A Task Abstraction and Mapping Approach to the Shimming Problem in Scientific Workflows

Cui Lin, Shiyong Lu, Xubo Fei, Darshan Pai and Jing Hua
Department of Computer Science, Wayne State University
{cuilin, shiyong, xubo, darshan, jinghua} @wayne.edu

Abstract

Recently, there has been an increasing need in scientific workflows to solve the shimming problem, the use of a special kind of adaptors, called shims, to link related but incompatible workflow tasks. However, existing techniques produce scientific workflows that are cluttered with many visible shims, which distract a scientist's focus on functional components. Moreover, these techniques do not address a new type of shimming problem that occurs due to the incompatibility between the ports of a task and the inputs/outputs of its internal task component. To address these issues, 1) we propose a task template model which encapsulates the composition and mapping of shims and functional task component within a task interface; 2) we design an XML-based task specification language, called TSL, to realize the proposed task template model; 3) we propose a service-oriented architecture for task management to enable the distributed execution of shims and functional components; and 4) we implement the proposed model, language and architecture and present a case study to validate them. Our technique uniquely addresses both types of shimming problems. To our best knowledge, this is the first shimming technique that makes shims invisible at the workflow level, resulting in scientific workflows that are more elegant and readable.

1 Introduction

Scientific workflows have recently emerged as a new paradigm for scientists to formalize and structure complex and distributed scientific processes to enable and accelerate scientific discoveries [4, 3]. During workflow design, third-party autonomous services and applications are frequently used. Very often, these services and applications are syntactically mismatching or semantically incompatible, necessitating the use of a special kind of workflow components, called *shims*, to mediate them. A shim takes the output data of an upstream workflow task, performs some transformation, and then feeds the data to the input of a downstream task. The shimming problem has been widely recognized as an important problem in the community [1, 10], leading to much efforts in the development of shims [6], shim-aware workflow composition [1] and the suggestion of a new discipline called *shimology* [10].

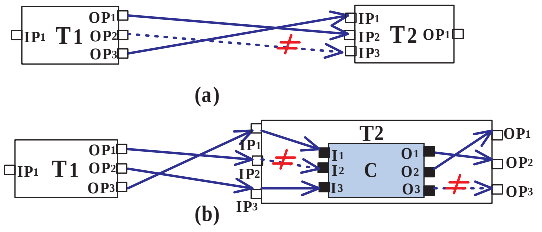


Figure 1. (a) The TYPE-I shimming problem; (b) The TYPE-II shimming problem. ($\neq$: mismatch)

We refer to the above shimming problem as *TYPE-I shimming problem*, which occurs at *the workflow level* due to the incompatibility of output ports of an upstream task with the input ports of a downstream task. For example, in Figure 1.(a), when the output port OP_2 of upstream task T_1 is incompatible with the input port IP_3 of downstream task T_2, a shim is needed to mediate them. While still not recognized by the community, we identify a second type of shimming problem, called *TYPE-II shimming problem* that occurs at the *task level* when tasks are created from third-party heterogeneous services and applications (called *task components*) and there is incompatibility between task ports and inputs/outputs of task components. For example, in Figure 1.(b), although $T_1.OP_i$ $(i = 1, 2, 3)$ and $T_2.IP_i$ $(j = 2, 3, 1)$ are compatible, inside T_2, input port IP_2 is incompatible with input I_2 of task component C and output O_3 of C is incompatible with output port OP_3 of task T_2.

Existing shimming techniques have two serious limitations. First, they produce scientific workflows that are cluttered with many visible shims. For example, a recent study of the 560 scientific workflows available from myExperiment (www.myexperiment.org) shows that over 30% of workflow tasks are shims. Ideally, these shims should be hidden from scientists so that they can better focus on functional components of workflows. Second, these techniques do not address TYPE-II shimming problem and thus require a user to write custom wrapper shim code around a task component according to the task programming model of a system. Moreover, these hard-coded implicit shims are *irreusable* across other tasks. Addressing TYPE-II shimming problem is more challenging due to the heterogeneity of task components and the needed flexible mapping between task ports and inputs/outputs of task components (see Figure 1.(b) for illustration).

To address these issues, 1) we propose a task template model which encapsulates the composition and mapping of shims and functional task components within a task interface, 2) we design an XML-based task specification language, called *TSL*, to realize the proposed task template model; 3) we propose a service-oriented architecture to enable the distributed execution of shims and functional components; and 4) we implement the proposed model, language and architecture and present a case study to validate them. Our technique uniquely addresses both types of shimming problems. To our best knowledge, this is the first shimming technique that makes shims invisible at the workflow level, resulting in scientific workflows that are more elegant and readable. We summarize the advantages of our approach in Section 3.3.

2 Task Template Model

Tasks are the basic building blocks of a scientific workflow. A *task model* provides the modeling primitives to model design-time and run-time behaviors of workflow tasks. As shown in Figure 2, the design-time behavior of a task is modeled in a *task template model* and specified in a *task specification language* (TSL) as a *task template specification* (TTS), which defines the interface of a task and its implementation details. A set of task templates in a system constitute a *task library*, from which one can instantiate *task instances* for the creation of a scientific workflow. During run-time, the execution status including run-time state and behavior of each task instance is maintained by a *task run*, which is modeled according to a *task run model* and described in a *task run description language* as a *task run descriptor*.

