High-Level Service Connectors for Component-Based High Performance Computing

Francisco Heron de Carvalho-Junior
Ricardo Cordeiro Corrêa
Gisele Azevedo Araújo
Jefferson Carvalho Silva
Departamento de Computação
Universidade Federal do Ceará
Fortaleza, Brazil
{heron, correa, gisele,jefferson}@lia.ufc.br

Rafael Dueire Lins
Departamento de Eletrônica e Sistemas
Universidade Federal de Pernambuco
Recife, Brazil
rdl@ufpe.br

Abstract

Component-based programming has been applied to address the requirements of applications in high performance computing (HPC). The usual service connectors of commercial component models do not fit some requirements of HPC, mainly regarding the support of parallelism, however. This paper looks at extensions to the usual notion of service connector to meet such requirements, using the # component model as a substratum, evidencing its expressiveness.

1. Introduction

The increasing relevance of applications that demand for high performance computing (HPC) has made parallel computing a mainstream technology, attracting the attention of the software industry. Such importance has been enforced by the increasing dissemination of off-the-shelf architectures for parallel computing, such as clusters and grids, and the consolidation of multicore processors in the hardware industry. Now, the knowledge on HPC techniques for parallel and distributed programming has become the key for reaching peak performance of contemporary computer architectures. Unfortunately, parallel programming is still hard to be incorporated into widespread platforms for software development [9, 33].

In the recent years, component technology has been applied to meet the requirements of HPC applications, yielding the rise of component models and frameworks inspired by their success in the commercial arena [36], such as CCA and its compliant frameworks [6], Fractal/ProActive [10], and P-COM [27]. However, current component models have limitations to express some of the requirements of HPC applications. Indeed, they do not support neither a suitable notion of parallel component nor adequate connectors for parallel synchronization. For this reason, aiming at better support for parallelism ad hoc orthogonal extensions to HPC component models have been proposed.

This paper looks at service connectors for components-based distributed and parallel programming for HPC applications. Advanced concepts from type systems theory are applied to provide a higher-level of abstraction from the underlying computation platforms. The connectors are described in terms of the # component model, whose compliant programming systems support user-defined primitive and composite connectors, being independent from the evolution of the application requirements and technological contexts. Thus, this paper also serves both to evidence the expressive power of the # component model to describe connectors and to enumerate the set of service connectors that will be supported by a programming environment based on #-components that has been developed [13].

In Section 2, the # component model is introduced. Section 3 concerns about client/server bindings, commonly used in distributed parallel applications. Section 4 presents peer-to-peer bindings, suitable for parallel synchronization. Section 5 shows how higher-order facilities for client/server and peer-to-peer bindings can give support for skeletal programming. Section 6 introduces generic and abstract bindings, for supporting high level of abstraction in the use of bindings. Section 7 concludes the paper, outlining further developments of this work.
2. The # Component Model

The # component model moves parallel programming from a process-based perspective to a concern-oriented one that is closer to software engineering, abstracting away the computation infrastructure. A first intuitive notion of #-components can be obtained from a simple parallel program described in Figure 1. For that, its processes are sliced using concerns addressed by #-components as a slicing criterion, like in Figure 2. Certain concerns are parallel, involving slices of many processes, such as the four slices that define allocation of processes to processors, the two slices of processes 0 and 1 that perform the matrix-vector product $U = B \times Y$ in parallel, and that ones defining communication channels (send and recv pairs between processes). #-components allow to encapsulate such kind of cooperative slices, represented by units, as highlighted by dashed ellipses in Figure 2. A unit defines the role of a process with respect to the concern addressed by the #-component. One should notice that #-components can also deal with non-functional concerns, such as the allocation of processes. Figure 3(b) illustrates the main operations involved in the overlapping composition of a set of #-components $B_1, B_2, \ldots, B_n$ to form a new #-component $A$. The description in the caption of the illustration provides elements for its understanding.

A # programmer views parallel programs from the horizontal perspective of Figure 2, by concerns, while a traditional programmer views them from the vertical perspective, by processes. The latter ones tend either to encapsulate individual slices in modules, separated from their cooperating slices, or to encapsulate cooperating slices in a single module, calling the appropriate slice according to the process identification like in SPMD programming. Both alternatives break modularization principles. The mixing of processes and concerns in the same dimension of software decomposition has made hard to meet software engineering and parallel programming.

