

A Hybrid Decomposition Scheme for Building Scientific Workflows

Wei Lu¹, Kenneth Chiu², Satoshi Shirasuna¹, Dennis Gannon¹

1. Computer Science Department, Indiana University

2. Department of Computer Science, State University of New York, Binghamton

Abstract

Two problem decomposition schemes, component assembly and workflow orchestration, have been widely adopted to architect large scale scientific applications. These two methodologies, however, approach problem decomposition from different distinctly perspectives with the result that most problem solving environments provide only one approach to the exclusion of the other. For example the Ccaffeine, a parallel CCA framework, provides the component assembly environment only; while Kepler, a widely-used scientific workflow toolkit, is designed mainly as the workflow orchestration environment. Each methodology has situations, within the same problem domain, where it may be more appropriate than the other, however. Thus, to bring benefits from both methodologies, in this paper we present a hybrid problem decomposition scheme. By augmenting Ccaffeine with the web services interface, we enable Ccaffeine as a special workflow actor in the Kepler environment, so that a Kepler user can gain the benefits of both approaches by applying the two methodologies for the subproblems depending on the various performance and resource-sharing requirements. The hybrid scheme will first use the workflow scheme to decompose the problem based on the distribution of the resource; thereafter adopts the component assembly scheme to further decompose those computationally intensive cores for the high performance solutions.

1 Introduction

Managing the complexity and scale of scientific computations requires decomposing systems into manageable and independent units. These systems can be decomposed along a number of different dimensions, however. Along one dimension is the instantiation of the software units that cooperate to perform the computation. Decomposing along this dimension results in the component assembly paradigm, whereby units of software functionality are encapsulated into independent components with well-defined interfaces. The problem is thus decomposed based on functionalities, and the solution can be built by composing the components together in a “provide-and-use” fashion. The Com-

mon Component Architecture (CCA) [3] is such a component architecture specification supporting component assembly for high performance scientific applications.

Alternatively, decomposing along the temporal dimension results in the workflow assembly scheme, in which the problem is resolved according to the temporal ordering between the processing steps. Then the software solution is formed by composing the components using a “pipe” fashion, namely connecting the input with the matched output of components. Kepler [2] is a popular workflow toolkit for the scientific applications.

Building a large-scale scientific application, by adopting only one of the schemes usually is insufficient. The two schemes describe the problem decomposition from different perspectives with different concerns, thus each have their own advantages and disadvantages. For example we may need to stage-in the data from a Grid data server before running a parallel CCA job, and stage-out the result after it runs. A workflow system can help the automating of the data staging processing. Hence, a hybrid solution which can integrate the two problem decomposition schemes to form a unified problem solving environment will be beneficial for most scientific application.

In this paper, we present our prototype solution which integrates the CCA framework into the Kepler workflow system by using a set of web services to bridge between the two systems. The CCA framework we are using is Ccaffeine [1], which is a high performance parallel infrastructure for the composition of CCA components.

2 Hybrid Decomposition Scheme

In this section we first summarize a number of current technologies which have been widely adopted by many scientific applications.

2.1 Software Components and CCA

A software component in this context is an independently deployable unit of software, which interacts with the rest of the world only through its well-defined interface, while its internal implementation remain opaque from the outside. The complexity of individual components is thus encapsulated. Based on these interfaces the components can

be composed into complex applications. This situation implies programming language independence, and the ability to construct complex systems via “plug-and-play”.

The Common Component Architecture (CCA) specification [3] is designed to bring the benefit of components to scientific computing software. The essence of CCA is the so-called *provides/uses* design pattern. A component offers its functions to the outside through the *provides port*, while it gets the needed functions from the outside via the *uses port*. Thus the components are decoupled by the ports and can be composed into an application by wiring the matched provides port and uses port together. The port interface of a CCA component is described in SIDL [13], which is partially based on the CORBA IDL, with extensions for needs commonly found in scientific computing, such as multi-dimensional arrays. By automatically generating glue code from SIDL for multiple languages, the Babel language interoperability tool allow components written in various languages to interoperate through their interface.

The Ccaffeine framework [1], is the mainstream CCA framework implementation designed for the parallel scientific computing. In Ccaffeine each processor runs the components in parallel, with the components wired in the exactly same way in each process. Analogously to the single-program-multiple-data (SPMD) model, this component oriented parallel paradigm is also referred as single-component-multiple-data (SCMD). The components in the same process interact through their CCA ports, while interprocessor communication among parallel instances of a component may use any available HPC communication layers (e.g., MPI, PVM).