In this section, we propose a task template model and its task specification language, TSL, for the specification of dataflow-based task templates, enabling the abstraction of various heterogeneous and distributed services and applications into uniform workflow tasks. Our proposed task template model is illustrated in Figure 3.(a), consisting of the following three layers:

- *The logical layer* contains the *task interface* that models the input ports and output ports of a task template.

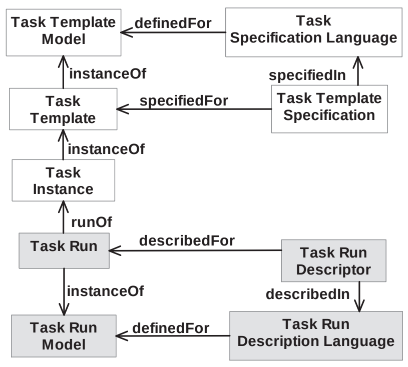


Figure 2. Main concepts and their relationships in a task model.

  In a scientific workflow, tasks are connected to one another via these ports through data channels. During workflow execution, tasks communicate with each other by passing data through data channels. The data type of each port is also defined as part of the task interface.

- *The physical layer* contains one or more *task components* that model the services or/and applications that are used to implement the task. The heterogeneous characteristics of a task component is modeled in this layer, including task type, inputs, outputs, location, invocation mechanism, authentication and protocol if needed.

- *The mapping layer* essentially consists of a list of mapping instructions that perform the mapping between the input/output ports of the task interface and the inputs/outputs of the task component. For each mapping, a shim is incorporated only if the type of input/output port and input/output are incompatible. All shims between input ports and inputs are formed an *inputports-to-inputs shim set*; while all shims between outputs and output ports are formed an *outputs-to-outputports shim set*.

The separation of the logical layer from the physical layer not only hides the implementation details of a task from its interface, thus providing a uniform interface of a task to the workflow engine, but also brings the opportunity to integrate various heterogeneous and distributed services and applications into a scientific workflow in a uniform way. However, the integration of heterogeneous services and applications into scientific workflows is challenging since these services/applications are often written in various programming languages, invoked via different invocation mechanisms and run in disparate computing environments. Currently, our proposed task template model focuses on the modeling of the following aspects of the heterogeneity of a task component:

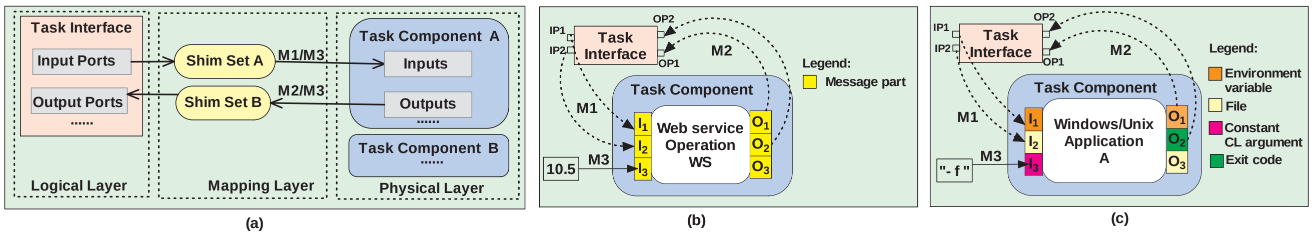


Figure 3. (a) An extensible task template model; (b) - (c) static mappings between input/output ports of a task interface and inputs/outputs of the task components WS and A.

- *Heterogeneous inputs of a task component.* A task component can take inputs from command line arguments (user-specified or constant), environment variables, input files, communication messages (e.g., SOAP messages for Web services), and the system standard input, etc.

- *Heterogeneous outputs of a task component.* A task component can produce outputs as environment variables, files, communication messages, the system standard output, the exit code, and the standard error, etc.

- *Heterogeneous invocation mechanisms.* Based on different computing environments, the types and locations of executables, various local and remote invocation mechanisms are modeled.

To hide the heterogeneous characteristics of a task component from the task interface, all the above heterogeneous aspects of a task component are modeled in the physical layer, while the mapping layer models the following three kinds of mappings between the input/output ports of the task interface and the heterogeneous inputs/outputs of a task component:

- The *inputports-to-inputs mapping* (M1) specifies how the input data taken from an input port IP_i of a task is mapped to an input I_j of the task component C. If IP_i is not mapped, then any data from IP_i will not be used by C. For each shim S in an inputports-to-inputs shim set, M1 contains the mapping between IP_i and the input of S and the mapping between output of S and I_j.

