed by hand, since programmers have explicit control over all parallelism concerns.

A #-framework defines a set of specialized ***components kinds***, each one containing a set of component types with an intended meaning, which may imply in different visual metaphors and model interpretations at the *Front-End* and *Back-End* sides. In fact, component kinds are supported by the specialization of the interfaces of the *# component model architecture*, presented in Figure 5. The use of component kinds for designing of # compliant PSE's (*Problem Solving Environments*) that uses visual metaphors that are near to the knowledge of specialists have been investigated. The # component model goes far beyond the idea to raise connectors to first-class citizens [23], by promoting them to components. For example, a CCA binding could be implemented as a #-component BINDING. Such approach leads to uniformity of concepts. Fractal also exercises the idea of components as connectors, by means of *composite bindings*, but *primitive bindings* are not components, breaking homogeneity. The # connectors are *exogenous*[19], like in P-COM, while they are *endogenous* in CCA and Fractal.

4 A # Environment for Parallel Programming

Now, the design of a #-framework for general purpose parallel programming on top of common message-passing programming technologies, called HPE (*# Programming Environment*), is presented. It is an extension to the Eclipse framework. GEF (*Graphical Editing Framework*) has been used to build an ADL for dealing with visual configuration of #-components. GEF adopts the MVC (Model-View-Controller) design pattern. A XML format for describing configurations have been also designed. The framework complies to the # component model architecture, specializing it to support the *component kinds* supported by HPE, including component types for *qualifiers*, *architectures*, *environments*, *data structures*, *computations* and *synchronization* concerns. Some built-in component types are supported by the framework, whose sub-typing hierarchy is depicted in Figure 6. For each component kind there is a proper *top* component type. Programmers may build their own component types and programs on top of component kinds and component types natively supported by the framework.

Figure 7(a) presents a screenshot of the the HPE's *Front-End*, showing a # program (application component) that solves a linear system $A \times x = B$. Inputs (matrix A and vector B), and output (vector x), are retrieved/saved in a remote data center defined by the component DATACENTER. It will illustrate the idea of *component kinds*. Input data is distributed across processes using the collective communication component Scatter, while output is joined in the *root* process using Gather. In Figure 7(b), the protocol of the unit *peer* is depicted, defining that scattering operations must be performed in parallel, followed by the solution computation and gathering.

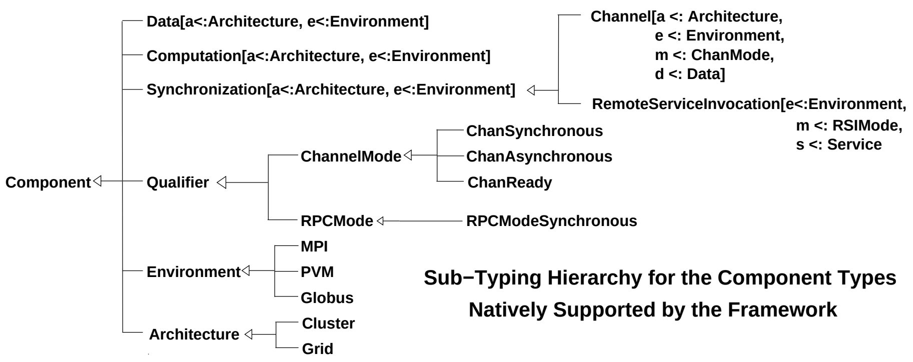
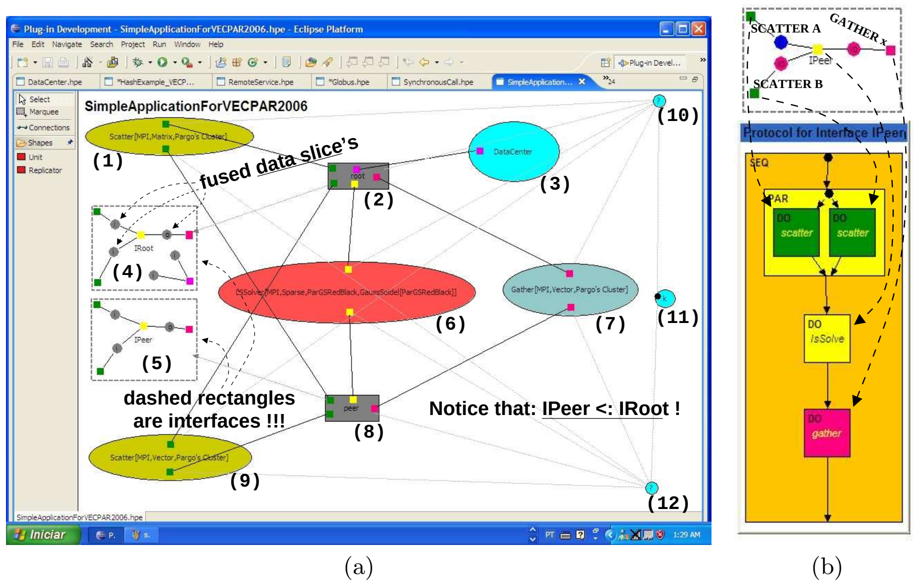