Ccaffeine provides a drag-drop GUI front-end (Figure 2a) as well as a set of commands to manipulate the components. In a parallel environment, as shown in the Figure 1a, the Ccaffeine framework is running on each participating processor, while a central multiplexer (call the *muxer*) between the front-end and the back-ground frameworks multiplexes the commands from the front-end to the background frameworks which are going to execute the commands in parallel.

2.2 Scientific Workflow

Scientific workflow systems aim at a unified problem-solving environment that combine scientific data management, analysis, simulation, and visualization tasks in a workflow fashion. Compared with general workflows, scientific workflows tend to be much more dataflow-oriented [5].

Kepler [2] is an open-source scientific workflow system that builds upon the dataflow-oriented PTOLEMY II system [7]. In Kepler, users develop workflows by selecting appropriate components called *actors* and wire the actors together to form the desired workflow graph, as shown in Figure 2b. Each actor usually fulfills a specific functionality, and accepts the incoming dataflow from its *input ports* and sends the outgoing dataflow through the *output ports*.

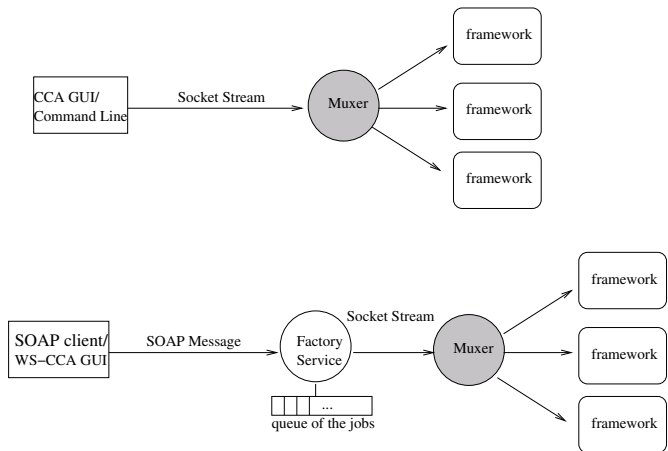


Figure 1. Figure (a) above, illustrates the parallel setting of Ccaffeine with the muxer. Figure (b) illustrates the structure of the *JobFactory* and the WS-CCA GUI in the parallel setting with the queue and multiple muxers.

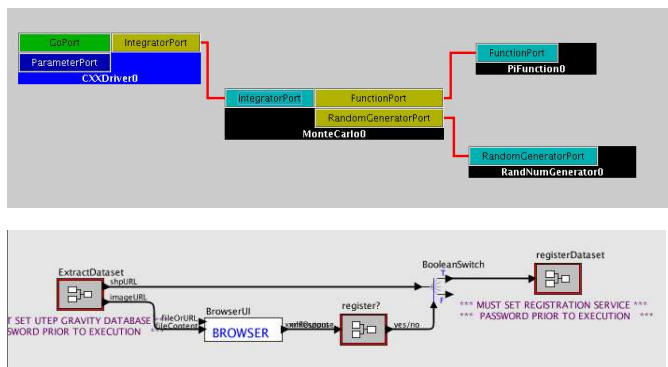


Figure 2. Figure (a) above shows the composition GUI of Ccaffeine. Figure (b) shows the workflow GUI of Kepler.

The composition can be formed by piping the input ports of an actor to the type-matched output ports provided by another actor.

Currently Kepler provides a number of domain-oriented scientific application packages (e.g., biology and geology packages) in the form of actors, together with a set of utility actors for the web service and the grid services (e.g., GridFTP).

2.3 The Hybrid Approach

Although the terms (i.e., component, actor, port, and connection) used by CCA and Kepler are very similar, their semantics are subtly different:

- A CCA component conceptually is a class instance

which can have multiple public methods, while one Kepler actor stands for one method for a dedicated task.