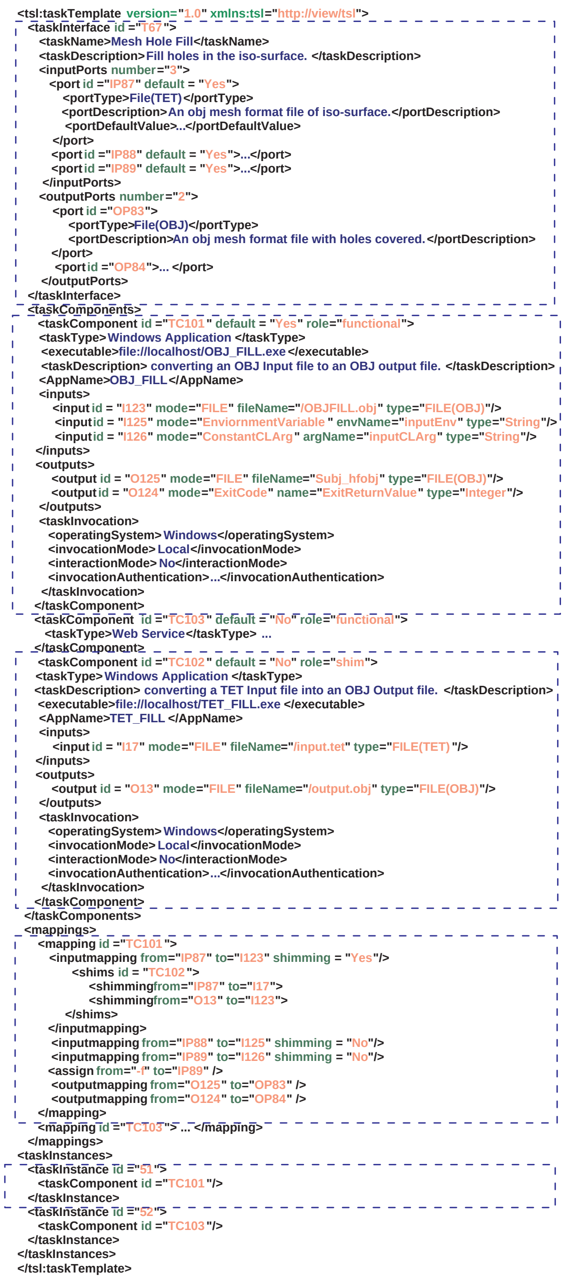
- The *outputs-to-outputports mapping* (M2) specifies how the output data produced from an output O_i of a task component is mapped back to an output port OP_j of the task. Similarly, if an output of a task component is not mapped, then such output data is discarded. For each shim S in an outputs-to-outputports shim set, M2 contains the mapping between O_i and the input of S and the mapping between output of S and OP_j.

- The *constant mapping* (M3) specifies a constant that will be assigned to an input of the task component before the execution of the task component. A constant mapping can also be used to assign a constant value to an output port of a task when the execution of the task component completes. Such flexibility is important to improve the configurability of a task template.

Figure 3.(b) - (c) illustrate two cases of the application of our proposed task template model: Web services and Windows applications. For simplicity, shims are not shown in these mappings. For M1, in a Web service operation WS, as shown in Figure 3.(b), the input port IP_1 is mapped to I_1, one part of the request message of WS; the input port IP_2 is mapped to I_2, a second part of the request message. For M3, a constant 10.5 is assigned to I_3, a third part of the request message. For M2, a part of the response message O_1 is mapped to the output port OP_1; O_2, a second part of the response message, is mapped to the output port OP_2; and O_3, a third part of the response message is not mapped, indicating that its value is discarded and never used afterwards. For Windows/Unix applications, both mappings are more sophisticated due to the rich modes of inputs and outputs. As illustrated in Figure 3.(c), for M1, the input port IP_1 is mapped to environment variable I_1, requiring that this environment variable be assigned the value from IP_1 before the execution of a Windows/Unix Application A, and such value will be taken as A's input; the input port IP_2 is mapped to file I_2, indicating that a file I_2 needs to be created with the content from IP_2 before the execution of A. For M3, a constant string of "-f" is assigned to I_3, indicating that the invocation of A is achieved via a constant command line argument of "-f". For M2, environment variable O_1 is mapped to output port OP_1, thus, after the execution of A, O_1 is produced as an environment variable and its value will be assigned to output port OP_1; the exit code O_2 is mapped to output port OP_2, therefore its value will be assigned to OP_2 after the execution of A; the execution of A will produce file O_3; however, since O_3 is not mapped, this file is discarded and will not be used afterwards. An optimization algorithm can delete such files to reclaim storage resources.