Since #-components are concrete software entities, a notion of type for #-components has been developed to work abstractly with them, where units are typed by interfaces and #-components by component types [16]. A subtyping relation has also been introduced for characterizing specialization relations. In the further sections, component types will be informally represented as diagrams as shown in Figure 3 for depicting the structure of #-components belonging to it.

The idea to move the focus of software architectures from components to exogenous connectors is not new [35, 2, 28], giving rise to algebras of connectors [23, 5]. The #-components capture a generalized notion of first-class connector, treating uniformly connectors and components as #-components, an approach that has been criticized in [28]. However, in # programming systems, connectors are conceptually separated from usual components by means of component kinds, introduced in the next section. For instance, an architectural connector, specified by a set of roles and a glue [2], may be viewed as a #-component, whose units correspond to its roles and configuration to its glue. The overlapping composition of #-component is an algebra for composing new connectors from more basic ones. Moreover, the ability to develop new primitive connectors from scratch is another important feature of # programming.

2.1 # Programming Systems

A programming system comply to the # component model if it materializes the notion of #-component. Since the nature of #-components is abstract, making possible to cover functional and non-functional concerns, # programming systems define a set of component kinds, which groups #-components of the same nature. For the #-components that belong to it, a component kind defines the semantics.
In (a), a visual notation for a #-components, represented by ellipses and their units by rectangles. From (b) to (e), some configuration scenarios. In (b), it is shown how a unit of a #-component $A$ is defined by unifying units of a set of inner components $B_1, B_2, \ldots, B_n$, which become slices of the enclosing unit. Arrows mean that units are, in fact, moved from the source to the target position. In (c), a unit of an inner #-component $B$ is lifted to be a unit of the enclosing component $A$. If all units of an inner #-component $B$ is lifted in a #-component $A$, we say that $A$ specializes $B$ and the type of $A$ is a subtype of the type of $B$. In (d), unified units of $B_1, B_2, \ldots, B_n$ are injected in a unit lifted from $C$. In (e), it is shown how, in the configuration of $A$, two inner #-components $C_1$ and $C_2$ of $B_1$ and $B_2$, respectively, are fused in $A$. For instance, if $B_1$ and $B_2$ are matrix transformations and $C_1$ and $C_2$ represent the matrices they operate, respectively, then the fusion of $C_1$ and $C_2$ means that $B_1$ and $B_2$ operate over the same data structure (data sharing, like in Fractal [10]). If $C_1$ is a specialization of $C_2$, $C_1$ supersedes $C_2$.

**Figure 3. Configuration of #-Components**

of their units, the semantics of the unification of such units, their deployment models, the rules and restrictions for combining them to #-components of other component kinds, and so on. The usual component models define only one component kind as a particular notion of software module. A #-programming system also provides a library of primitive #-components and/or facilities to build them from scratch according to component kind definitions.

The first # programming system was Haskell [14], for coarse-grained parallel functional computations on clusters of multiprocessors. Recently, an architecture for # frameworks for developing # programming systems has been proposed [13], originating the HPE framework (Hash Programming Environment), a # framework as a plugin of the Eclipse Platform. A # framework can be extended with component kinds for defining complete programming systems. The HPE programming system, implemented on top of CLI/Mono [1], was developed by instantiating the HPE framework with seven component kinds, targeting at clusters of multiprocessors: computation, data structure, synchronization, architecture, environment, application, and qualifier. There are also plans to incorporate component kinds for distributed parallel computations using Forró, a CCA framework. The service connectors proposed in this paper will be incorporated to HPE/Forró.

In the rest of this paper, it will be used the terms components environment, for the set of components deployed in a given computation platform, and components infrastructure, for those artifacts that control deployment and launching of components in the environment.

3. Client/Server Bindings

A client/server binding enables a client component to execute a service offered by a server component, involving transfer of data and control. Client/server relationships are common in distributed applications [4], but the client and the server components may also reside in the same memory space. The client/server bindings approached by HPC programming artifacts can be classified in 1-1 (sequential client, sequential server), 1-N (sequential client, parallel server), M-1 (parallel client, sequential server), and M-N (parallel client, parallel server).