- Both of them use a *port* to represent the interaction interface. However a CCA port essentially is an interface set containing multiple method signatures, while the Kepler port stands for the structural type of the incoming or outgoing data-flow channels.
- The connection of a provides/uses port pair between two CCA components indicates one component will invoke the other in order to complete its execution. From this sense the connection is more like the function dependency in the general program language. In contrast, the connection between two actors in a workflow usually implies the temporal order relationship of their processing. That means that until the execution of one actor is finished, the data as well as the control will be transferred to another one via the ports. Thus here the connection represents the data and control flow. As a result, usually the component assembly is also referred as *spatial composition*, while the workflow assembly as *temporal composition* [10, 15].

Those differences root in the distinct perspectives of the two decomposition schemes. The composition by the component assembly focuses on the “*what to compose*” rather than the “*how to compose*”, which is encapsulated in the component implementation. Conversely, the workflow assembly is mainly concerned about “*how to compose*”, which is described in the workflow language or visual notations. The workflow actor itself is merely aware of its input/output flows without any idea of how to interact with other actors.

The different perspectives leads the merit of the two schemes. For example, due to their inherent complexity, it is much harder to describe scientific algorithms, especially those parallel ones, in the workflow language or visual notation, than to use the general programming languages. Even if we were able to describe it as a workflow, the fairly intricate network of the workflow would be hard to understand, design, and maintain. Another concern working against the use of workflow assembly for HPC is the performance issue. Most scientific workflow systems usually have a centralized workflow engine which manages the running of the actors by following the data/control flow. The indirection between the actors cause the extra communication, and thus significant overhead. In contrast, in the component assembly, particularly the CCA, the relationship between the components is the simple caller-callee and the communication between the components is direct and could even be in the same address space as Ccaffeine has. However since the component needs to take care of all its interactions with other components by itself, the system tends to be tightly-coupled, whereas with the workflow, it is easier to deploy and maintain in a loose-coupled distributed system, due to the simpler running context, namely only input and output, of each actor.

Based on the above observation, we posit that to build a large-scale scientific application, a hybrid approach is more promising since it can bring the benefit from both of the two schemes. Typical scientific applications involve multiple data processing phases, which often are distributed with the available computation or data resource. Among those phases there are number of computationally intensive cores, which are often the classical numerical algorithm and need the high performance execution environment. The hybrid scheme will first use the workflow scheme to decompose the problem based on the distribution of the resource; thereafter once those computationally intensive sub-problems have been identified the component assembly scheme can be adopted to further decompose them to lead the high performance solutions.

3 Building Web Services for Ccaffeine

The most efficient way to enable the hybrid scheme is to build a web services interface for the Ccaffeine because:

- Kepler has already provided a handy web services actor to support general web services;
- With the web services interface, Ccaffeine itself instantly shares all the benefit of the web services, such as the interoperability as well as the rich set of web/grid service resources.

The difference between a service and a component is debatable, it, however, is the conventional wisdom that the interface granularity of a service should be more coarse than the one of the component. Like the Facade [9], the web service with a coarse-grained interface serves as the external view of the system and hides the internal complexity and detail, while internally the functionality of the system can still be implement by the software component.

Another design issue raised from the workflow semantic is that the web service interface of the Ccaffeine frameworks should be task-specific rather than frameworks oriented. That means merely exposing the Ccaffeine *BuildService* [4] port, which provides an interface for component manipulation, as a web service makes little sense for a scientific workflow.

3.1 Architecture

Based on the design, we first introduce the *Job* concept, which is a specific task performed by a group of wired Ccaffeine components. For example, the calculation of the value of Pi could be considered a job. Each job should have a specific and meaningful interface, as well as its own facade service, called the *JobProxy* service, which provides the corresponding web service interface.

As the service is not instantiable, a factory service is needed to maintain the life cycle of the *JobProxy* service, namely the creation, modification, and destruction. Also

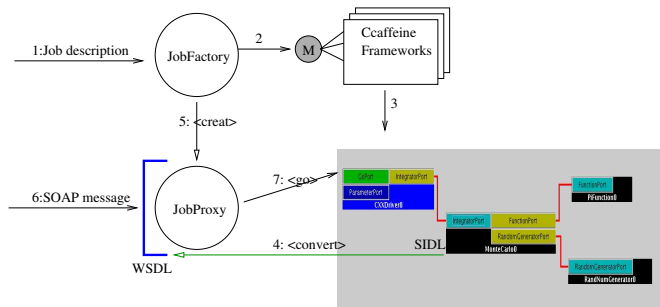


Figure 3. The procedure of invoking *JobFactory* service to create the *JobProxy*: (1) user sends the job creation request to the *JobFactory*; (2) *JobFactory* deploys the job by forwarding the composition scripts to muxer; (3) *Caffeine* framework executes the scripts parallel; (4) the gateway-port SIDL is converted to the equivalent WSDL; (5) *JobFactory* creates the *JobProxy* service; (6) the user invokes the *JobProxy* service through the WSDL; and (7) the *JobProxy* run the components by calling “go” on *Driver*.

