Fine–Grained Workflow Interoperability in Life Sciences

Dissertation

der Mathematisch–Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von

M.Sc. Luis Javier de la Garza Treviño

aus Monterrey, Mexiko

Tübingen
2019

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	13.05.2020
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Oliver Kohlbacher
2. Berichterstatter:	Prof. Dr. Thomas Walter

Erklärung

Ich erkläre hiermit, dass ich die zur Promotion eingereichte Arbeit mit dem Titel:

Fine–Grained Workflow Interoperability in Life Sciences

selbständig verfasst, nur die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich übernommene Stellen als solche gekennzeichnet habe. Ich erkläre, dass die Richtlinien zur Sicherung guter wissenschaftlicher Praxis der Universität Tübingen (Beschluss des Senats vom 25.5.2000) beachtet wurden. Ich versichere an Eides statt, dass diese Angaben wahr sind und dass ich nichts verschwiegen habe. Mir ist bekannt, dass die falsche Abgabe einer Versicherung an Eides statt mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft wird.

Datum: Tübingen, den 13.05.2020 Unterschrift: Luis de la Garza

A mis padres.

Abstract

Recent decades have witnessed an exponential increase of available biological data due to advances in key technologies for life sciences. Specialized computing resources and scripting skills are now required to deliver results in a timely fashion: desktop computers or monolithic approaches can no longer keep pace with neither the growth of available biological data nor the complexity of analysis techniques.

Workflows offer an accessible way to counter against this trend by facilitating parallelization and distribution of computations. Given their structured and repeatable nature, workflows also provide a transparent process to satisfy strict reproducibility standards required by the scientific method.

One of the goals of our work is to assist researchers in accessing computing resources without the need for programming or scripting skills. To this effect, we created a toolset able to integrate any command line tool into workflow systems. Out of the box, our toolset supports two widely–used workflow systems, but our modular design allows for seamless additions in order to support further workflow engines.

Recognizing the importance of early and robust workflow design, we also extended a well–established, desktop–based analytics platform that contains more than two thousand tasks (each being a building block for a workflow), allows easy development of new tasks and is able to integrate external command line tools. We developed a converter plug–in that offers a user–friendly mechanism to execute workflows on distributed high–performance computing resources—an exercise that would otherwise require technical skills typically not associated with the average life scientist's profile.

Our converter extension generates virtually identical versions of the same workflows, which can then be executed on more capable computing resources. That is, not only did we leverage the capacity of distributed high–performance resources and the conveniences of a workflow engine designed for personal computers but we also circumvented computing limitations of personal computers and the steep learning curve associated with creating workflows for distributed environments. Our converter extension has immediate applications for researchers and we showcase our results by means of three use cases relevant for life scientists: structural bioinformatics, immunoinformatics and metabolomics.

Zusammenfassung

In den vergangenen Jahrzehnten führten Fortschritte in den Schlüsseltechnologien der Lebenswissenschaften zu einer exponentiellen Zunahme der zur Verfügung stehenden biologischen Daten. Um Ergebnisse zeitnah generieren zu können werden sowohl spezialisierte Rechensystem als auch Programmierfähigkeiten benötigt: Desktopcomputer oder monolithische Ansätze sind weder in der Lage mit dem Wachstum der verfügbaren biologischen Daten noch mit der Komplexität der Analysetechniken Schritt zu halten.

Workflows erlauben diesem Trend durch Parallelisierungsansätzen und verteilten Rechensystemen entgegenzuwirken. Ihre transparenten Abläufe, gegeben durch ihre klar definierten Strukturen, ebenso ihre Wiederholbarkeit, erfüllen die Standards der Reproduzierbarkeit, welche an wissenschaftliche Methoden gestellt werden.

Eines der Ziele unserer Arbeit ist es Forschern beim Bedienen von Rechensystemen zu unterstützen, ohne dass Programmierkenntnisse notwendig sind. Dafür wurde eine Sammlung von *Tools* entwickelt, welche jedes Kommandozeilenprogramm in ein Workflowsystem integrieren kann. Ohne weitere Anpassungen kann unser Programm zwei weit verbreitete Workflowsysteme unterstützen. Unser modularer Entwurf erlaubt zudem Unterstützung für weitere Workflowmaschinen hinzuzufügen.

Basierend auf der Bedeutung von frühen und robusten Workflowentwürfen, haben wir außerdem eine wohl etablierte Desktop–basierte Analyseplattform erweitert. Diese enthält über 2.000 Aufgaben, wobei jede als Baustein in einem Workflow fungiert. Die Plattform erlaubt einfache Entwicklung neuer Aufgaben und die Integration externer Kommandozeilenprogramme. In dieser Arbeit wurde ein *Plugin* zur Konvertierung entwickelt, welches nutzerfreundliche Mechanismen bereitstellt, um Workflows auf verteilten Hochleistungsrechensystemen auszuführen—eine Aufgabe, die sonst technische Kenntnisse erfordert, die gewöhnlich nicht zum Anforderungsprofil eines Lebenswissenschaftlers gehören.