Using the CCA terminology, a 1-1 binding links a uses port of the client to a provides port of the server [6]. A port carries a service description specified by a signature of methods that can be invoked by the client on the server. Invocations cause remote procedure calls only if client and server reside in disjoint memory spaces. The invocation of methods may be synchronous or asynchronous. In the former, the client waits for the server to finish the execution of
the invoked method. In the latter, the client returns from the call after invocation and continues execution, using special constructors to demand for the results of the method invocation and to synchronize with the server, such as futures of Proactive/Fractal, iterators of Forró/CCA, and channels. Such feature allows to overlap computation and communication, reducing the pause overheads of synchronization.

In Figure 4, BINDING depicts the component type for #-components that represent 1/1 bindings, with two units named uses and provides. They are configured by overlapping two #-components of component types PORTSTUB and PORTSKELETON, whose units represent, respectively, the stub client’s slice and the skeleton server’s slice that map the remote procedure call performed by the client onto a local procedure call in the server side, taking care with marshalling and serialization of parameters and results, transmission of data across the network, and synchronization semantics. In #-components that will act as the client and the server, the inner #-components of component types UPORT and PPORT must exist, respectively, such that PORTSTUB (PORTSKELETON) is a subtype of UPORT (PPORT). In a configuration where a client and a server are linked through a BINDING, such restriction makes possible to supersede the inner #-component of component type UPORT (PPORT) with its concrete version of component type PORTSTUB (PORTSKELETON), fusing them as in Figure 5. Such operation binds the uses port to the provides port. The #-components of component type UPORT and PPORT are abstract versions that only knows the signature of methods of the port, while that ones of component type PORTSTUB and PORTSKELETON are their concrete versions, enriched with the code that performs the call. Figure 6 (left) depicts the configuration of a component type PORT, supertype of both UPORT and PPORT. Each method of its interface is described by an inner #-component of component type METHOD<_, for 1 ≤ i ≤ N. The activation semantics of units of these slices corresponds to a method call.

In a # programming system, a BINDING #-component must exist for each kind of service. Moreover, several ones may exist for the same service, covering different kinds of execution platforms, intended semantics, type of some data structure to be passed as parameter, and so on. Section 6.2 shows how a type system can enable some degree of adaptability when programming with bindings. Since primitive binding connectors are programmable entities in the # programming systems, the set of supported bindings may evolve according to the needs of the users and in response to the advance of technologies. Other component models, like Fractal and CCA, normally require to be extended to support different kinds of primitive connectors.

1-1 bindings have been extended for distributed parallel applications, allowing efficient method invocations of a parallel client onto a parallel server, running in distinct parallel computers, respectively. Such scenario is typical in large-scale HPC simulations where several phenomenon models are coupled, each one running in a disjoint computing sites (clusters, supercomputers, or grids) connected to the Internet. The efficient support for distributed parallel applications has been approached by several works [25, 34, 20, 30, 37, 8]. In general, they separate the cases 1-N, M-1, and M-N, listed before. Fractal has been recently extended with collective interfaces of kinds multicast (1-N) and gathercast (M-1); There is a promise of introducing M-N interfaces in the near future [7]. The efficient approaches for collective-method-calls improve the first practices of hard-coded solutions employed by users of CORBA, with centralized control at the client and server sides for distributing and aggregating parameters and results, with a high overhead and poor scalability. A better and scalable solution is to let parallel processes at the client and server sides to synchronize directly, without intermediation.

It is possible to design primitive #-components for collective bindings by a simple extension of the previous approach of 1/1 bindings. In Figure 7, a #-component of component type UPORT (PPORT) is now composed by a set of homogenous units, each one becoming a slice of a
client (server) process participating in the collective call. The same idea is applied to #-components of component types PORTSTUB and PORTSKELETON. The processes of the collective stub and the collective skeleton must perform distributed synchronization to complete the service, as emphasized in Figure 7. No centralized control is needed.

Implementors of collective interfaces must define how a set of remote collective calls performed by $M$ client processes are mapped onto a set of local collective calls performed by $N$ server processes, where $M$ and $N$ may differ. This problem is relatively easy to resolve when there is either one client (1-N) or one server (M-1), but it is challenging in the general case. CCA community works hard to propose solutions for the “$M \times N$ problem”.

