Combining Workflow Templates with a Shared Space-based Execution Model

Javier Rojas Balderrama∗, Matthieu Simonin∗, Lavanya Ramakrishnan†, Valerie Hendrix†, Christine Morin∗, Deborah Agarwal†, Cédric Tedeschi†∗

∗INRIA. Rennes, France
†Lawrence Berkeley National Lab. Berkeley, USA
‡University of Rennes 1. Rennes, France

Abstract—The growth for scientific data has led to data analysis being a critical step in the scientific process. The next-generation scientific data analysis environment needs to address two challenges i) productivity of the end-user and ii) scalability of the workflows. The need to ensure both goals requires us to revisit the design and implementation of workflow tools. In this paper, we study the interaction of Tigres and HOCL-TS towards meeting these goals. Tigres and HOCL-TS have evolved separately; however their complementary foci allows us to study these issues in greater detail. We describe the pros and cons of an approach that integrates Tigres and HOCL-TS and HOCL-TS extension to support common non-functional requirements such as logging and monitoring that can be made available to the users through the Tigres API.

I. INTRODUCTION

The scientific community is generating large data sets at increasing rates leading to data analysis being integral to the scientific process. Existing data analysis methods, tools and infrastructure are often difficult to use and unable to handle the “data deluge”. A scientific data analysis environment needs to address two key challenges: i) productivity: allow users to easily compose and execute analysis algorithms while focusing on the science problem and not the computational elements, and ii) scalability: allow large-scale workflows to be run efficiently.

The development of data analyses pipelines evolves often with project needs and if not tackled properly, leads to codes that can be very costly to maintain and extend. Workflow tools allow users to capture their pipelines and ensure provenance tracking and reproducibility (e.g., Taverna [1], Kepler [2], and MOTEUR [3]). However, the need to ensure both productivity and scalability in next-generation analysis environments requires us to revisit the design and implementation of workflow tools.

The DALHIS project that includes researchers from INRIA (France) and Lawrence Berkeley National Lab (USA), targets the development of a software ecosystem to facilitate seamless data analysis across desktops, HPC and cloud environments. In the context of this project, we have studied the interaction of two important components in a workflow ecosystem: Tigres and HOCL-TS. Tigres [4] provides a workflow library that allows users to express workflows in common programming languages like Python and C. HOCL-TS [5] is a decentralized workflow engine based on a shared-space based service coordination model inspired by the chemical programming model.

Specifically, this paper makes the following contributions:

• We introduce the design and early implementation of the DALHIS workflow manager, which integrates Tigres and HOCL-TS.

• We describe the HOCL-WMS execution model based on a decentralized coordination mechanism that improves the scalability of workflow executions.

Section II briefly recalls the state of the art in scientific workflow management, with a focus on programming models and run-time environments. Section III describes the DALHIS workflow manager system and presents the current state of the two building blocks considered: Tigres and HOCL-TS. Section IV exposes more concretely: a) the translation of Tigres templates to HOCL programs, and b) the steps envisioned towards the implementation of non-functional properties such as logging and monitoring. Section V discusses the positioning of this work in the landscape of workflow management systems. Section VI concludes and gives hints for future work.

II. BACKGROUND

Scientific workflows are designed to support the automation of complex, service-based data-intensive applications. They tend to combine a dataflow model, whereby a flow consists of a set of services (tasks) that are associated through data dependencies, with a functional model that accounts for collection-oriented processing. This combination of models is designed to strike a balance between expressiveness and simplicity. The data-driven approach is appealing to the distributed computing community due to implicit parallelism (i.e., the ability to automatically interpret and exploit the parallelism contained in computations expressed by dataflow constructs or patterns). Note that parallelism in workflows can be found at different levels. First, a workflow can be instantiated in parallel on different input data. Second, the same goes at the service level. Finally, in a workflow, two tasks that are not linked by some data dependency may run in parallel.

In Section II-A, we describe the programming models behind Tigres and HOCL-TS. Tigres builds on the idea of reusable templates to define communication patterns. HOCL-TS focuses on the enactment of the workflow orchestration
in a decentralized fashion. After briefly positioning HOCL-TS in the landscape of workflow tools in Section II-B, we recall the importance of the support of non-functional requirements, in particular logging and monitoring. We then briefly position HOCL-TS in this regard by emphasizing how the shared-space based coordination model can be naturally leveraged to support them.