Data[a<:Architecture, e<:Environment]
Computation[a<:Architecture, e<:Environment]
Synchronization[a<:Architecture, e<:Environment]
Channel[a <: Architecture, e <: Environment, m <: ChanMode, d <: Data]
RemoteServiceInvocation[e<:Environment, m <: RSIMode, s <: Service
Component
Qualifier
ChannelMode
ChanSynchronous
ChanAsynchronous
ChanReady
RPCMode
RPCModeSynchronous
Environment
MPI
PVM
Globus
Architecture
Cluster
Grid
Sub−Typing Hierarchy for the Component Types Natively Supported by the Framework

Fig. 6. The # Component Kinds of the Framework

4.1 Component Kinds

Qualifier components address non-functional concerns intended to describe characteristics of components. In practice, they may act selectively among components in a component class, allowing programmers to control choice of component instances from its representant component type. For example, suppose a component class for point-to-point communication channels, represented by the component type Channel$[\cdots, mode <:$ ChannelMode]. The syntax says that, among other parameters (reticences), their component instances may vary according to the intended communication semantics. For that, the parameter *mode* must be supplied with a sub-type of the qualifier component ChannelMode, which comprises two units, respectively intended to be slices of sender and receiver units of a channel component. The natively supported sub-types of ChannelMode in the framework are ChanSynchronous, ChanAsynchronous, and ChanReady. Programmers may define other channel modes. The component type Channel$[\cdots,$ ChanSynchronous] represents the class of channel components with synchronous semantics. In Figure 7, qualifier components are also used to describe the solution method in a LSSolver component.

Architecture components intend to describe parallel architectures where #-components intends to run. Their units are their processing nodes. Using sub-typing capabilities, supported architectures must be organized in hierarchies, in order to be classified according to their common characteristics. For example, a Cluster architecture component type may be specialized in component types referring to common cluster designs, possibly distinguished by the processor


Plug-in Development - SimpleApplicationForVECPAR2006.hpe - Eclipse Platform
SimpleApplicationForVECPAR2006
Select
Marquee
Connections
Shapes
Unit
Replicator
Scatter[MPI,Matrix,Pargo's Cluster]
root
DataCenter
IRoot
IPeer
LSSolver[MPI,Sparse,ParGSRedBlack,GaussSeidel[ParGSRedBlack]]
Gather[MPI,Vector,Pargo's Cluster]
peer
Scatter[MPI,Vector,Pargo's Cluster]
fused data slice's
dashed rectangles are interfaces !!!
Notice that: IPeer <: IRoot !
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
(9)
(10)
(11)
(12)
SCATTER A
SCATTER B
GATHER x
IPeer
Protocol for Interface IPeer
SEQ
PAR
DO scatter
DO scatter
DO IsSolve
DO gather
(a)
(b)

LEGEND: **(1)** Component SCATTER[MPI,MATRIX,PARGO'S CLUSTER]; **(2)** Unit *root*; **(3)** Component DATACENTER; **(4)** Interface *IRoot* (of unit *root*); **(5)** Interface *IPeer* (of unit *peer*); **(6)** Component LSSOLVER[MPI, SPARSE, GAUSSSEIDEL[PARREDBLACKMETHOD]]; **(7)** Component GATHER[MPI, VECTOR, PARGO'S CLUSTER]; **(8)** Unit *peer*; **(9)** Component SCATTER[MPI, VECTOR, PARGO'S CLUSTER]; **(10)** Replicator ($\times 1$); **(11)** Split replicator; **(12)** Replicator ($\times n$).

Fig. 7. The Visual Configuration of a Simple #-Component (Screen Shot)

type, communication network type, homogeneity/heterogeneity, and so on. At the leaves of the hierarchy, there are component types for describing specific clusters. Thus, a programmer may target a class of architectures by using architecture types at non-leave nodes. A specific architecture could be in more than one intersecting classes. Similarly, grid-based architectures could be classified. The example in Figure 7 runs in a specific cluster, named PARGO'S CLUSTER.