For now, each collective binding in a # programming system implements its own reduction and distribution strategies. This is flexible but poorly abstract for use in practice. Section 5 will show how to parameterize collective bindings with specific strategies. In the M-N case, strategies must be for redistribution of argument data from the $M$ client processes to the $N$ server processes and of result data from the $N$ server processes to the $M$ client processes. Such redistribution strategies may be very irregular, but only a matter of how synchronization among process slices representing stubs and skeletons is performed.

Implementors of collective interfaces must define how a set of remote collective calls performed by $M$ client processes are mapped onto a set of local collective calls performed by $N$ server processes, where $M$ and $N$ may differ. This problem is relatively easy to resolve when there is either one client (1-N) or one server (M-1), but it is challenging in the general case. CCA community works hard to propose solutions for the “$M \times N$ problem”.

For now, each collective binding in a # programming system implements its own reduction and distribution strategies. This is flexible but poorly abstract for use in practice. Section 5 will show how to parameterize collective bindings with specific strategies. In the M-N case, strategies must be for redistribution of argument data from the $M$ client processes to the $N$ server processes and of result data from the $N$ server processes to the $M$ client processes. Such redistribution strategies may be very irregular, but only a matter of how synchronization among process slices representing stubs and skeletons is performed.

Implementors of collective interfaces must define how a set of remote collective calls performed by $M$ client processes are mapped onto a set of local collective calls performed by $N$ server processes, where $M$ and $N$ may differ. This problem is relatively easy to resolve when there is either one client (1-N) or one server (M-1), but it is challenging in the general case. CCA community works hard to propose solutions for the “$M \times N$ problem”.

For now, each collective binding in a # programming system implements its own reduction and distribution strategies. This is flexible but poorly abstract for use in practice. Section 5 will show how to parameterize collective bindings with specific strategies. In the M-N case, strategies must be for redistribution of argument data from the $M$ client processes to the $N$ server processes and of result data from the $N$ server processes to the $M$ client processes. Such redistribution strategies may be very irregular, but only a matter of how synchronization among process slices representing stubs and skeletons is performed.

Implementors of collective interfaces must define how a set of remote collective calls performed by $M$ client processes are mapped onto a set of local collective calls performed by $N$ server processes, where $M$ and $N$ may differ. This problem is relatively easy to resolve when there is either one client (1-N) or one server (M-1), but it is challenging in the general case. CCA community works hard to propose solutions for the “$M \times N$ problem”.

For now, each collective binding in a # programming system implements its own reduction and distribution strategies. This is flexible but poorly abstract for use in practice. Section 5 will show how to parameterize collective bindings with specific strategies. In the M-N case, strategies must be for redistribution of argument data from the $M$ client processes to the $N$ server processes and of result data from the $N$ server processes to the $M$ client processes. Such redistribution strategies may be very irregular, but only a matter of how synchronization among process slices representing stubs and skeletons is performed.

Implementors of collective interfaces must define how a set of remote collective calls performed by $M$ client processes are mapped onto a set of local collective calls performed by $N$ server processes, where $M$ and $N$ may differ. This problem is relatively easy to resolve when there is either one client (1-N) or one server (M-1), but it is challenging in the general case. CCA community works hard to propose solutions for the “$M \times N$ problem”.

For now, each collective binding in a # programming system implements its own reduction and distribution strategies. This is flexible but poorly abstract for use in practice. Section 5 will show how to parameterize collective bindings with specific strategies. In the M-N case, strategies must be for redistribution of argument data from the $M$ client processes to the $N$ server processes and of result data from the $N$ server processes to the $M$ client processes. Such redistribution strategies may be very irregular, but only a matter of how synchronization among process slices representing stubs and skeletons is performed.

Implementors of collective interfaces must define how a set of remote collective calls performed by $M$ client processes are mapped onto a set of local collective calls performed by $N$ server processes, where $M$ and $N$ may differ. This problem is relatively easy to resolve when there is either one client (1-N) or one server (M-1), but it is challenging in the general case. CCA community works hard to propose solutions for the “$M \times N$ problem”.

For now, each collective binding in a # programming system implements its own reduction and distribution strategies. This is flexible but poorly abstract for use in practice. Section 5 will show how to parameterize collective bindings with specific strategies. In the M-N case, strategies must be for redistribution of argument data from the $M$ client processes to the $N$ server processes and of result data from the $N$ server processes to the $M$ client processes. Such redistribution strategies may be very irregular, but only a matter of how synchronization among process slices representing stubs and skeletons is performed.