the factory will be general to create any *JobProxy* service dynamically based on the user’s request. We call the factory service the *JobFactory* service, which essentially is the facade of the background *Caffeine* frameworks, and hides the complexity of the *Caffeine* framework, components and jobs from the user.

As shown in the Figure 3, the usage from the users’ perspective contains two explicit phases: creating the *JobProxy* service followed by the invocation to it, For convenience this set of services is called *CCA-Services*.

3.2 *JobFactory* Service

The *JobFactory* service basically can be viewed as a web service front-end for the *Caffeine* frameworks. As shown in the Figure 1b, it builds a socket connection with a *Caffeine* muxer which is connecting parallel *Caffeine* frameworks. Whenever the *JobFactory* receives the SOAP request messages from the user, it parses the command scripts from the SOAP message, and forward to the muxer, which in turn multiplex the command to the parallel *Caffeine* frameworks. The command scripts is going to be executed in parallel by the *Caffeine* frameworks, and the result will be collected and sent back to the user in a SOAP response message.

However the *JobFactory* doesn’t merely do the I/O redirection to the framework as the work in [11]. Since *Caffeine* is not aware of the concept of job, it is the responsibility of *JobFactory* to create, deploy, maintain and destroy the job and related service. The extra job management layer

certainly enables more flexibilities, such as the scheduling of the job on the multiple frameworks.

As a proof-of-concept work, the interface of the *JobFactory* service is designed simply to provide

- `create()` : create the job and its proxy service,
- `modify()` : modify the composition for an existing job,
- `destroy()` : destroy the job and stop its related proxy service.

To create the job and its proxy service, the user is required to provide at least two parameters to the *JobFactory* service, *gateway-port* and *composition script*. The *gateway-port* refers to a meaningful port type which will be to exposed to outside. *CCA* defines a special port, called *GoPort*, whose *go()* method is regarded as the start entry point to trigger the running of wired components as the *main()* function in C. The component implementing the *GoPort* usually is called *Driver*. However just as the *main()* function the *go()* method of *GoPort* doesn’t have a specific method signature and its return value just indicate is the success or failure of execution. Hence to create the job-specific *JobProxy* service we have to explicitly specify a gateway port interface, which provides the meaningful, task-specific interface and will be connected by the *Driver* as the use port. Since *CCA* uses the *SIDL* to describe the port interface, the *JobFactory* service will search its local *SIDL* definition repository for the given *gateway-port*, and will try the best to convert the *SIDL* definition to equivalent *WSDL* definition. The another parameters, *composition script*, describes how the components are assembled through their *provide/use* ports to perform the job. For the sake of simplicity, we use the *Caffeine* scripts directly. It would be more web service friendly to use XML schema to describe the composition scripts in XML syntax.

Once the *WSDL* conversion and *Caffeine* commands running succeed, the *JobFactory* generates a unique ID for the job, launches the specific *JobProxy* service, and returns a URL of the generated *WSDL* file as well as the job ID to the user. Those properties of the job (i.e., job ID, *WSDL* file and composition structure) will be book-kept in the local repository for the job management. For instance, user can modify the composition of the job by providing the job ID and new composition to the *JobFactory*. As long as the modification will not affect the *gateway-port* interface, the internal composition structure of the *JobProxy* service can be changed transparently at run-time by the power of “plug-and-play” of the component assembly.

3.3 *JobProxy* service

The *JobProxy* service receives and validates the incoming SOAP message against the specific *WSDL*, and then extracts the typed argument values from message and feeds them to the *Caffeine* frameworks via the muxer.

Each job has an associated Driver component, from which the execution of the components begins. Since GoPort doesn't have meaningful parameters, in order to pass the initial arguments and get back the return value from the Driver we utilize the port property, which is a map of key and value and can be treated as the context of the execution of the Driver. For every input argument the JobProxy set the corresponding port-property of the GoPort of the Driver, which gets those arguments from the port properties before its running. Once the execution is done Driver passes the result back to JobProxy by setting a special port-property. Then JobProxy packed the result within a SOAP response message back to the user.