A. Programming models

Programming patterns have been extensively used before in scientific and HPC environments [6]. The Message Passing Interface (MPI) standard [7], shared-memory models as implemented for parallel machines by the OpenMP standard [8] or the thirteen dwarfs [9] are examples of traditional programming patterns used in HPC. Note that Tigres uses templates at the higher-level of the software stack and this has not been explored before. MapReduce [10] is based on the idea of moving computation to data. Our earlier evaluation shows that MapReduce has significant gaps for general scientific workflow [11], [12].

At the origin of HOCL is Gamma, a programming model introduced in [13] where computation can be seen as chemical reactions arising between data represented as molecules floating in a chemical solution. More formally, chemical models use i) a multiset as a base structure where to put these molecules of data, and ii) a set of rules rewriting the multiset, to be applied in no specific order at run time. A rule consumes some molecules from the multiset satisfying a pattern and a condition to create a new set of molecules to replace the molecules consumed. HOCL is a language based on the \( \gamma \)–calculus [14], a higher-order chemical computation model generalizing Gamma. The HOCL-TS architecture is a workflow execution system using HOCL as a workflow descriptor. It enables a decentralized workflow execution through the use of the multiset as a shared space through which different HOCL interpreters will coordinate.

B. Workflow tools

Workflow tools have been developed to represent and run scientific processes mostly in distributed environments. Over the last decade, various workflow tools such as Taverna [1], Kepler [2], MOTEUR [3] Galaxy[15], Pegasus [16], Triana [17], Swift [18], Trident [19], Makeflow [20] have been developed, that allow users to compose their applications and services into a logical sequence. The Condor DAGMan [21] uses the graph representation to manage dependencies between jobs and hence acts as a meta-scheduler for jobs submitted to a Condor system.

CloudWF [22] and Oozie [23] are workflow systems built atop Hadoop for cloud environments. The Spark [24] framework uses a data abstraction called resilient distributed data sets. Tigres differs from these tools as it is a workflow library rather than a workflow system.

Workflow managers are mostly designed using a centralized architecture despite potential weakness. Centralized solutions, suffer from poor scalability and low reliability due to communication bottlenecks and single points of failures. Early works [25], [26], [27] proposed decentralized workflow coordination through the exchange of messages. This coordination mechanism implied a tight coupling of services in terms of spatial and temporal composition. Other works [28], [29], [30] as well as HOCL-TS provide a decentralized execution based on the principles behind the coordination found in data-driven languages such as Linda [31] where distributed shared spaces store data as tuples. This allows loose-coupling in space and time among processes. HOCL-TS mainly differs from these approaches by not only storing data, but also control information in the shared space.

C. Logging and monitoring

Users commonly want to track down the status of their workflows. They need to receive notifications upon execution anomaly and failure, then perform some troubleshooting and have an automated analysis of the workflow to help them categorize and qualify the results [32]. Logging and monitoring of scientific workflows are useful for performance analysis, to collect provenance data and gather feedback for future decisions, related to the optimization of resource usage.

Logging facilities are necessary for scientific workflows to meet requirements of reliability, security, and accountability [33]. A workflow manager needs to record its execution history in sufficient detail to provide efficient retrieval, performance monitoring, and the ability to restore the current state of an execution after data have been revised. A log service allows users to monitor workflows without excessive audit of trails as well as to recover its current state or an earlier state after a failure.

Real-time monitoring has many potential benefits, such as knowledge extraction, performance analysis, provenance, and visualization. It is important to distinguish two levels of monitoring during the execution of a workflow [34], namely, a low-level monitoring from instrumented entities required for enactment, and a high-level aggregated description of the experiment which concerns the execution of services and the progress of the workflow.

The DALHIS workflow manager, described in the following section, besides combining the programmability of Tigres and the scalability of HOCL-TS, will focus on integrating tools for logging and monitoring so as to offer to the user a way to deal with the tracking/debugging of workflows. In particular, we argue that a shared-space based coordination model eases the development of such features.