Implementors of collective interfaces must define how a set of remote collective calls performed by $M$ client processes are mapped onto a set of local collective calls performed by $N$ server processes, where $M$ and $N$ may differ. This problem is relatively easy to resolve when there is either one client (1-N) or one server (M-1), but it is challenging in the general case. CCA community works hard to propose solutions for the “$M \times N$ problem”.

For now, each collective binding in a # programming system implements its own reduction and distribution strategies. This is flexible but poorly abstract for use in practice. Section 5 will show how to parameterize collective bindings with specific strategies. In the M-N case, strategies must be for redistribution of argument data from the $M$ client processes to the $N$ server processes and of result data from the $N$ server processes to the $M$ client processes. Such redistribution strategies may be very irregular, but only a matter of how synchronization among process slices representing stubs and skeletons is performed.

Implementors of collective interfaces must define how a set of remote collective calls performed by $M$ client processes are mapped onto a set of local collective calls performed by $N$ server processes, where $M$ and $N$ may differ. This problem is relatively easy to resolve when there is either one client (1-N) or one server (M-1), but it is challenging in the general case. CCA community works hard to propose solutions for the “$M \times N$ problem”.

For now, each collective binding in a # programming system implements its own reduction and distribution strategies. This is flexible but poorly abstract for use in practice. Section 5 will show how to parameterize collective bindings with specific strategies. In the M-N case, strategies must be for redistribution of argument data from the $M$ client processes to the $N$ server processes and of result data from the $N$ server processes to the $M$ client processes. Such redistribution strategies may be very irregular, but only a matter of how synchronization among process slices representing stubs and skeletons is performed.

Implementors of collective interfaces must define how a set of remote collective calls performed by $M$ client processes are mapped onto a set of local collective calls performed by $N$ server processes, where $M$ and $N$ may differ. This problem is relatively easy to resolve when there is either one client (1-N) or one server (M-1), but it is challenging in the general case. CCA community works hard to propose solutions for the “$M \times N$ problem”.

For now, each collective binding in a # programming system implements its own reduction and distribution strategies. This is flexible but poorly abstract for use in practice. Section 5 will show how to parameterize collective bindings with specific strategies. In the M-N case, strategies must be for redistribution of argument data from the $M$ client processes to the $N$ server processes and of result data from the $N$ server processes to the $M$ client processes. Such redistribution strategies may be very irregular, but only a matter of how synchronization among process slices representing stubs and skeletons is performed.
gramming [3], leading to HOC-SA (HOCs Service Architecture), one of the Globus incubator projects. The service of a high-order binding is parameterized by a computation, encapsulated in a component supplied by the client. HOCs are services that encapsulate patterns of parallel interaction in the resources of a grid. The skeletons in # programming systems are partial topological skeletons.

A generic #-component for high-order bindings is obtained only by parameterizing bindings described in the previous sections with other #-components addressing computations required by the service. For example, a 1-N client/server binding for bag-of-tasks computations could be provided, where the client passes a #-component that implements the computation performed by each server process. Another use of high-order bindings is to parameterize 1-N, M-1, and M-N collective bindings by distribution, reduction, and redistribution strategies, instead of supporting several bindings for the service varying according to the strategy. This is exemplified in Figure 9, where BINDING(t : REDISTRIBUTE, u : REDISTRIBUTE) is a component type for M-N bindings parameterized by inner #-components t and u for redistribution strategies.

6. Polymorphic Bindings

A polymorphic type system for # programming systems has been proposed, supporting generic and abstract component types with semantics in bounded quantification enriched with type operators. The three main purposes of this type system are safety, abstraction, and modularity. Safety is ensured by forbidding stuck configurations where programming mistakes that are common in parallel programming with low-level message-passing approaches, like MPI and PVM, may occur, such as: non-matching send and receive calls, conflicting messages arriving at the receiver, some kinds of deadlocks that can be avoided statically, and so on. This paper does not provide a formal specification for the type system of # programming systems. In the next two sections, it shows, informally, how generic and abstract component types can be achieve high level of abstraction and modularity in the specification of service bindings.