3 Shimming in TSL

In this section, we first propose an approach to the TYPE-II shimming problem, and then provide an algorithm to reduce the TYPE-I shimming problem to the TYPE-II shimming problem, and finally summarize the advantages of our shimming approach.



```
<tsl:taskTemplate version="1.0" xmlns:tsl="http://view/tsl">
 <taskInterface id ="T67">
   <taskName>Mesh Hole Fill</taskName>
   <taskDescription>Fill holes in the iso-surface. </taskDescription>
   <inputPorts number="3">
     <port id ="IP87" default = "Yes">
       <portType>File(TET)</portType>
       <portDescription>An obj mesh format file of iso-surface.</portDescription>
        <portDefaultValue>...</portDefaultValue>
     </port>
     <port id ="IP88" default = "Yes">...</port>
     <port id ="IP89" default = "Yes">...</port>
   </inputPorts>
   <outputPorts number="2">
     <port id ="OP83">
        <portType>File(OBJ)</portType>
        <portDescription>An obj mesh format file with holes covered.</portDescription>
     </port>
      <port id ="OP84">... </port>
   </outputPorts>
 </taskInterface>
 <taskComponents>
   <taskComponent id ="TC101" default = "Yes" role="functional">
   <taskType>Windows Application </taskType>
    <executable>file://localhost/OBJ_FILL.exe </executable>
    <taskDescription> converting an OBJ Input file to an OBJ output file. </taskDescription>
   <AppName>OBJ_FILL</AppName>
   <inputs>
      <input id = "I123" mode="FILE" fileName="/OBJFILL.obj" type="FILE(OBJ)"/>
      <input id = "I125" mode="EnviornmentVariable " envName="inputEnv" type="String"/>
      <input id = "I126" mode="ConstantCLArg" argName="inputCLArg" type="String"/>
   </inputs>
   <outputs>
      <output id = "O125" mode="FILE" fileName="Subj_hfobj" type="FILE(OBJ)"/>
      <output id = "O124" mode="ExitCode" name="ExitReturnValue" type="Integer"/>
   </outputs>
   <taskInvocation>
     <operatingSystem>Windows</operatingSystem>
     <invocationMode>Local</invocationMode>
     <interactionMode>No</interactionMode>
     <invocationAuthentication>...</invocationAuthentication>
    </taskInvocation>
   </taskComponent>
   <taskComponent  id ="TC103" default = "No" role="functional">
     <taskType>Web Service</taskType> ...
   </taskComponent>
   <taskComponent id ="TC102" default = "No" role="shim">
   <taskType>Windows Application </taskType>
   <taskDescription> converting a TET Input file into an OBJ Output file. </taskDescription>
    <executable>file://localhost/TET_FILL.exe </executable>
    <AppName>TET_FILL </AppName>
    <inputs>
      <input id = "I17" mode="FILE" fileName="/input.tet" type="FILE(TET) "/>
   </inputs>
   <outputs>
      <output id = "O13" mode="FILE" fileName="/output.obj" type="FILE(OBJ)"/>
    </outputs>
    <taskInvocation>
      <operatingSystem>Windows</operatingSystem>
      <invocationMode>Local</invocationMode>
      <interactionMode>No</interactionMode>
      <invocationAuthentication>...</invocationAuthentication>
     </taskInvocation>
   </taskComponent>
 </taskComponents>
 <mappings>
   <mapping id ="TC101 ">
     <inputmapping from="IP87" to="I123" shimming = "Yes"/>
          <shims id = "TC102 ">
              <shimmingfrom="IP87" to="I17">
              <shimmingfrom="O13" to="I123">
         </shims>
     </inputmapping>
      <inputmapping from="IP88" to="I125" shimming = "No"/>
      <inputmapping from="IP89" to="I126" shimming = "No"/>
     <assign from="-f" to="IP89" />
      <outputmapping from="O125" to="OP83" />
      <outputmapping from="O124" to="OP84" />
   </mapping>
   <mapping id ="TC103"> ... </mapping>
  </mappings>
<taskInstances>
  <taskInstance id ="51">
    <taskComponent id ="TC101 "/>
  </taskInstance>
  <taskInstance id ="52">
    <taskComponent id ="TC103 "/>
  </taskInstance>
</taskInstances>
</tsl:taskTemplate>
```

Figure 4. An example of a task template specification.

3.1 Addressing the TYPE-II shimming problem

According to the above task template model, an XML-based task template specification language, called *TSL*, is proposed to model heterogeneous and distributed services and applications, including shims. In TSL, both shims and functional task components are uniformly modeled as *task components* with the *shim* role and the *functional* role, respectively. A task component can be registered with a system with one role or both roles.

Due to space limit, we will not present the full syntax and semantics of TSL but illustrate it with an example. Figure 4 presents an example of task template specification (TTS) for a task template written in TSL. The logical layer, the physical layer, and the mapping layer are realized by the `taskInterface` element, the `taskComponents` element and the `mappings` element, respectively. At the logical layer, the `taskInterface` element contains sub-elements `inputPorts` and `outputPorts` to define the input and output ports of the task template.

At the physical layer, the `taskComponents` element contains a set of `taskComponent` elements, modeling either functional task components (specified by `role = "functional"`) or shim task components (specified by `role = "shims"`). Each functional `taskComponent` element specifies one possible implementation of the task interface of the task template. Similar to functional task components, shims are heterogeneous, distributed and system-independent. For each task component (shim or functional), we model its input/output information, invocation details, such as operating system, invocation mode (e.g., local or remote), interaction mode (interactive or non-interactive), and authentication information. Shims are introduced into `taskComponents` only if there is an inputports-to-inputs shim set or outputs-to-outputports shim set as a result of the TYPE-II shimming problem.