III. OVERVIEW

The DALHIS workflow manager targets the integration of HOCL-TS with Tigres. In other words, the DALHIS workflow manager is directed to offering an HOCL-TS-based execution of Tigres workflows. More generally, the idea is to leverage the inherent decentralization of HOCL in order to improve the scalability of Tigres workflow executions. As we detail in the following, Tigres workflows consist in an ordered set of templates (i.e., sequence, parallel, split, merge), each of these templates combining a set of tasks. Note that the current Tigres execution system supports various execution mechanisms to run these tasks (a task execution can be either local or submitted to a computation platform through a batch scheduler such as Sun Grid Engine [35]).
The planned integration of HOCL-TS in the workflow execution process is located at the template level. Figure 1 depicts the execution scenario envisioned for the DALHIS workflow manager. The global coordination of the workflow is left to Tigres, while each template is dynamically translated into an HOCL workflow (HOCL-WF) and sent to the HOCL workflow management system (HOCL-WMS), a user-friendly encapsulation of HOCL-TS architectures responsible for the decentralized execution of the translated pattern.

In Section III-A and III-B, we present the Tigres workflow library and the HOCL-TS engine in more detail, respectively. Section IV-A describes the process of translating a Tigres template to an HOCL workflow. The HOCL-WMS is described in Section IV-B.

Fig. 1: DALHIS workflow manager: execution scenario.

A. Tigres

The Template Interface for Agile Parallel Data-Intensive Science (Tigres) provides a library to compose and execute large-scale data-intensive scientific workflows on variety of resources including clusters and HPC systems. Tigres has a new concept of reusable “templates” that enable scientists to easily compose, run and manage scientific workflows. These templates define common computation patterns used in analyzing a data set.

Templates. The API supports four templates (sequence, parallel, split and merge). Figure 2 shows the Tigres templates.

Concepts. Tigres supports some key concepts:

A Template’s inputs are a TaskArray, that is a collection of tasks to be executed together, and an InputArray with the corresponding inputs to the tasks. The TaskArray is a collection of tasks that need to execute with some connection between them (i.e., they are part of a sequence or parallel).

A Task is defined using InputTypes that defines the type of inputs that the task takes. The InputArray is a collection of InputValues where InputValues are the values that are inputs to the task. Its types match what is defined in the InputTypes.

InputValues are passed to the task during execution and not included in the task definition.

Architecture. Tigres has a layered architecture that consists of a) User API, b) Core API, c) State Management, d) Execution Management and e) Monitoring. The user uses the external User API to define the elements of their workflows, log messages and monitor their program execution. The Core API supports graphing, creating tasks, managing task dependencies and running templates. Monitoring is used to log system or user-defined events. State management encompasses different management aspects of the workflow including monitoring and provenance. It maintains state as each template is invoked and manages integrity of the running Tigres program. The execution layer can work with one or more resource management techniques including desktop, cluster and HPC systems. The idea in this paper is to explore using the Tigres interface with HOCL-TS execution mechanisms.

B. HOCL-TS

HOCL-TS [5] is a recent workflow run-time environment using chemical programming as a high-level workflow execution language and a shared-space based coordination model to decentralize the workflow management at run time. Such a combination targets primarily the injection of decentralization and dynamics in the management of workflows. It also shows an inherent facility for logging and monitoring such an execution.

The language. The Higher-Order Chemical Language (HOCL) is a language inspired by the chemical programming model [14]. Following the chemical metaphor, computation is seen as a set of reaction rules applied on a multiset of molecules representing data. When writing a chemical program, a user assumes that the different defined rules can be applied in any order without compromising the validity of the outcome of the program. This assumption allows to relieve the programmer from the burden of specifying useless control flows to solve inherently parallel problems while letting him concentrate on the functional aspects of it. The execution is data-driven: the presence of a molecule suffices to trigger a reaction requiring it. Nevertheless, sequential can still be expressed if needed.

More formally speaking, an HOCL program is a multiset of data and rewriting rules, rules being applied non-deterministically on data. Instead of going into the details of the syntax, let us consider a simple example. Consider the following HOCL program extracting the maximum value among a set of numbers.

\[
\text{let } \max = \text{replace } x, y \text{ by } x \text{ if } x \geq y \text{ in } \langle 2, 3, 5, 8, 9, \max \rangle
\]

The \( \max \) rule consumes any two integers \( x, y \) with \( x \geq y \) and produces a new integer taking the value of \( x \). Initially, several reactions are possible: \( \max \) can react with any couples of integers satisfying the condition: 2 and 3, 2 and 5, and 8, and 9. One scenario among the set of possible scenarios is the following:

\[
\langle 2, 3, 5, 8, 9, \max \rangle \rightarrow^* \langle 3, 5, 9, \max \rangle \rightarrow^* \langle 9, \max \rangle
\]

where \( \rightarrow^* \) models the concurrent application of the rule. Merely looking at these two execution steps is not enough...
to infer what pairs of numbers reacted together. Maybe 2 and 5 reacted together while 8 and 9 were also reacting.