3.4 From SIDL to WSDL

Web Service Definition Language (WSDL)[8] is the standard interface description language for the web service, and internally it uses the XML Schema[18] as its type system. Both SIDL and WSDL are belong to the interface description languages. They have the similar set of the primitive data types. SIDL also defined the generic array type, which can be simply described by the sequence construct of the XML Schema in WSDL. Both of them have similar structure to describe the interface: the SIDL interface construct contains multiple methods, while the WSDL portType¹ includes multiple operations. Therefore it would be straightforward to map a SIDL interface definition to the WSDL portType definition by following the rules:

- Each SIDL interface is mapped to the WSDL portType
- Each method in the SIDL interface is mapped to the operation in the WSDL portType
- For each method, its input parameters will be grouped and to be mapped to an invocation message; its result value will be mapped to a response message; and a fault message will be generated for potential exceptions.

However, from the perspective of the object oriented model, SIDL is fundamentally distinct from the WSDL. The WSDL, inherently designed for message passing paradigm and the distributed system, focuses the message structure on the wire which server/client agree upon. Any data type in the WSDL has to be defined in the XML Schema so that it can be referred by the WSDL interface to describe the structure of the exchanged XML message. A WSDL interface itself basically is not a data type, instead it would better be viewed as a group of message exchanges in which a Web service is prepared to participate [17]. Consequently a WSDL interface can't be referred as a data type, neither as a operations parameter type, nor as an element type in the XML schema. Also the WSDL interfaces don't have

¹the portType is renamed as interface in WSDL 2.0

the inheritance relationship² Conversely, just like most object oriented language SIDL is about how an object is accessed through its interface. A SIDL interface basically is a set of method signatures, and it can be referred within any other SIDL type with the polymorphism. That also means there is no way for us to know the structural information of an interface definition since it can have multiple different implementations. Moreover to be purely object-oriented the current SIDL specification doesn't support the structure construct, which is supported by the CORBA IDL.

Thus, to summarize, the key challenge to the conversion becomes how to map the reference of a SIDL interface to the counterpart in the WSDL. We could solve this problem by creating a web service for each referred interface and using the URL of the created web service as the reference to the interface. However having a high performance algorithm invoke an interface via the web service during the execution will generate unacceptable performance penalty. Since only the gateway-port interface needed to be converted, a more practical solution is prohibiting the SIDL interface type from being the method parameter type in the gateway-port interface. In other word our current restriction is that only the primitive data types are allowed in the gateway-port interface. To supported more complicated data types, we suggest that SIDL introduce the structure construct as IDL did at least for those distributed use cases³. The mapping of the structure constructs to the XML Schema definition has been proposed in the specification of CORBA WSDL/SOAP Interworking [16]. Thus with the support of the structure construct, the conversion from SIDL to WSDL will be quite straightforward by extending the CORBA WSDL/SOAP Interworking specification.

4 Orchestrating CCA-Service

The web service interface enables the orchestration of the CCA services with any other web/grid services seamlessly by the web service enabled workflow tool (e.g., Kepler) or the web service workflow protocols, like WS-BPEL [12].

Kepler provides a general web service actor which can be configured dynamically to be a specific actor for a given WSDL file [14]. Then it is straightforward to introduce the JobProxy service into the Kepler environment due to its web service interface. It, however, still needs the user to create the JobProxy service by calling the JobFactory service first. The separated procedures will be fairly inconvenient for the users due to the different toolkits to use and the unnecessary detail, such as job ID, the URL of the JobProxy service, etc. It will be desired to have a unified development environment for both component assembly and work-

²wsdl 2.0 supports the extension, but it is not a real inheritance, since no polymorphism is allowed

³To our best knowledge, the introduction of structure into SIDL is being planned.