Unsere Konverter–Erweiterung generiert quasi identische Versionen desselben Workflows, welche im Anschluss auf leistungsfähigen Berechnungsressourcen ausgeführt werden können. Infolgedessen werden nicht nur die Möglichkeiten von verteilten hochperformanten Rechensystemen sowie die Bequemlichkeit eines für Desktopcomputer entwickelte Workflowsystems ausgenutzt, sondern zusätzlich werden Berechnungsbeschränkungen von Desktopcomputern und die steile Lernkurve, die mit dem Workflowentwurf auf verteilten Systemen verbunden ist, umgangen. Unser Konverter–Plugin hat sofortige Anwendung für Forscher. Wir zeigen dies in drei für die Lebenswissenschaften relevanten Anwendungsbeispielen: Strukturelle Bioinformatik, Immuninformatik, und Metabolomik.

Acknowledgments

Progress is often oversimplified as a linear sequence of refinements. Hans Lippershey improved upon *reading stones* to craft quality spectacles, and his patent application for *“an instrument for seeing things far away as if they were nearby”* would soon inspire Galileo Galilei—allegedly. This thesis and the work it documents are certainly no different. I am ever so grateful to Prof. Oliver Kohlbacher for giving me the opportunity to rightfully refer to him as *mein Doktorvater*. His patience, knowledge and guidance helped me see things far away as if they were nearby.

More accurately, though, progress is a directed acyclic graph of events[1], a *workflow* of sorts. There are countless people I am thankful to, but a single page would not be enough to mention them, for I would have to thank gods and titans (thanks for the [BBQ] fire, Prometheus), bus and train drivers showing up sober for work, long–gone inventors of hand axes, and so it goes. In the spirit of fairness, I should mention those whose omission would be outright insulting.

I would like to thank my parents and sister. They make me feel as if I had won the galactic lottery by being born into a loving, caring family that always knew how to foster curiosity during my formative years. Thanks to Alexander Fillbrunn, Peter Ohl, Bernd Wiswedel, Thorsten Meinl and Stephan Aiche for showing me KNIME's guts. Kitty Vargas, Ákos Balaskó, István Márton and Zoltán Farkas deserve my gratitude for their constant help and pointers that shaped me into the university's resident expert on WS–PGRADE/gUSE—a trophy that I never truly deserved.

Thanks to colleagues and friends who made my doctoral experience the best I could have wished for: from the cat–lovers to the dog–lovers; from the Mensa naysayers to the Mensa club[2]; from the Voronoi cartographers and Nexus listeners to those that helped me write those words that, upon reading them, would prompt Mark Twain to remind us of that Californian student in Heidelberg who would rather decline two drinks than one German adjective; from those who asked too often about my writing to the ones that never asked, lest I asked them about the status of their own projects; from the sister in arms who showed me the ropes of molecular docking to the brother in arms who showed me that iconic line from *őszödi beszéd*, a line that opened doors that, more often than not, I would not know how to close afterwards. I am even thankful to those who have—jokingly or seriously—demanded to be acknowledged.

[1]Citation needed. Also, whether progress can be modeled as a tree or a forest is left as an exercise for the reader.
[2]Sadly, I am referring to the university's cafeteria, not to the so–called intelligent people club.

General Remarks

In accordance with the standard scientific protocol, the personal pronoun ***we*** will be used throughout this document to indicate the reader and the writer, or my scientific collaborators and myself.

Contents

Chapter 1

Introduction

Motivation

The ability to independently replicate reported results is crucial to the scientific method. This not only serves as a self–check mechanism to separate spurious, incorrect claims from facts, but it also paves the way for scientists to build upon the findings of other researchers. Modern scientific studies have become highly specialized and complex, often requiring resources or trained personnel not available to every laboratory, so fully independent replication is often difficult to attain. Nevertheless, the advancement of scientific endeavors requires the means and methods to at least reproduce reported findings.

In the current scientific language, *reproducibility* and *replicability* are two intertwined but different concepts. A scientific experiment is replicated when a separate group of scientists reaches the same findings and conclusions after acquiring data in an independent manner, using the same instruments and methods[1] . Given that not every group of scientific investigators has access to the same resources (e.g., not all laboratories have facilities to collect neuronal data from mice), replication frequently becomes a taxing effort. There is, however, a minimum standard that any experiment could attain in order to deliver reproducible results.