At the mapping layer, the `mappings` element contains the instructions for M1 (by the `inputmapping` element), M2 (by the `outputmapping` element) and M3 (by the `assign` element). If there is no shim for an inputmapping/outputmapping, the `shim` attribute inside the `inputmapping/outputmapping` is set to “No”; otherwise (`shim` = “Yes”), each `shimmings` element is encoded inside an `inputmapping` or `outputmapping` element. A `shimmings` element is uniquely identified by a shim’s `taskComponent id`. The `shimming` elements are encoded inside the `shimmings` element to provide the mappings among input/output ports, inputs/outputs of task components and input/outputs of shims.

The `taskInstances` element contains all task instances that are instantiated from the same task template and hence share the same task interface. In our model, we consider all functional task components in a task template is *functionally* equivalent but might have different implementations and deployments and thus might provide different types of inputs and outputs. Each task instance uses a unique functional component, which uniquely identifies the necessary mapping and shimming to provide the same

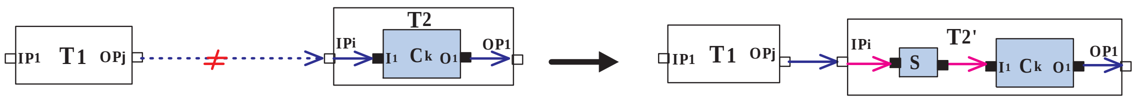


Figure 5. Reducing the TYPE-I shimming problem to the TYPE-II shimming problem.

task interface. Therefore, in TTS, each task instance encoded in the `taskInstance` element contains one specific functional task component from alternative task components provided by the task template. The taskComponent's `id` inside each `taskInstance` can be used to retrieve the corresponding inputmapping and outputmapping of this task component.

Essentially, our example of task template specification, called *Mesh Hole Fill (MHF)*, provides three input ports and two output ports at the interface. MHF encapsulates two functional task components: one is called OBJ_FILL (`taskComponent id = TC101`), a Windows application that can be locally executed without user interaction. Another functional component encapsulated in MHF is developed as a Web service (`taskComponent id = TC103`). OBJ_FILL has three inputs with the modes of file, environment variable and constant command-line argument. Two outputs are defined with the modes of file and exit code. As the input of OBJ_FILL (`input id = I123`) is incompatible with the inputport (`port id=I123`) in input mapping, a shim (`taskComponent id = TC102`) is incorporated into the physical layer and the mapping layer of the TTS.

3.2 Addressing the TYPE-I shimming problem

We propose a reduction algorithm that reduces the TYPE-I shimming problem to the TYPE-II shimming problem and provide a transparent solution to both problems. As shown in Figure 5.(a), given two task instances T_1 and T_2, in which T_2 encapsulates functional task component C_k. When the type of output port $T_1.OP_j$ is incompatible with the type of input port $T_2.IP_i$, a TYPE-I shimming problem occurs. A new task template T_2' can be created from T_2's task template by encapsulating an appropriate shim S and C_k inside, and then an instance of T_2' can be used as a replacement of T_2. The pseudocode of the reduction algorithm, *ReduceTYPE-I2TYPE-II*, is sketched in Figure 6. First, the TTS of T_2' is copied from the TTS of T_2. Second, if possible, a suitable shim S is retrieved automatically based on the types of $T_1.OP_i$ and $T_2.IP_j$. Finally, different layers of T_2' are updated accordingly, in particular, T_2''s input port is mapped to S's input and S's output is mapped to the input of the task component C_k.

```
Algorithm: ReduceTYPE-I2TYPE-II
Input: TypeOf($T_1.OP_i$): a type of a task instance $T_1$'s output port $OP_i$ and
       TypeOf($T_2.IP_j$): a type of task instance $T_2$'s input port $IP_j$
Output: a new task instance $T_2'$ initialized by a new task template $T$
Begin
(1) If TYPE-I problem occurs
(2) Then Retrieve a shim from system or third-party
(3)   If ∃ a shim $S$ and TypeOf($S.in$) = TypeOf($T_1.OP_i$) and TypeOf($S.out$) =
      TypeOf($T_2.IP_j$)
(4)   Then
(5)       Create new task template $T$ by copying $T_2$'s TTS
(6)       Initialize a instance $T_2'$ based on $T$
(7)       TypeOf($T_2'.IP_j$) = TypeOf($T_1.OP_i$) /*update TTS's logical layer*/
(8)       Add $S$ into $T$'s taskComponents /*update TTS's physical layer*/
(9)       Map $T_2'.IP_j$ to $S.in$ /*update TTS's mapping layer*/
(10)      Map $S.out$ to the input of $T_2'$'s task component $C_k$
(11)  Else
(12)      Report to Type Match Error
(13)Else
(14)  No shim required to reduce
End Algorithm
```

Figure 6. Algorithm ReduceTYPE-I2TYPE-II

3.3 Advantages of Our Approach

we identify the following advantages of our shimming approach:

1) *Transparent shimming*. This is the first shimming technique that hides all shimming and mapping details inside a task interface and thus produces scientific workflows in which all shims are invisible. As a result, a scientist can better focus on the functional part of a scientific workflow without being distracted by the clutter of shims, which are usually not science-relevant to the scientist but are technically needed.