Environment components define the parallelism enabling software technology intended for a component. Typical examples are message-passing libraries, such as MPI and PVM, for cluster and capability computing, and Globus and OurGrid, for grid computing. Some MPI implementations also target grids. Notice that a pair architecture/environment defines a complete parallel run-time execution context for a component. Hierarchies of component types may also be used to define classes of environments of special interest, for example, software technologies for enabling message-passing or bag-of-tasks parallelism.

Data components are formed by one unit, whose interface is attached to a SIDL (Scientific Interface Description Language) interface. SIDL has been supported by the Babel toolkit [18] to be a neutral language for specification of CCA components interfaces. Sub-typing is supported for data components, resembling multiple inheritance in object-oriented programming. For that, a data component type D must be composed from a set of data components supertypes, whose units becomes slices of the units of D. In Figure 7, there are data component types VECTOR and MATRIX, sub-types of the component type DATA.

Synchronization components allows inter-process communication. There are synchronization components for dealing with *point-to-point message-passing* (the usual *send* and *receive* primitives), collective communication (structure parallel programming [16]), and remote service invocation (such as RPC, RMI, and so on). The class of all channel components is represented by the component type CHANNEL[a<:ARCHITECTURE, s<:ENVIRONMENT, d<:DATA, m<:CHANMODE]. A highly tuned member of the component class of CHANNEL[$\cdots$] may be specific to a given architecture, environment, data type, and channel semantics. The class of *remote service invocation* components is represented by the component type REMOTESERVICEINVOCATION[e<:ENVIRONMENT, m<:RSIMODE, s<:SERVICE]. They comprise two externally visible units: *client* and *server*. The activation of a *client slice* is a *null* operation. Client slices only carries *stub objects* for each interface provided by the service. A *server slice* only implements methods of the service interfaces. Collective communication components correspond to that supported by MPI. All of them comprise only one replicated unit. Some of them, such as BROADCAST, SCATTER, and GATHER distinguish a root unit, the first one in the replication enumeration. Qualifier components are used to define channel communication semantics, as described above.

Computation components specify parallel computations over distributed data structures encapsulated in data components that makes part of their constitution. Their units define *state transformer procedures* over a set of local *data slices*, units of the inner data components. Data slices may be *private* or *public*. Private data slices become local variables in the unit procedure, where public ones become their parameters and return values, which are visible to procedure callers. There are three kinds of public data slices: *in*, for input data; *out*, for output data; *in/out*, for input and output data. Such modifiers are supported by SIDL, covering possible parameter passing semantics. Slices that comes from other inner computation components are called *computation slices*. Computation slices also define procedures whose parameters are their public data slices. Data slices from different unified computation slices may be fused to refer to the same data item (data sharing mentioned in Section 2.1). The protocol of the unit dictates a control flow for calling the *procedures* of computation slices (denotation of slice activation for computation slices). In HPE, *behavior expressions* are used for specifying protocols, with combinators from synchronized regular expressions, a formalism that reaches expressiveness of terminal labelled Petri nets. The *Front-End* may partially generate the code for procedures, using the signature and protocol of the computation slice to define parameters, local variables, and control flow. In Figure 7, the *root* process comprises three *data slices* (one input vector B, one input matrix A, and one output vector X), two *service slices*, for accessing a data center (DATACENTER component) where input data is retrieved and where output data is stored for further analysis, three *synchronization slices*, for data distribution across processes (GATHER and SCATTER operations for collective synchronization), and one *computation* *slice*, whose procedure computes a solution to the linear system $A \times x = B$. The *peer* process does not have a service slice for accessing the remote data center, because only *root* needs to access it. Fusion of *data slices*, represented by circles attached to the synchronization and computation slices involved, is used to set input data (A and B) and output data (x) for the LSSolve component.

5 Conclusions and Lines for Further Work

This paper sketched the architecture of frameworks that complies the # component model. The design of a #-framework for general purpose parallel programming was presented. The # component model intends to reconcile software engineering disciplines with efficient parallel programming, meeting the needs of the HPC community. Besides that, it is another attempt to adapt component technology to the demands of HPC software development. Compared to other HPC component models, the # component model is parallel by nature, targeting expressiveness of message passing parallel programming. Its main principles comes from the study of reasons that make difficult software engineering disciplines and parallel software development to be compatible with each other. The implementation of # compliant frameworks intends to make possible experimental evaluation of the hypothesis underlying the # component model principles.