Note that the only needed constraint is the so-called atomic capture which states that one molecule can react at most once. A molecule is always consumed in a reaction. As a corollary, when \( x, y \) is replaced by \( x \), the two \( x \)s are actually two different molecules, the first one being consumed in the reaction, the second one being created in the reaction. In other words, no molecule ever survives a reaction. The atomic capture may become problematic in distributed settings where multiple nodes may actually apply rules at the same time. These nodes may then know a particular molecule and thus may want to use it to fire (possibly different) rules. In this case, ensuring the atomic capture requires some distributed protocol to be applied between these nodes so as only one node is granted the right to use it, the others giving up their reaction. A possible approach for such a protocol has been devised in [36].

The workflow execution environment. HOCL-TS is an environment inspired by HOCL, but with the goal of using it as a coordination model so as to allow to decentralize the workflow execution control. The philosophy behind HOCL-TS stands in the following key ideas: a) the workflow is defined as an HOCL (multiset) program, b) this program is executed in part by a set of HOCL co-engines, c) the HOCL engines execute the workflow by reading and rewriting the multiset, and d) as a consequence, the progress/state of the workflow is captured by the current content of the multiset.

The HOCL-TS architecture, depicted in Figure 3, comprises two main types of elements: the Chemical Web Service (ChWS) and the multiset. A ChWS is an encapsulation of a Web service. It is co-responsible with other ChWSes of the coordination of the execution of workflow, as no central coordinator is available.

Each ChWS is made of an HOCL interpreter, acting as one co-engine, a local storage space used by the HOCL engine and containing a copy of the sub-solution corresponding to this service, and the service caller (i.e., the interface between the local co-engine and the actual service to be called). The multiset acts as a space shared by all ChWSes involved in the workflow, containing an HOCL description of the state of the workflow execution. Let us consider the 4-service workflow defined in Figure 4. The multiset/solution contains as many sub-solutions as services. Dependencies between ChWSes are expressed through molecules of the form \texttt{DEST:ChWS\textsubscript{i}} with \texttt{ChWS\textsubscript{i}} being the destination ChWS where some information needs to be transferred. For instance, ChWS\textsubscript{1} will transfer some information (typically its outcome) to ChWS\textsubscript{2} and ChWS\textsubscript{3}.

Notice the rule contained in ChWS\textsubscript{2}. It basically expresses the fact that, as soon as ChWS\textsubscript{1}’s result appears in this sub-solution, a \texttt{CALL:S\textsubscript{2}} molecule can be created, which will in turn be consumed by some generic rule that will actually call the service. To execute the workflow, some additional generic rules (i.e., common to every workflow) are needed.

For the sake of illustration, three of the generic rules are defined in Figure 5. The \texttt{invokeServ} rule encapsulates the actual invocation of services. When reacting, it invokes the service \( S_i \) by consuming the tuples \texttt{CALL:S\textsubscript{i}} representing the invocation itself, and \texttt{PARAM:⟨in\textsubscript{1},...,in\textsubscript{n}⟩} representing its input.

---

1. Knowing a molecule requires to discover it. As argued in [36], this discovery can be built atop a distributed hash table.
2. The HOCL-TS architecture was initially defined in the context of service composition. However, the concept can be applied to any kind of tasks such as system executables or user scripts, etc. For the sake of consistency, we will keep the same vocabulary in this section.

---

```plaintext
## Sequence (name, task_array, input_array)
output [] = Sequence("sequence", task_array_12, input_array_12)

## Parallel (name, task_array, input_array)
output [] = Parallel("parallel", task_array_12, input_array_12)

## Split (name, split_task, split_input_values, task_array, task_array_in)
Split("split", task_x1, input_value_1, spl_t_arr, spl_i_arr)

## Merge (name, task_array, input_array, merge_task, merge_input_values)
Merge("merge", syn_t_arr, syn_i_arr, task_x1, input_value_1)
```