2) *Addressing both TYPE-I and TYPE-II shimming problems*. This is the first solution that addresses the TYPE-II shimming problem. Moreover, our approach enables the reduction of the TYPE-I shimming problem to the TYPE-II shimming problem, providing a consistent solution to both types of shimming problems.

3) *System and language independent*. Since our shimming technique is based on an XML-based TSL language, which models all the details of abstraction, shimming and mapping. TSL can be implemented by different systems using different languages and thus provides a system and language independent solution.

4) *Reusable and extensible*. In our approach, similar to functional task components, shims can be arbitrary local and remote heterogeneous services and application written in various languages and run in different platforms. As a result, shims are reusable across tasks, workflows and systems. Moreover, TSL is easily extensible for more sophisti-

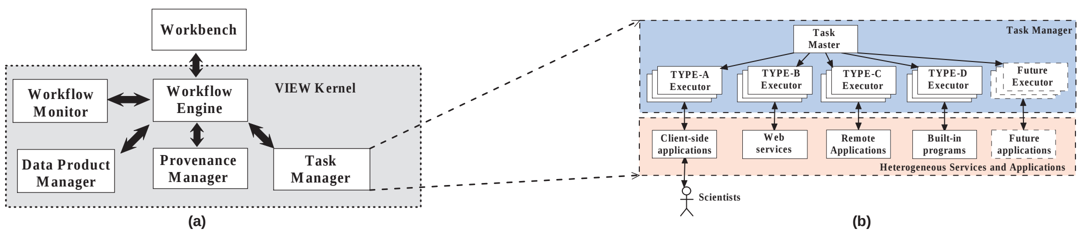


Figure 7. (a) Service-oriented architecture for the VIEW system; (b) VIEW Task Manager for the execution of heterogeneous shims and functional task components.

cated shimming techniques, such as the composition of basic shims to construct composite shims.

4 Service-Oriented Architecture for Task Management

We have previously designed a service-oriented architecture for VIEW, a visual scientific workflow management system [8]. As shown in Figure 7.(a), the VIEW system consists of six loosely-coupled subsystems: a *Workbench* to visually design workflows, a *Workflow Engine* to schedule workflows, a *Task Manager* to dispatch and execute workflow tasks, a *Provenance Manager* to store and query provenance, a *Data Product Manager* to store, query and manage data products, and a *Workflow Monitor* to monitor workflow and task execution status. Here, we propose a service-oriented architecture for the *Task Manager* to provide run-time support for our proposed task template model, enabling the parallel execution of tasks (shims and functional task components) in a heterogeneous and distributed environment.

As shown in Figure 7.(b), the architecture of the Task Manager consists of a *Task Master* and a set of *Task Executors*. The Task Master manages all task templates, task instances, and task runs, while Task Executors are responsible for the invocation and execution of various heterogeneous task components. Four types of Task Executors are proposed but the extensibility are provided for future types of Task Executors:

1) A *TYPE-A* executor provides an execution environment mostly for user-interaction and visualization intensive tasks, or the tasks that can be executed in the host of the TYPE-A executor. A TYPE-A executor is typically deployed at a client-side machine such that a user can view and interact with the graphical user interfaces of tasks assigned to the executor. Each TYPE-A executor is required to communicate *remotely* with the Task Master and *locally* with tasks. To avoid the clutter of display, tasks are executed sequentially in an execution environment provided by a TYPE-A executor.

2) A *TYPE-B* executor provides an execution environment mostly for tasks with tasks components being Web services, whose interfaces are described by WSDL. A TYPE-B executor can be deployed either at the host of the Task Master or at any other standalone host. Each TYPE-B executor is required to communicate *remotely* with tasks, which can be executed in parallel.

3) A *TYPE-C* executor provides an execution environment for tasks that are registered and specified to execute on remote systems, including the underlying high-end computing environment, such as Grids and Clusters. Typically, those tasks require long-duration back-end computations without user interactions. This type of executors can be deployed either at the host of the Task Master or at any other standalone host. Each TYPE-C executor is required to communicate *remotely* with tasks, which can be executed in parallel.

4) A *TYPE-D* executor provides an execution environment for built-in tasks and those that are registered and specified to execute at the host where the Task Master is deployed. Those built-in tasks can be hard-coded into the subsystem and installed with the Task Master. Each TYPE-D executor communicates *locally* with both the Task Master and tasks, and tasks can be executed in parallel.

Different types of Task Executors implement different internal functions to accommodate tasks using programming languages and invocation mechanisms, but all of them provide uniform interfaces to the Task Master on one hand and uniform interfaces to services and applications on the other hand. The architecture of Task Executors are extensible in nature: to support new types of task components in the future, it is only required for a particular Task Executor to add new functions to incorporate their invocation methods without affecting other Task Executors and the Task Master.