Reproducibility takes into account this uneven access to resources across laboratories and expects publishers to make data and analytical methods fully available[2]. Independent researchers could access such data and reproduce reported results. Furthermore, in the context of computational science, reproducibility implies that experiments can be designed to keep a detailed track of the actions taken to collect and analyze data. Given the widespread use of software and computers in scientific fields nowadays, this bare minimum standard could be easily met by any scientific experiment.

Even though the guidelines on design of reproducible studies are technically simple, there have been multiple reports of a *reproducibility crisis* in major scientific journals and media outlets with a wide audience[3–9], suggesting that the problem does not lie in technical limitations but rather in either a lack of competence and training or in the attitude and behavior of scientists. Researchers could satisfy the bare minimum of reproducibility by using tools such as workflows.

Workflows offer structured, abstract recipes that help their users build a series of steps towards more complex and specific analyses in an organized way. Each of the building blocks of a workflow, often called *job* or *node*, is a parametrized, specific, simple action that receives inputs and, after some calculations, produces some output. Typically, each of these building blocks performs a domain–independent task (e.g., download a file, add a column to a data table). The organized, collective execution of these building blocks is, in contrast, seen as a domain–specific task (e.g., produce a list of compounds that bind to a specific protein).

With the availability of biological *big data*, the need to promptly process considerable amounts of data has become a pressing matter[10,11]. Bioinformatic calculations are therefore turning more complex and require more computing resources to complete in a timely manner. Simple personal computers or monolithic approaches to solve scientific questions in bioinformatics can no longer keep pace with the growth of available biological data, the complexity of computations, and the need for faster results. Structuring solutions to scientific questions using workflows is not only best practice, but it also helps researchers to expedite reliable generation of repeatable results.

Modeling a scientific experiment as a structured, organized set of cooperating tasks has several benefits. Not only intermediate results can be stored for further analysis or troubleshooting but also bottlenecks can be easily identified. Furthermore, the domain–independent building blocks can be reused in other pipelines, cutting down development times. The ability to perform sections of workflows with different values of a given setting in parallel (i.e., a *parameter sweep*) is a feature often sought after[12]. Parallel execution of independent workflow branches is simplified if the computations of a complex analysis are structured as a workflow.

While the capabilities of workflow engines might differ, they all—at the very least—allow users to design, execute, and monitor workflows. Whether execution happens on the user's desktop computer or on a remote computing cluster is an aspect that varies among implementations. However, the best case scenario for users is to be able to design workflows in a visual, intuitive way, while being able to seamlessly access powerful resources, such as those found on grids or clouds, to execute their workflows.

There are several points to consider when choosing a workflow engine. In spite of the great value of a user–friendly, responsive graphical user interface (GUI) during the construction phase of a new pipeline, not all workflow engines feature an uncomplicated design tool suite. Complementary to this, although access to high–performance computing (HPC) resources tends to diminish workflow execution times, not all engines allow for a smooth transition between designing workflows and executing them on remote, more capable computing resources. Extensibility and scaling are important additional criteria to consider. Since each workflow engine was initially developed thinking of a specific user community, each engine offers a different set of features and might lack certain capabilities. Clearly, there is a gap that can be bridged by combining different workflow engines in order to incorporate their features and circumvent their shortcomings.

The objective of our work is to offer friendly, intuitive, workflow design along with open access to HPC resources for workflow execution by combining the features of different engines.

Main Challenges

Compatibility between workflow engines cannot always be guaranteed because there is no widely–accepted language to represent all aspects of workflows. Representation of facets such as the topology of a workflow—available in the workflow languages we have studied—is not sufficient to fully convey details required to execute a workflow.

Interoperability across workflow management systems varies between *coarse–grained* and *fine–grained*. The former refers to approaches in which a complete workflow is invoked from within a single task of another workflow, the latter is accomplished when an automatic full conversion takes place and all involved engines are able to *natively* execute their own versions of the same workflow. Fine–grained approaches provide a more precise control over workflows, hence promoting optimization (e.g., by executing several independent tasks in parallel).