---

Fig. 2: Tigres templates

---

Fig. 3: HOCL-TS architecture
\[
\begin{align*}
\text{let invokeServ =} & \\
\text{replace ChWS}_1: \{\text{CALL : } S_1, \text{PARAM : } \{\text{val}_1\}\} & \text{by } \text{ChWS}_1: \{\text{RESULT : ChWS}_1: \{\text{res}_1\}, \omega\}
\end{align*}
\]

\[
\begin{align*}
\text{let preparePass =} & \\
\text{replace ChWS}_1: \{\text{RESULT : ChWS}_1: \{\text{res}_1\}, \text{DEST : ChWS}_2, \omega\} & \text{by } \text{ChWS}_1: \{\text{PASS : ChWS}_2: \{\text{res}_1\}, \text{RESULT : ChWS}_3: \{\text{res}_2\}, \text{DEST : ChWS}_4, \omega\}
\end{align*}
\]

\[
\begin{align*}
\text{let passInfo =} & \\
\text{replace ChWS}_1: \{\text{PASS : ChWS}_2: \{\text{res}_1\}, \omega\} & \text{by } \text{ChWS}_1: \{\omega\}, \text{ChWS}_1: \{\omega, \omega\}
\end{align*}
\]

**Implementation.** The HOCL-TS software prototype was developed and experimented over a large-scale platform using different scientific workflows [5]. It relies on ActiveMQ for the communications between the ChWSes and the multiset: the multiset is encapsulated into an ActiveMQ server to allow its concurrent, periodic reading and writing by the co-engines. The sub-solutions are read periodically by co-engines, locally stored and processed. If some rule is triggered and the sub-solution is modified, it is pushed back to the multiset so as to be made available to other co-engines. The service caller was implemented using the DAIOS invocation framework [37].

**Execution scenario.** Figure 6 shows the evolution of the multiset during the first steps of the execution. To explicit, the generic rules have been added in each chemical sub-solution. When triggered, generic rules appear in bold. First, the passInfo rule is triggered within ChWS$1$ (as its left hand side is present in ChWS$1$’s sub-solution): the HOCL interpreter of ChWS$1$ triggers it, which in turn calls $S_1$ through the invoker. The result of the execution is collected, and inserted into the solution as a new molecule. Then, the RESULT: ...
molecule enables the preparePass rule, which creates the message molecule to be delivered to ChWS₁'s outgoing dependencies. In a third step, the passInfo rule is triggered, still by ChWS₁'s HOCL interpreter. When invoked, this rule actually creates a message which is sent to the multiset (as a new molecule to be delivered to ChWS₂). The molecule enables the rule, inserts it into ChWS₂'s sub-solution, which is read by the co-engine of ChWS₂. The same goes for ChWS₃, and so on.

IV. ARCHITECTURE AND IMPLEMENTATION

We target the design and the implementation of a framework which enables users to i) issue workflows using the expressiveness of Tigres, ii) transparently translate Tigres templates into HOCL code, and iii) use the HOCL-TS distributed engine in order to run them and get monitoring and logging feedback. The integration of the Tigres language and the HOCL-TS distributed execution engine led us to propose the DALHIS workflow manager, responsible for the three goals enumerated above. Therefore, the resulting workflow manager system should be able to a) offer an extensive API to allow users to manage workflows, b) allow an automatic compilation of Tigres templates to HOCL, and c) submit the HOCL workflow generated to a manager responsible for creating the HOCL-TS engine that will run the workflow.

Section IV-A gives some examples of the translation from TIGRES templates to their HOCL multiset representations. Section IV-B is dedicated to HOCL-WMS, a service that comes on top of HOCL-TS to ease the execution, monitoring and logging of HOCL workflows. Finally, Section IV-C depicts the current implementation status.

A. From Tigres to HOCL

We illustrate the translation of Tigres templates to HOCL code fragments through simple examples. Our goal is neither to show in detail the mapping between both languages, nor to make them fully explicit. We illustrate the main concepts behind them and their mapping through the study of few typical sample examples.

Sequence. Figure 7 gives the Tigres code for a workflow composed of two tasks, named T₁ and T₂, to be run as a sequence. The first two lines specify, through the Task() construct, that they will call two executables, named exec1 and exec2, respectively. Their inputs are also specified in the Task() constructs. The two tasks are then compound by the TaskArray() construct, to be passed as a parameter to the Sequence() template construct, through which this sample workflow is actually launched, with values 2 and 3.5 as inputs for the first task.