5 Implementation and a Case Study

We implemented our proposed models and architecture in VIEW 2.1 system. The Task Master and Task Executors are developed using Microsoft Visual Studio 2005 and Microsoft .NET Framework and implemented as Web Services. The communications among the Task Master, Task Executors and other VIEW service components are based on synchronous/asynchronous SOAP messages via the HTTP protocol.

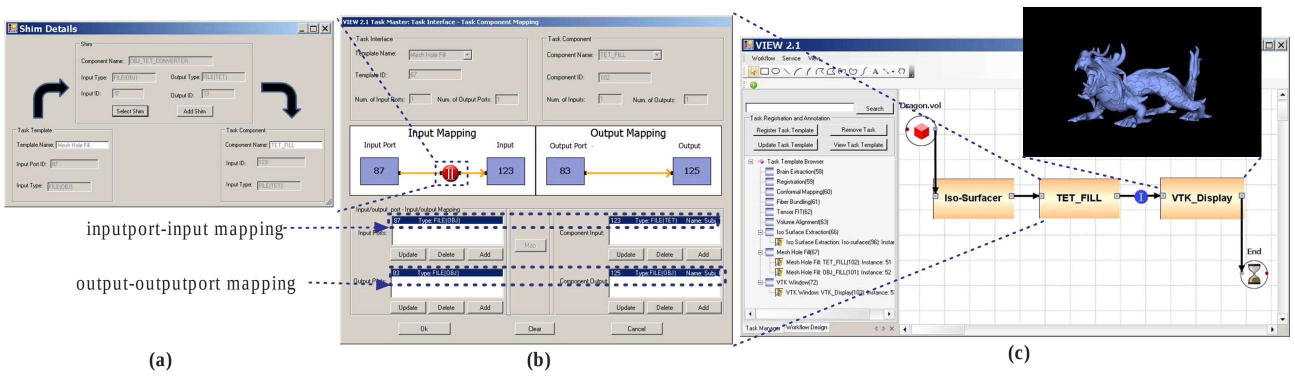


Figure 8. A scientific workflow composed in the VIEW 2.1 system with shims to the TYPE-I and TYPE-II problem.

Figure 8 presents a typical scientific workflow designed in VIEW 2.1 for surface extraction from volume data, a required preprocessing step for surface analysis. The workflow is composed of three task instances: the first is the *Iso-Surfacer* task instance which uses the marching cubes algorithm to extract the surface from volume data. The second task instance, *TET_FILL*, analyzes the extracted surface to fill holes generated in an image file. The resulting surface is rendered in a 3D-interactive display using *VTK_Display* task instance as shown in Figure 8.(c). The data types of input/output ports for each task instance are listed as follows: the Iso-surfacer task instance reads a volume file formatted as *VOL* from its inputport, and output a file formatted as *OBJ*; the inputport and outputport of TET_FILL task instance are typed as File(OBJ); The VTK_Display task instance read a VTK file and visualize it on a display window.

The *TET_FILL* task instance is initialized by the *Mesh Fill Hole* task template which encapsulates two task components: the *TET_FILL* task component is a third-party Windows application using C++, invoked by a TYPE-D Executor. Another task component is called *OBJ_FILL*, implemented by a Web service that receives and outputs a datastream encoded in SOAP messages. This task component is invoked by a TYPE-B Executor. The *Mesh Fill Hole* task template's TSL can be viewed by Task Template Browser in Figure 8.(c) and stored in the VIEW Task Master.

Figure 8.(c) illustrates the Type-I shim that occurs between TET_FILL and VTK_Display task instances. The input port of VTK_Display is typed as File(VTK), incompatible with the type of TET_FILL's output defined as File(OBJ), then a Type-I shimming problem is detected automatically by the system (see the blue Type-I shimming detection icon in Figure 8.(c)). By clicking the icon, the system allows scientists to either select system-provided shims or register any third party shims if there is no existing shim available. In addition, the system allows scientists to automatically hide shims inside a task instance by applying our proposed ReducingType-I2Type-II algorithm.

The Type-II shim problem in this workflow occurs when mapping from the TET_FILL task instance's inputport to an input of its task component. The type of the input port is defined as File(OBJ), while the input requires a tetrahedral mesh file typed as File(TET). The incompatibility is automatically detected by system with the red Type-II shimming detection icon in Figure 8.(b). After clicking the icon, a system-provided shim called OBJ_TET_CONVERTER is automatically applied to the input mapping. Figure 8.(a) shows the shimming between the input port (ID:87,Type:File(OBJ)) and the shim's input (ID:17,Type:File(OBJ)), and the shimming between the shim's output (ID:13,Type:File(TET)) and the task component input (ID:123,Type: File(TET)). The implementation details of the OBJ_TET_CONVERTER shim is encoded in the Mesh Fill Hole task template's TTS, which is implemented as a Windows application using C++ and invoked remotely by a TYPE-A Executor.