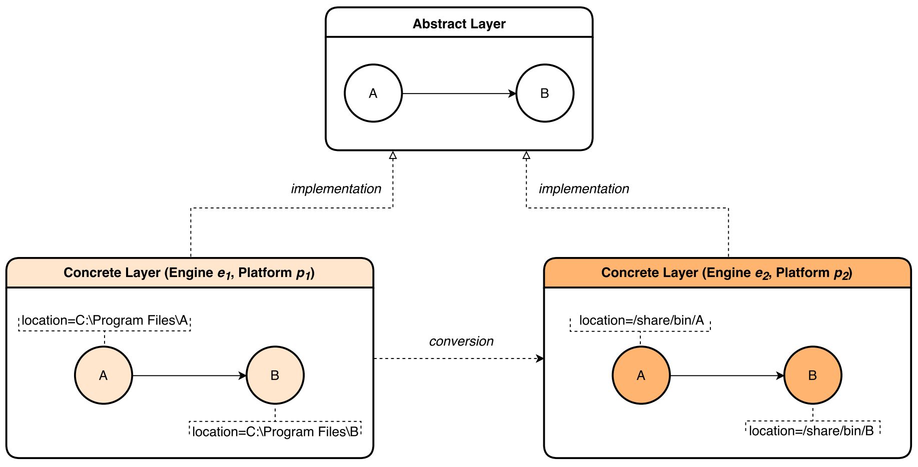

Abstract Layer
A
B
implementation
implementation
Concrete Layer (Engine e_1, Platform p_1)
location=C:\Program Files\A
A
B
location=C:\Program Files\B
conversion
Concrete Layer (Engine e_2, Platform p_2)
location=/share/bin/A
A
B
location=/share/bin/B

Figure 1.1: Schematic view of a workflow conversion. The same *abstract* workflow has different implementations across engines, each being a *concrete*. Conversion of full workflows happens across workflow engines. Properly converted workflows have the same *abstract* as their source. Figure adapted with permission from[13].

Conversion of workflows across engines to provide fine–grained interoperability is not a trivial task due to the fact that each engine implements similar features in a different way (see Figure 1.1). The first challenge in combining different engines is to understand how the building blocks of workflows (i.e., each of the independent tasks) are represented and executed in each of them. A correct approach must convert not only the topology of workflows but also each of the individual tasks. Only after conversion of these has been elucidated, converting complete workflows can be undertaken. There is no approach that works for all workflow engines, making this a weighty challenge.

An interesting question arises when features of workflow engines are combined: is it possible that a given engine properly implements or emulates missing features from another one? Furthermore, some workflow engines make no real separation between the bare workflow topology and the resources it needs to be executed, while some other engines separate these two perspectives in several steps throughout the design phase. This, of course, must also be taken into account when different engines are combined. Since this is a very specific implementation detail of each engine—often not thoroughly documented—acquiring a solid understanding of how workflows are represented is a demanding endeavor. The disparity of representations poses a challenge to scientists who desire to reuse workflows. Ideally, a scientist would be able to design and test a workflow only once and execute it on any other engine after some transformation.

Significant work has been made to achieve interoperability across workflow engines. The *SHaring Interoperable Workflows for large–scale scientific simulations on Available distributed computing interfaces* (SHIWA) Project allows users to run previously existing workflows from different platforms on the SHIWA Simulation Platform[14]. However, due to privacy concerns, scientists might give a second thought to execute workflows and store sensitive data on the SHIWA Simulation Platform. Likewise, Tavaxy, focusing on genome comparison and sequence analysis, was developed to provide interoperability between Taverna and Galaxy workflows[15,16]. Similarly, the work of Grunzke et al.[17] brings the Konstanz Information Miner (KNIME) Analytics Platform closer to more powerful computing resources by integrating it with the Uniform Interface to Computing Resources (UNICORE) middleware. These approaches achieve only coarse–grained interoperability, whereas our main interest lies in reaching fine–grained interoperability.

Contributions

The KNIME Analytics Platform offers over two thousand modules to build workflows and facilitates both development of new nodes and integration of external command line tools, yet it might be limited in computing power (i.e., it is designed to run on personal computers). The Web Services Parallel Grid Runtime and Developer Environment Portal (WS–PGRADE), together with the Grid and Cloud User Support Environment (gUSE), in contrast, compose a framework that taps into several distributed computing interfaces (DCI), easing inclusion of arbitrary HPC resources, but its workflow editor poses a steep learning curve to its user base.

Considering these aspects, we developed a plug–in for the KNIME Analytics Platform, *KNIME2Grid*, that allows users to export KNIME workflows to WS–PGRADE/gUSE, where they can be executed on any supported HPC. Engines such as Galaxy and gUSE utilize *proxies* to represent the individual jobs that comprise a workflow. Typically, these proxies do not contain an executable file, rather, a suitable command line pointing to the location of a binary. These kind of nodes reference an external executable existing outside the context of a workflow engine. The KNIME Analytics Platform, on the other hand, relies mostly on native nodes (*KNIME Nodes*). These are Java classes whose code is hosted by the process executing the KNIME Analytics Platform: a KNIME Node can be executed only inside a running instance of the KNIME Analytics Platform. This poses quite a challenge for the proper conversion of KNIME workflows.