Figure 7 also gives the corresponding HOCL-TS code to the sample Tigres sequence. The resulting multiset is made of two sub-solutions containing the information of the tasks. Note that, in this current state, only sub-solution T₁ can be activated, using the invokeServ generic rule. Then, the rule defined in sub-solution T₂ expresses the control dependency between T₁ and T₂, as soon as the result from T₁ has been put inside sub-solution T₂ (by the activation of the passInfo rule described earlier). The generic rule can be matched by the co-engine and consumed by the local rule described inline in T₂. Thus creating the Param:⟨⟩ molecule, which in turn will be used to trigger the invokeServ generic rule, actually calling the second executable.

Split. Figure 8 gives the Tigres code for a workflow composed of one split pattern. The pattern mimics a workflow encoding a string in parallel by splitting it into a set of characters to

```python
from task_encode = Task("encode", FUNCTION,
    inputs = InputArray (InputValues ("IV1", [2,3.5]),
    tasks = TaskArray ("tasks", [task1, task2])
   task1 = Task ("T1", EXEC, exec1, [int, float])
   task2 = Task ("T2", EXEC, exec2, [int, string])
Sequence ("MySeq", tests, inputs)

Fig. 7: Sequence template

(a) Tigres source code

(b) HOCL translation
```

```python
from task_to_list = Task("str2List", FUNCTION,
    lambda x:x, [str])
   task_encode = Task("encode", FUNCTION,
    encode, [str])
   tasks = TaskArray ("tasks", [task_encode])
   split("encodeSplit", task_to_list, 
     InputValues(None, ["Hello Dalhis!"]),
     tasks, InputArray (None, [])

Fig. 8: Split template

(a) Tigres source code

(b) HOCL translation
```
be encoded separately. The \texttt{str2List} task returns the initial string that is given as a parameter. The Tigres’ semantic in this case is that when an input is iterable, each of its elements will be processed by a distinct instance of the split task. In our example, the \texttt{encode} task will be launched in parallel on each character of the initial string. We do not detail the encode function that is independent from the behavior of the pattern. The reader may have inferred that the second parameter is the type of the task. It can be either a \texttt{FUNCTION} meaning that the third parameter is the name of a function defined using the \texttt{def} Python construct, or \texttt{EXEC}, meaning the third parameter is the name of the underlying system command.

The HOCL code corresponding to this workflow is given by Figure 8. While the structure of the code is quite similar to the previous template, it differs by the local rule in \texttt{T2}. This rule assumes a molecule of the shape \texttt{RESULT :. . .} containing the iterable result of \texttt{T1} under the shape of a set of elements. It creates a pair of \texttt{CALL} and \texttt{PARAM} molecules to be consumed by the \texttt{invokeServ} rule, for each of the elements in the result.

\subsection*{B. The HOCL Workflow Manager System}

Once the HOCL code has been produced by the Tigres translator, it is ready to be deployed in an HOCL-TS distributed environment. The HOCL-WMS, depicted in Figure 9, is responsible for this deployment and the execution. It is a standalone framework that comes on top of the HOCL-TS architecture (see Section III-B). The HOCL-WMS is responsible not only for the deployment and execution of an HOCL-TS environment for each workflow it receives, but also for logging and monitoring prior to and during run-time.

The HOCL-WMS exposes a REST API to the user to allow classical CRUD operations on HOCL workflows as well as control over the lifecycle of the running workflows —typically users may want to be able to store and modify their workflows. On the other hand, it is able to provision the HOCL-TS environment (engines and shared multiset) needed to run the workflows. Therefore, HOCL-WMS manages both workflows and HOCL-TS engines while keeping track of their different executions. This service provides also a graphical user interface which allows user to monitor their HOCL workflows. Finally, it leverages the publish/subscribe capabilities of the HOCL-TS engine (described in III-B) to monitor and log the execution of the workflows.

Figure 9 illustrates the logical view of HOCL-WMS. In particular, it shows the interaction of its different components during the lifecycle of a workflow. Let us describe a typical HOCL-WMS usage scenario. When a workflow is submitted, it is first registered in the central database of the service so it can be retrieved later. The compilation of the HOCL workflow can also be triggered at the same moment. Once compiled, the workflow is sent to the provisioner. The provisioner is a key component because it is responsible for a) creating an HOCL-TS environment for the workflow (set of co-engines and shared space), and b) pushing the runnable representation of the HOCL workflow into it.