6 Related Work

The term "shims" and the shimming problem were first introduced in [6]. In an open world such as the Web, the shimming problem is unavoidable when third-party autonomous and heterogeneous services and applications are used to compose scientific workflows, but the output of one task is *incompatible* with the input of another task. Incompatibility comes in two forms: 1) Although the output of a task is syntactically compatible with the input of another task, they are still not compatible semantically. For example, both tasks might use `xsd:string` to encode underlying different complex data types. 2) Although the output of a task is syntactically incompatible with the input of another task, they could still be semantically equivalent. For example, DNA sequences might be represented in different formats and data types, which are semantically equivalent. In both cases, shims are proposed as the treatment of the shimming problem and are defined as the "software that transforms between closely related (either syntactically or semantically) in order to join outputs and inputs of two components" in [6].

Several shimming techniques have been proposed to address the shimming problem. Szomszor et al. [12] proposed an architecture to support automatic translation between two semantically equivalent but syntactically different XML documents. This technique does not address the first form of incompatibility and the translation of other data types. Bowers and Ludäscher [10] proposed an ontology-based approach to the shimming problem by associating each port of a task with an XML-based *structural type* and an ontology-based *semantic type*, respectively. An output port of an upstream task can be directly connected to an input port of a downstream task if and only if these two ports are both semantically and syntactically compatible (called *semantically valid* and *structurally valid* in their terms). If two ports are semantically compatible but syntactically incompatible, then an XML shim is created whenever possible to mediate the two ports. Similarly, the solution is limited to shims that perform data transformation on XML data. Ambite and Kapoor [1] proposed a planning approach to automatically construct scientific workflows that process relational data. Shims can be automatically inserted into the workflow when necessary. However, only shims that process relational data are supported within this framework. Hull et al. [5] proposed that semantic types should be related to each other by other relationship types, such as *has-Part* in addition to the subsumption relationship. In this way, a richer types of shims can be created, such as *extractor* shims. A preliminary classification of shims are available in [6]. Existing scientific workflow management systems [9, 11, 2] provide limited support to the TYPE-I shimming problem; shims are visible in these systems. Finally, our previous version of the VIEW system [8, 7] focused on the investigation of a reference architecture for SWFMSs, while this paper focuses on addressing the shimming problem.

None of the above techniques addresses the TYPE-II shimming problem, which is the focus of this paper. Our approach provides a general solution to both TYPE-I and TYPE-II shimming problems and naturally converts the TYPE-I shimming problem to the TYPE-II shimming problem by only XML-based TTS operations. Shims, similar to functional components, can be executed in heterogeneous and distributed environments. Once they are registered, they can be discovered and reusable to other workflows. In addition, our approach uniquely allows shims to be either invisible or visible at the workflow level, supporting both functional and operational perspectives of scientific workflows.

7 Conclusions and Future Work

In this paper, we firstly proposed a task template model which encapsulates the composition and mapping of shims and functional task component within a task interface. Second, we designed an XML-based task specification language, called TSL, to realize the proposed task template model. Third, we proposed a service-oriented architecture for task management to enable the distributed execution of shims and functional task components. Finally, we implemented the proposed models, languages and architecture and presented a case study to validate them. Ongoing work includes the extension of the TSL to support more types of shim services, such as P2P services and Grid services, and the development of various scientific workflow applications enabled by the presented techniques.

References

[1] J. Ambite and D. Kapoor. Automatically composing data workflows with relational descriptions and shim services. In *ISWC/ASWC*, pages 15–29, 2007.

[2] S. Callahan, J. Freire, E. Santos, C. Scheidegger, C. Silva, and H. Vo. VisTrails: visualization meets data management. In *SIGMOD Conference*, pages 745–747, 2006.

[3] S. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD Conference*, pages 1345–1350, 2008.

[4] E. Deelman and A. Chervenak. Data management challenges of data-intensive scientific workflows. In *CCGRID*, pages 687–692, 2008.

[5] D. Hull, R. Stevens, and P. Lord. Describing web services for user-oriented retrieval. In *Proc. of W3C Workshop on Frameworks for Semantics in Web Services*, pages 9–10, 2005.

[6] D. Hull, R. Stevens, P. Lord, C. Wroe, and C. Goble. Treating shimantic web syndrome with ontologies. In *AKT-SWS04*, 2004.

[7] C. Lin, S. Lu, X. Fei, A. Chebotko, Z. Lai, D. Pai, F. Fotouhi, and J. Hua. A reference architecture for scientific workflow management systems and the VIEW SOA solution. *IEEE Transactions on Services Computing*, 2(1):79–92, 2009.

[8] C. Lin, S. Lu, Z. Lai, A. Chebotko, X. Fei, J. Hua, and F. Fotouhi. Service-oriented architecture for VIEW: A visual scientific workflow management system. In *IEEE SCC*, pages 335–342, 2008.

[9] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[10] B. Ludäscher, S. Bowers, T. McPhillips, and N. Podhorszki. Scientific workflows: More e-science mileage from cyberinfrastructure. In *e-Science*, pages 145–152, 2006.

[11] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, R. Greenwood, T. Carver, K. Glover, M. Pocock, A. Wipat, and P. Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[12] M. Szomszor, T. Payne, and L.Moreau. Automated syntactic medation for web service integration. In *ICWS*, pages 127–136, 2006.