Contrasting to the KNIME Analytics Platform, WS–PGRADE does not maintain a proper application repository from which end users can simply select an appropriate version of a required dependency. The workflow configuration process in WS–PGRADE requires users to provide, for each workflow node, a script to invoke a remote executable, along all required command line parameters. End users do not have the benefit of simply *drag and dropping* configured visual representations of jobs. In order to alleviate this intricate procedure, we devised an extension that provides users with the ability to manage a basic *application database* in WS–PGRADE. Using our add–ons in conjunction, users are able to configure their converted WS–PGRADE workflows using the KNIME Analytics Platform by selecting a desired version of an executable from a list. To demonstrate the capabilities of KNIME2Grid and our application database extension, we present the following use cases in the field of computational biology:

- Structural bioinformatics: conversion of a molecular docking workflow.
- Immunoinformatics: conversion of a population–based vaccine design pipeline.
- Metabolomics: conversion of a biomarker discovery pipeline.

Furthermore, in order to extend the capabilities of workflow management systems, we developed a toolset able to convert Common Tool Descriptors (CTD), which are platform–independent tool representations, to other formats. Out of the box, our toolset is able to generate descriptors for Galaxy and in the Common Workflow Language (CWL) format, but our modular design allows for a smooth extension to support additional formats.

Outline

The *Background* section formerly defines what a workflow is and also introduces some selected workflow engines we found to be commonly used in the bioinformatics field. We close this section by briefly discussing workflow languages, which are important in the context of workflow interoperability.

Chapters 3 and 4 present the detailed account of the development of an open–source suite of software solutions designed to provide fine–grained workflow interoperability across platforms. The subsections range from the introduction of a platform–independent job representation and its applications to the description of the features we implemented into existing workflow engines. Each of these subsections presents work that satisfies the needs of the scientific community. Each chapter contains a discussion comparing our work with other available technologies.

Our closing remarks and an outlook are presented on Chapter 5.

Chapter 2

Background

2.1 Workflows

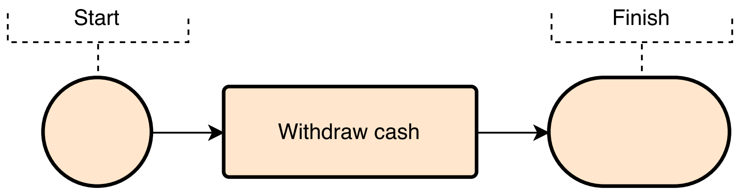
The abstraction of processes was originally developed to increase productivity and decrease costs in the workplace by focusing on the optimization of routine work activities[18]. Early literature introduced so–called *process charts* in order to visualize operations with the purpose of improving them[19]. The importance of enforcing the standards derived from these diagrams was soon recognized. Process charts also support the notion that any given detail of a procedure is more or less unaffected by every other detail, allowing adopters to effectively identify profitable adjustments, preventing *inventing downward* (i.e., detrimental changes) and stimulating cumulative inventions of permanent value[19].

Before the widespread use of information technologies these processes were exclusively carried out by humans, who, in turn, operated machinery or simple tools to assist them in the executions of these tasks[18]. Nowadays, and since the use of computing technologies in the workplace, these processes have been partially or fully automated by computer software able to execute tasks and oversee the enforcement of a defined set of rules. The nature of these processes can be categorized in three areas[20]:

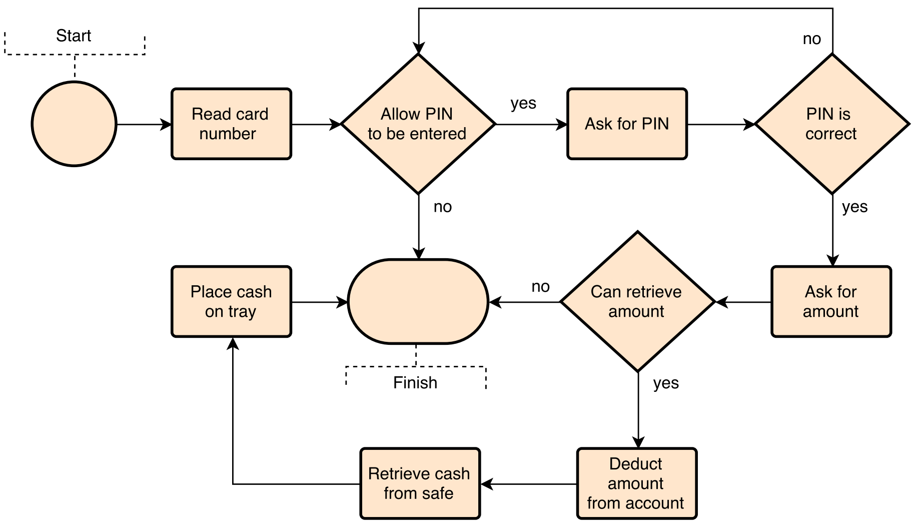
- Material processes: activities in which physical components are transformed and assembled into products, such as classical manufacturing and transportation of goods.
- Information processes: the current ubiquitous usage of computers and the high availability of information at almost all times blurs the boundaries of this category of processes. These processes are related to automated tasks performed by software whose purpose is to transform, create, manage and provide information, such as payroll management software and search engines.
- Business processes: these combine both material and information processes in order to satisfy specific needs. Online shopping is an example of a business process that involves

both information processes (e.g., customers ordering items through their web browsers) and material processes (e.g., shipping orders).