6.1 Generic Bindings

In the high-order M-N binding of Figure 9, a programmer could pass any #-component of subtypes of REDISTRIBUTION to parameters t and u. However, the # programming type system supports a polymorphism approach that is more expressive than subtyping polymorphism, based on universal bounded quantification [12]. Using this approach, #-components of component type

\[ \forall D : \text{REDISTRIBUTE}. \text{BINDING}(t : D, u : D) \]

become also parameterized by the type of redistribution #-component. This approach allows to fix a more specific bound for the types of allowable redistribution #-components. Generic bindings could be introduced to HOCs for introducing polymorphic skeletal services.

6.2 Abstract Bindings

The implementation of a binding for a given service may vary according to the kind of execution platform; the strategy employed for distribution, reduction, and/or redistribution in collective bindings; the intended semantics; and so on. For such cases, the # programming type system introduces partial abstract components, with semantics in existential bounded quantification. For instance, consider the generic M-N binding of Figure 9. Suppose that its implementation may vary according to the execution platform and some data type involved in the communication. The following notation is adopted to capture this scenario:

\[ \text{BINDINGOf}[\hat{P} : \text{PLATFORM}, \hat{T} : \text{DATA}] = \forall P : \hat{P}, \forall T : \hat{T}, \forall D : \text{REDISTRIBUTE}. \text{BINDING}(t : D, u : D) \]

Figure 9. A High-Order Collective Binding

Figure 10. An Abstract Collective Binding
BINDINGOp is a partial abstract component, depicted in Figure 10. In fact, \( \hat{P} \) and \( \hat{T} \) are existentially quantified type variables \(^1\), respectively bounded by component types PLATFORM, for classifying execution platforms, and DATA, for classifying types of data structures. In fact, a partial abstract component is a bounded existential type, analogous to partial abstract data types in usual programming languages, for which there is one deployed \#-component of exactly such type in the environment. The terms abstract component and partial abstract component will be used interchangeably. Examples of \#-components that inhabits BINDINGop are 

\[
\{ \text{RMI}, \text{DATA}; \text{binding}_{1} \}, \quad \{ \text{RMI}, \text{ARRAYOFDOUBLE}; \text{binding}_{2} \},
\{ \text{WEBSERVICE}, \text{ARRAYOFINTEGER}; \text{binding}_{3} \},
\{ \text{CORBA}, \text{ARRAYOFINTEGER; binding}_{4} \},
\]

\( \text{binding}_{i} \), for \( i = 1, \ldots, 4 \), are \#-components for the same service, with implementations tuned for a certain execution platform and data structure. In usual programming languages that support ADTs, like Ada, programmers are responsible to pack and unpack modules as ADTs. In \# programming, programmers are still responsible to pack primitive \#-components as inhabitants of abstract components, explicitly, but they let to the components infrastructure the task of unpacking at launch time. Programmers build composite \#-components by overlapping abstract components viewed as parameterized abstract components \([16]\), which are type operators that, when applied to other abstract components supplied as parameters, return the \#-component of the abstract component. This is the justification for the “parameterized-like” notation for abstract components. By supplying parameters \( \hat{E} \) and \( \hat{T} \) of BINDINGop, a programmer may control the choice of the system for an appropriate binding of the \#-component. For example, if a configuration claims for an inhabitant of BINDINGop by supplying it as BINDINGop[RMI, ARRAYOFDOUBLE], then the system returns, at launch time, the \#-component \( C \), such that 

\[
\{ \text{RMI}, \text{ARRAYOFDOUBLE}; \text{binding}_{4} \}
\]

exists in the environment. For instance, amongst the \#-components listed before, \( \text{binding}_{4} \) is chosen for \( C \). If BINDINGop[RMI, ARRAYOFDOUBLE] is not deployed in the environment, the system will search for a more generic \#-component than \( \text{binding}_{4} \). Since RMI and ARRAYOFDOUBLE are bounds for universally quantified type variables, by subtyping rules it can be found by looking for an inhabitant of BINDINGop[RMI, DATA], assuming that DATA is the lowest supertype of ARRAYOFDOUBLE. In such case, the \#-component \( \text{binding}_{4} \), could be chosen, which is a more generic implementation, probably less efficient because it is suitable for any data structure. Furthermore, if there is no \#-component \( \{ \text{RMI}, \text{DATA}; \; C \} \), the system will continue to generalize BINDINGop until to find a suitable \#-component. Such approach is possible if the system keeps track of the subtyping hierarchy of abstract components deployed in the environment. In order to avoid ambiguities, only one \#-component can inhabit an abstract component in a given environment, but \# programming systems could easily introduce some versioning mechanism for managing \#-components. If no generalization of BINDINGop[RMI, ARRAYOFDOUBLE] is found in the environment at the deployment time of a configuration that refers to it, the user is warned. If such condition still happens at launch time, an unrecoverable error exception occurs. The approach of searching and combining components at launch time, from specifications given by programmers at configuration time, has been used by P-COM \([27]\).