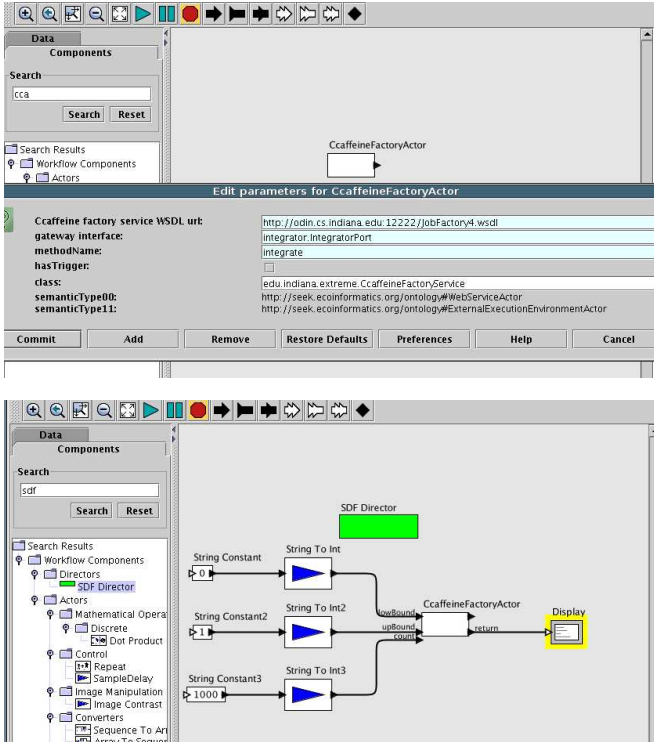


Figure 4. a (above): CCA-Service actor and its parameters; b: Building Workflow over the CCA-Service actors in Kepler

flow composition within the Kepler, and hide any unnecessary procedures and information from the users. With that motivation, we create a special Kepler actor for the Ccaffeine service, CCA-Service actor. The CCA-Service actor has four configuration parameters (Figure 4a): the URL of the WSDL of the JobFactory service, the composition script, the gateway-port type, and the name of the delegated method. Once the user completes the configuration, the CCA-Service actor sends SOAP request to the JobFactory service at the specified URL, which in turn creates the job and JobProxy service based on the gateway-port type and composition script and return the URL of the JobProxy WSDL back to the CCA-Service actor. With the JobProxy WSDL, then the CCA-Service actor simply acts as the general web service actor does, namely dynamical creating the ports of the actor based the interface/operation/type information defined in the WSDL. At this moment the general CCA-Service actor has been transformed to a job-specific actor which can perform the job in the remote Ccaffeine frameworks. And all the above steps are transparent to the user.

Also writing the lengthy Ccaffeine composition script is difficult and error prone for Kepler users, particularly when lacking of the framework context information. To facilitate the writing/editing of the composition script, we modify

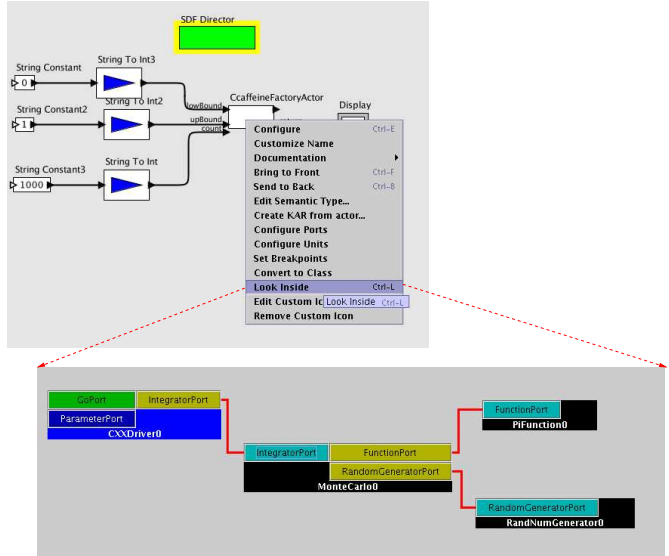


Figure 5. Thy Hybrid Visual Composition GUIs provided by CCA-Service actor and Kepler

the Ccaffeine GUI from the original socket-stream based to be the SOAP-message based. Thus every gesture done by the GUI user will be first buffered locally, and eventually sent to the Ccaffeine framework as a SOAP message in a batch-mode. The message based composition GUI is embedded within the CCA-Service actor as its configuration window, also called *tableau* in the term of PTOLEMY [6]. As shown in the Figure 5, the CCA GUI become visible by right-clicking on the CCA-Service actor and selecting “Look Inside” from the resulting pop-up menu. Additionally, benefiting from the component system the user still can change the internal component assembly of the CCA-Service actor after creation as long as the gateway-port type keeps same. And the GUI definitely makes the modification much easier.