The first applications of process flows, or *workflows*, were found in the context of office and manufacturing tasks[19]. Workflows represent business processes and offer a well–defined paradigm to execute complex, domain–specific tasks in a structured, repeatable fashion, making them ideal not only for industry but also for scientific applications. The execution order of each individual step in a process is determined by the workflow's structure (i.e., its topology). Each individual step, often referred to as *job*, can take inputs, produce outputs and be configured using parameters.



(a) A workflow seen as a black box. From a user's perspective, withdrawing cash is a simple activity.



(b) Detailed view of the same operation using *flowchart* elements. Each task executes a very specific action that could be reused in other workflows. Some tasks have been either left out or grouped together for the sake of brevity.

Figure 2.1: A cash withdrawal operation represented using two different levels of detail.

A single workflow addresses a domain–specific problem and could be treated as a *black box* that takes some input data, has some parameters and produces some output. On the other hand, the individual jobs that constitute a workflow are often domain–independent. Figure 2.1 depicts the relationship between a workflow, seen as a whole, and its comprising building blocks.

An advantage of structuring complex activities into workflows is the reusability of highly–specialized building blocks. In the example provided in Figure 2.1b, manufacturers could, for instance, reuse components represented by the task labeled *Read card number* in other machines. Splitting the work into simple tasks means that several teams could work in parallel to design, support and improve these.

2.1.1 Workflow Layers

Processes represented with workflows contain three different dimensions or layers, i.e., *case*, *process* and *resource dimension*[21]. These can be summarized as follows[22,23]:

- The process dimension, also referred to as the *abstract* layer, deals with the application domain. Technical details such as architecture, platform, libraries, implementation and programming language, are not present in this dimension. Only the structure and purpose of the workflow are available in this dimension. This layer is, so to speak, the *foundation* of a workflow.
- The resource dimension, often called *concrete* layer, encompasses all technical aspects required to execute the desired process. Implementation details such as architecture and quantity of processors that a task requires, software dependencies and parameters, are described here.
- The case dimension refers to the execution instances of a workflow.

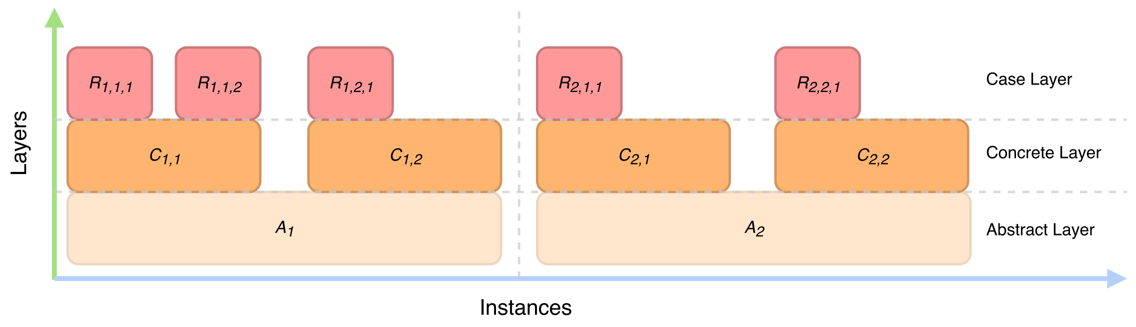


Figure 2.2: Schematic view of the workflow layers.

As Figure 2.2 illustrates, the case layer rests upon the concrete layer, which in turn is built upon the abstract layer. Each abstract can contain several concrete representations. Every execution of a concrete is represented in the case dimension. Going across the *Instances* axis represents independence of the depicted items. Conversely, the *Layers* axis indicates dependency on lower layers. For instance, the abstract A_1 is independent from A_2. Similarly, the concrete representations $C_{1,1}$, $C_{1,2}$ and $C_{2,1}$ are independent from each other. However, the execution $R_{2,2,1}$ depends on the concrete $C_{2,2}$, which in turns depends on the abstract A_2.

For the sake of clarity and brevity, we prefer the use of the *abstract/concrete* terminology throughout our work.

2.1.2 Formal Representation

In his doctoral dissertation, *Kommunikation mit Automaten*[1], Carl Adam Petri introduced the foundations of a theory of communication to formally and precisely describe the several phenomena that occur during the exchange and transformation of information in a system[24]. Petri immediately recognized a direct application of his novel theoretical framework: design and programming of information systems[24]. His ideas quickly found their way into scholarly journals and were picked up both by the scientific and the industry communities[25–27].