The HOCL-WMS leverages the presence of the central multiset to enable non-functional requirements such as logging and monitoring, introduced in Section II-C. Recall the role of the shared multiset in the HOCL-TS architecture: it contains a representation (under the shape of a running HOCL program) of the current state of the workflow, primarily allowing the co-engines to coordinate through reading and modifying it. We argue that the multiset can be naturally used to enhance HOCL-WMS with logging and monitoring. A large part of the events of workflow execution such as information passing and task termination, is reflected by a change in the description of the workflow contained in the multiset. As shown in Figure 9, publishing the state of the multiset to a queue and displaying it in a GUI comes with no deep architectural add-ons or addition of an independent logging system.

\subsection*{C. Implementation status}

As described in Section III, we are building a new framework making reuse of two available, validated tools. Thus, the remaining implementation will mostly concern the interoperability layers needed to integrate them. This interoperability can be achieved by defining the components described previously in this section.

The DALHIS workflow manager, as described in Figure 1, relies on the notion of \textit{template executor} (HOCL-TS being one of them) for Tigres and an automatic translation service from Tigres to HOCL. At the time of writing, this is achieved in Tigres by defining a common API to every template executor. The implementation of this API in the case of using HOCL-TS as a template executor contains the translation and the interaction with the HOCL-WMS service.

The HOCL-WMS, as described in Figure 9, sits on top of the HOCL-TS execution environment, which has been implemented and deployed over a large-scale platform prior to this work [5]. Nevertheless, it will integrate new components such as the compiler itself and the provisioner. The latter component is under development and will replace the current manual handling needed to provision an HOCL-TS environment. The
HOCL-WMS front-end will be implemented using the Rails framework [38]. The interoperability between the HOCL world (written in Java), and the front-end (written in Ruby) will rely on the STOMP protocol [39].

V. DISCUSSION

There has been recently a continuous effort in the scientific community to improve scientific workflow tools. Some initiatives focus on interoperability for reuse [40], [41]. They propose a workflow representation designed so as to enable portability of workflows originally written in different languages. The DALHIS workflow manager leverages only marginally this approach. In fact, to our knowledge, no initiative has actually provided a satisfying level of interoperability between workflow tools and enabled productivity and scalability at the same time without taking advantage of external mechanisms or platform optimization. The target of the DALHIS framework is more to ease the composition and execution of science workflows and allow large-scale deployment thanks to the horizontal scalability of HOCL-WMS, than to study the frameworks themselves and their interoperability.

One of the main gain expected from the DALHIS workflow manager lies in its execution model and decentralized coordination. This represents an important difference compared to conventional workflow managers where the enactment is delegated to an engine that typically relies on a centralized architecture.

Also, many potential extensions of the current design of the DALHIS workflow manager come from that the shared space, at the center of the design of the DALHIS workflow manager, contains the complete status of the workflow leaving room for implementing other non-functional concerns such as data provenance. Furthermore, based on this space, we started to study how we could enhance the DALHIS workflow manager to support checkpoint and rollback by defining additional generic rules in the HOCL description of the workflow. These rules would save the information of the workflow execution by isolating it in a particular sub-solution to be used in case of rollback.

VI. CONCLUSION AND FUTURE WORK

The DALHIS workflow manager system is a decentralized execution framework that employs a multi-layered model for integration of Tigres and HOCL-TS. In this paper, we have described the architecture and the path towards its implementation. In particular, we showed how to translate Tigres templates to HOCL code, and how the targeted HOCL-WMS will be built on top of the HOCL-TS workflow engine, so as to build the DALHIS workflow manager. We briefly emphasized how logging and monitoring can be obtained within HOCL-WMS at a low cost due to the shared-space coordination model.

The current work consolidates the interoperability ensuring that most of features of both approaches are exploited. In the same way, we plan to extend the DALHIS system to address other non-functional concerns such as data provenance and check-pointing. This time however, we contemplate the definition of such requirements to be explicitly tackled at the HOCL language level, through the definition of new HOCL rules to provide clean and elegant procedures to be conveniently used on-demand. Therefore these rules may be expressed in the specification of the workflow and not only embedded at framework level as with logging or monitoring.

ACKNOWLEDGEMENTS

This research was done in the context of the Inria DALHIS associate team, a collaboration between the Inria Myriads project-team and the LBNL's Advanced Computing for Science Department. The Tigres work was funded by the Office of Science, Office of Advanced Scientific Computing Research (ASCR) of the U.S. Department of Energy under Contract Number DE-AC02-05CH11231. The HOCL-TS work is partially funded by Inria ADT 2014 GinFlow.

REFERENCES