Aid by the CCA-Service actor and the embedded CCA GUI, now the user is able to visually compose the system from the perspectives of both temporal composition and spatial composition in the Kepler environment. Different from the work of [15], in which temporal and spatial compositions are treated as two different views of one same system but at same level, our solution provides a hierarchical style to build the large scale scientific workflow. That is workflow assembly is the basic decomposition scheme, based on which the component assembly will be adopted for the performance sensitive actors.

5 Conclusion

In this paper, we introduce two common problem decomposition schemes for scientific applications: component as-

sembly and workflow assembly. They each have their own merits, arising from their own distinct perspectives to decomposition. In this paper, we provide a novel hybrid solution, which uses Kepler workflow to arrange the temporal ordering of computations, but CCA to organize the functional units of the parallel computation. The hybridization hinges on the web services interface which we have provided to Ccaffeine. The factory pattern enables the dynamic creation of the job specific web service for the Ccaffeine and the CCA-Service actor, which hides the underlying web service interactions, enables the use of CCA components in Kepler transparently. One challenge is the conversion of SIDL to WSDL, since they are inherently designed around different models. For a practical solution, we suggest that SIDL introduce the structure construct.

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] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludscher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows. In *16th Intl. Conf. on Scientific and Statistical Database Management (SSDBM'04)*, 21-23 June 2004 2004.
- [3] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. R. Kohn, L. McInnes, S. R. Parker, and B. A. Smolinski. Toward a common component architecture for high-performance scientific computing. In *HPDC*, 1999.
- [4] D. E. Bernholdt, R. C. Armstrong, and B. A. Allan. Managing complexity in modern high end scientific computing through component-based software engineering. In *Proc. of HPCA Workshop on Productivity and Performance in High-End Computing (P-PHEC 2004)*, Madrid, Spain, 2004.
- [5] S. Bowers and B. Ludaescher. Actor-oriented design of scientific workflows. In *International Conference on Conceptual Modeling (ER'2005)*, 2005.
- [6] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, and H. Z. Y. Zhao. Heterogeneous concurrent modeling and design in java, volume 1: Introduction to ptolemy ii. Technical report, EECS, University of California, Berkeley, 2004.
- [7] U. B. Department of EECS. Ptolemy ii project and system. <http://ptolemy.eecs.berkeley.edu/ptolemyII/>, 2004.
- [8] E. Christensen et. al. Web Services Description Language (WSDL) 1.1. <http://www.w3.org/TR/wsdl>, 2001.
- [9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- [10] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, and R. Bramley. Merging the cca component model with the ogis framework. In *Proceedings of CC-Grid2003, 3rd International Symposium on Cluster Computing and the Grid, Tokyo, Japan*, pages 182–189, May 12–15 2003.
- [11] V. P. Holmes, W. R. Johnson, and D. J. Miller. Integrating web service and grid enabling technologies to provide desktop access to high-performance cluster-based components for large-scale data services. In *Annual Simulation Symposium*, number 167-174, 2003.
- [12] IBM. Business Process Execution Language for Web Services version 1.1. <http://www-128.ibm.com/developerworks/library/specification/ws-bpel/>, 2005.
- [13] G. Kumfert, T. Dahlgren, T. Epperly, and J. Leek. Babel 1.0 release criteria: A working document. <http://www.llnl.gov/CASC/components/docs/BabelReleaseCriteria.pdf>, December 2003.
- [14] B. Ludscher. Toward actor-oriented web service-based scientific workflows. Technical report, San Deigo Supercomputer Center, 2004.
- [15] A. Mayer, S. McGough, N. Furmento, W. Lee, S. Newhouse, and J. Darlington. Icen dataflow and workflow: Composition and scheduling in space and time. In *UK e-Science All Hands Meeting*, Nottingham, UK, 2003.
- [16] OMG. Corba to wsdl/soap interworking specification. <http://www.omg.org/docs/formal/03-11-02.pdf>, 2003.
- [17] J. W. S. Parastatidis. Why wsdl is not yet another object idl. *SOA Web Services*, 2004.
- [18] W3C. Xml schema. <http://www.w3.org/XML/Schema>, 2004.