After further refinement, the concepts presented in Petri's doctoral dissertation were condensed into what we know today as *Petri Nets*, offering a concise, abstract, formal model of information flows[28]. Petri Nets contain the necessary mathematical rigorousness to precisely describe and study distributed, concurrent, asynchronous, non–deterministic, parallel, and stochastic information processing systems, allowing scientists to formulate algebraic equations and other models to represent the states and transitions of a system[25].

a Petri Net is a 5–tuple, $PN = (P, T, A, W, S_0)$, where:	
$P = \{p_1, p_2, ..., p_n\}$	is a finite set of places,
$T = \{t_1, t_2, ..., t_m\}$	is a finite set of transitions,
$A \subseteq (P \times T) \cup (T \times P)$	is a set of arcs,
$W: A \rightarrow \{1, 2, 3, ...\}$	is the arc weight function,
$S_0: P \rightarrow \{0, 1, 2, ...\}$	is the initial state,
$P \cap T = \emptyset$, $P \cup T \neq \emptyset$	
$N = (P, T, A, W)$	represents the Petri Net *structure* (i.e., a Petri Net without a given initial state)

Definition 2.1: Formal definition of a Petri Net. Adapted with permission from[25]. Copyright 1989 IEEE.

Petri Nets are formally defined as directed, weighted, bipartite graphs containing two kinds of nodes, *places* and *transitions*, and an initial state, often referred to as the *initial marking*, S_0. Each edge, or arc, has an assigned weight and either originates from a place and is directed towards a transition or stems from a transition and ends on a place (this restriction is what makes Petri Nets bipartite graphs)[25].

Each place models the set of conditions that must be fulfilled for a transition to occur or *fire*. On the other hand, transitions represent actions or events that modify the state of the

[1]Communication with Automata

system and have a number of *input* and *output places*. Places and transitions thus model the pre– and postconditions of events, respectively. In order for a transition to actually occur, all of its preceding places must have satisfied all of their represented conditions. Conversely, when a transition has fired and is completed, a postcondition is satisfied, effecting a change in the state of subsequent places to which this transition is connected to.

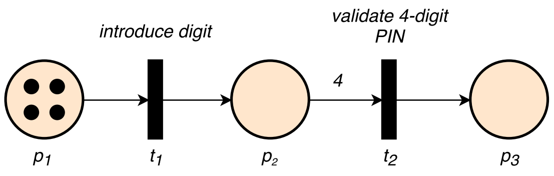
Each state assigns a non–negative integer number to each place. When a state has assigned a number i to a place p, then it is said that p has been *marked* with i tokens. States are represented by n–dimensional vectors, where n is the total number of places. The j–th component of a given state S, denoted by $S(p_j)$, is the number of tokens in place p_j. The existence of a token in a place signifies that the condition associated with the place has been fulfilled[25].

In order to depict changes in the modeled system, Petri Nets introduce state changes (i.e., the assignment of tokens) according to the following transition rules[25]:

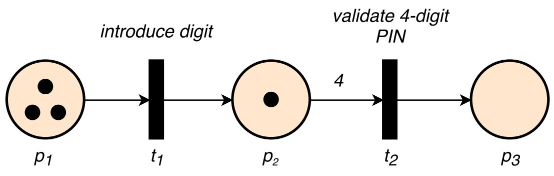
- If each input place p of a transition t contains at least $W(p,t)$ tokens, where $W(p,t)$ is the weight of the arc that originates in p and is directed towards t, then t is *enabled*.
- A transition that has been enabled may or may not fire. This depends on whether the event represented by the transition actually occurs.
- Whenever an enabled transition t is fired, $W(p,t)$ tokens are removed from each input place p of t.
- Whenever an enabled transition t is fired, $W(t,p)$ tokens are added to each output place p of t, where $W(t,p)$ is the weight of the arc that originates in t and ends in p.

Transitions lacking output places are called *sink transitions*, while transitions without any input places are referred to as *source transitions*. Source transitions are inherently enabled. Sink transitions, while able to consume tokens, do not generate any[25].

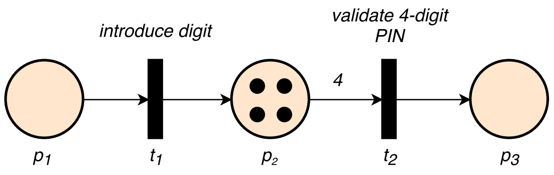
Petri Nets also offer visual portrayals able to express complex systems where information is passed among elements, providing the required graphical symbols to represent the concurrent and dynamic states of a system[25]. Graphically, places are usually displayed as circles, while transitions are drawn as bars or rectangles. Tokens are represented as dots inside places. Arcs, as it is usually done in depictions of weighted directed graphs, are drawn as arrows annotated with their corresponding weight. These start from a place and are directed to a transition or vice versa[25,27,28]. Figure 2.3 illustrates a Petri Net that models the capture and validation of a 4–digit personal identification number (PIN).