The authors are currently working in the design and implementation of # compliant frameworks. Its formal semantics and analysis of the properties of the # component model are been studied under Category Theory and the Theory of Institutions. The study the use of #-frameworks as a platform for implementation of interoperable PSE's (Problem Solving Environments) is already planned. For that, the use of visual metaphors for component kinds is proposed to bring closer together programming abstractions and the needs of users of HPC.

References

1. B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl. The CCA Core Specification in a Distributed Memory SPMD Framework. *Concurrency and Computation: Practice and Experience*, 14(5):323–345, 2002.
2. R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Towards a Common Component Architecture for High-Performance Scientific Computing. In *The Eighth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society, 1999.
3. F. Baude, D. Caromel, and M. Morel. From Distributed Objects to Hierarchical Grid Components. In *International Symposium on Distributed Objects and Applications*. Springer-Verlag, 2003.
4. Bernholdt D. E., J. Nieplocha, and P. Sadayappan. Raising Level of Programming Abstraction in Scalable Programming Models. In *Workshop on Productivity and Performance in High-End Computing (in HPCA'2004)*, pages 76–84. Madrid, 2004.
5. R. Bramley, R. Armstrong, L. McInnes, and M. Sottile. High-Performance Component Software Systems. *SIAM*, 49:, 2005.

6. E. Bruneton, T. Coupaye, and J. B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *European Conference on Object Oriented Programming (ECOOP'2002)*. Springer, 2002.
7. F. H. Carvalho Junior and R. D. Lins. Haskell$_{\#}$: Parallel Programming Made Simple and Efficient. *J. of Univ. Computer Science*, 9(8):776–794, August 2003.
8. F. H. Carvalho Junior and R. D. Lins. Separation of Concerns for Improving Practice of Parallel Programming. *INFORMATION, An International Journal*, 8(5), September 2005.
9. F. H. Carvalho Junior, R. D. Lins, and R. M. F. Lima. Translating Haskell$_{\#}$ Programs into Petri Nets. *Lecture Notes in Computer Science (VECPAR'2002)*, 2565:635–649, 2002.
10. K. Chiu. *An Architecture for Concurrent, Peer-to-Peer Components*. PhD thesis, Department of Computer Science, Indiana University, 2001.
11. M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30:389–406, 2004.
12. A. Denis, C. Pérez, and T. Priol. PadicoTM: An Open Integration Framework for Communication Midleware and Runtimes. *Future Generation Computing Systems*, 19:575–585, 2004.
13. J. Dongarra. Trends in High Performance Computing. *The Computer Journal*, 47(4):399–403, 2004.
14. J. Goguen. Higher-Order Functions Considered Unnecessary for Higher-Order Programming. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 309–351. Addison-Welsey, Reading, MA, 1990.
15. J. Goguen and R. Burnstal. Institutions: Abstract Model Theory for Specification and Programming. *Journal of ACM*, 39(1):95–146, 1992.
16. S. Gorlatch. Send-Recv Considered Harmful? Myths and Truths about Parallel Programming. *ACM Trans. in Programming Languages and Systems*, (1):47–56, January 2004.
17. K. Koahey and D. Gannon. PARDIS: A Parallel Approach to CORBA. In *Proc. of the 6th IEEE Intl. Symposium on High Performance Distributed Computing (HPDC'97)*, pages 31–39. Springer, August 1997.
18. S. Kohn, G. Kumfert, J. Painter, and C. Ribbens. Divorcing Language Dependencies from a Scientific Software Library. In *10th SIAM Conference on Parallel Processing*. Springer-Verlag, March 2001.
19. K. Lau, P. V. Elizondo, and Z. Wang. Exogenous Connectors for Software Components. *Lecture Notes in Computer Science (CBSE'2005)*, 3489:90–108, 2005.
20. N. Mahmood, G. Deng, and J. C. Browne. Compositional Development of Parallel Programs. In *16th International Workshop on Languages and Compilers for Parallel Computing*, October 2003.
21. H. Milli, A. Elkharraz, and H. Mcheick. Understanding Separation of Concerns. In *Workshop on Early Aspects (in AOSD'04)*, pages 411–428, March 2004.
22. C. Pérez, T. Priol, and A. Ribes. A Parallel Corba Component Model for Numerical Code Coupling. In *Proc. of the 3rd Intl. Workshop on Grid Computing (published in LNCS 2536)*, pages 88–99. Springer, November 2002.
23. M. Shaw. Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In *International Workshop on Studies of Software Design*, Lecture Notes in Computer Science. Springer-Verlag, 1994.
24. A. J. van der Steen. Issues in Computational Frameworks. *Concurrency and Computation: Practice and Experience*, 18(2):141–150, 2005.