After the system unpacks \( \{ \text{RMI}, \text{ARRAYOFDOUBLE}; \text{binding}_{2} \} \) for BINDINGop[RMI, ARRAYOFDOUBLE], \( \text{binding}_{2} \) must be applied to the component type of the intended redistribution strategy and to two \#-components of such component type. Since it is not desirable programmer intervention at launch time, it is necessary to include configuration facilities that allow programmers to set these parameters at configuration time. In fact, this is possible because the kind of the type of \( \text{binding} \), is known at configuration time, but it is possible to resolve this problem by using parameterized abstract components:

\[
\text{BINDINGop}[ \hat{P}, \hat{C}; \text{PLATFORM}, \hat{T}, \text{DATA}; \hat{D}, \text{REDISTRIBUTE}] =
\{ \forall \hat{P} < \hat{P}; \forall \hat{T} < \hat{T}; \forall \hat{D} < \hat{D}; \text{BINDING}(t, u, v, w) \}
\]

The configuration continues similarly to the scenario depicted in Figure 10. If a programmer uses BINDINGop[RMI, ARRAYOFDOUBLE, TRANSPOSE] in a configuration, where TRANSPOSE is an abstract component for \#-components that implements a particular redistribution strategy, the system will look, in the client’s environment, for inhabitants of subtypes of TRANSPOSE for each reference to \( D \) in the configuration body. For a given choice of \( E, T, \) and \( D \), and by forcing only abstract components with all parameters supplied in the configuration body of BINDINGop[E, T, D], the component infrastructure can choose the actual inner \#-components to be overlapped to form a \#-component for BINDING[E, T, D] by looking at its corresponding abstract components. One should remember that the inner abstract component of BINDING[E, T, D] are \( E, T, \) PORTSTUB[D], and PORTSKEL[D] (Figure 10). Such approach frees programmers of making assumptions about parameters of higher-order \#-components whenever building configurations, letting such responsibility to the component infrastructure without compromising the explicit control of programmers. In fact, the component infrastructure sees any composite \#-component as a high-order one, parameterizing it by its inner \#-components by default. Thus, to build a composite \#-component, the infrastructure only supplies its parameters (inner components) by providing the unpacked #-
component for its abstract component type. Thus, the composite #-component can be viewed as a combinator whose definition depends on its component kind.

Abstract Components enable a general notion of skeleton service, as proposed originally by Cole, whose implementation varies according to the underlying computation platform. It is more general than HOCs, because a service may have several implementations in the same server, each one more appropriate for certain combinations of parameters.

7. Conclusions and Lines for Further Works

The support for a suitable set of connectors is a key issue for the success of a component model for developing high performance computing (HPC) applications targeted at high-end distributed and parallel architectures. Service connectors are a widespread artifact to plug components in successful component models of commercial applications, but they must be extended to address some peculiar requirements of HPC applications, mainly regarding the expression of parallel patterns of synchronization amongst components. For such purpose, this paper provides a set of extensions to the usual service connectors that have been approached in the design of component models for HPC, showing how advanced concepts from type systems theory could be used to lift the level of abstraction in their use in programming. The expressiveness of the # component model, whose compliant programming systems support user-defined primitive and composite connectors, is evidenced by mapping the proposed service connectors in terms of #-components. Indeed, such connectors will be introduced in HPE/Forró, a # programming system that is a CCA framework for distributed parallel applications.

8. Acknowledgments

CNPq - Conselho Nacional de Pesquisas e Desenvolvimento Tecnológico, Brazilian Government, sponsors this project (grant 475826/2006-0).

References