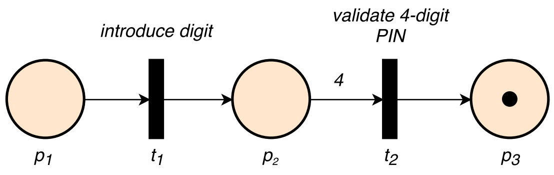
(a) The Petri Net in its initial state. The transition t_1 is enabled because its only input place, p_1, has at least as many tokens as the weight of the arc (i.e., 1) connecting them.



(b) The Petri Net after t_1 has fired once. p_1 lost one token: the weight of the arc connecting it to t_1 is one. Similarly, p_2 has been marked with one token as a result of the unitary weight of the arc connecting it to t_1, which is still enabled.



(c) The Petri Net after all four digits have been introduced. t_2 is now enabled because there are at least four tokens on its precondition place, p_2. Since p_1 contains no more tokens, t_1 is no longer enabled.



(d) The Petri Net after transition t_2 has fired and consumed four tokens from p_2. p_3 has been marked with a token.

Figure 2.3: Simplified capture and validation of a 4–digit PIN modeled using a Petri Net and applying transition rules. Unitary arc weights are omitted. Transitions have been annotated with a brief description of the modeled events for the sake of explanation.

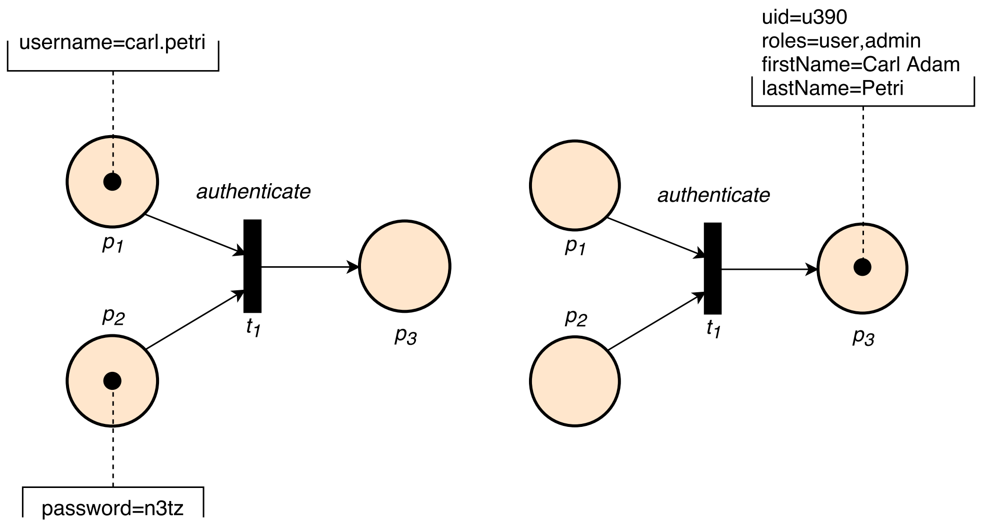
High–Level Petri Nets

Due to their rigorousness, Petri Nets were promptly proposed as a unifying modeling tool for information systems, these ranging from computer hardware to distributed databases[26–29]. However, adopters quickly noticed that modeling processes using pure, classical Petri Nets often lead to complex, unmaintainable networks without an appropriate level of detail, rendering analysis a tedious and cumbersome feat[30–34].

Systems modeled by Petri Nets often depend on information about the represented entities comprising a system. Tokens typically depict material or human resources, yet, in their purest form, they lack the elements to succinctly describe the components they model. Moreover, the notion of time, an important facet of dynamic information systems, is not formally encoded in classical Petri Nets. Representation of processes requiring either individualized treatment of tokens or a clear definition of how properties and relations change across time is therefore not possible using classical Petri Nets.

There have been several proposed extensions to Petri Nets[30–32,35–38]. Literature refers to non–classical, non–pure Petri Nets as *high–level Petri Nets*. Genrich and Lautenbach[38] introduced the first well–known high–level Petri Nets, *predicate/transition Petri Nets* (PrT–nets), adding expressiveness by extending places to make them able to change properties and relations between the modeled individuals. Transitions in PrT–nets were also modified accordingly to feature *templates* or *schemes* for the purpose of modifying token assignment rules to better represent the processes carried modeled systems.

Building upon these concepts, tokens have also been added *color*[32,36,37,39]. This does not refer to the chromatic phenomena, rather, to information contained in tokens to differentiate them. Thus, transitions are not only able to fire depending on their *firing colors*, but they also determine the color of produced tokens based on the color of their input[34], as depicted in Figure 2.4.

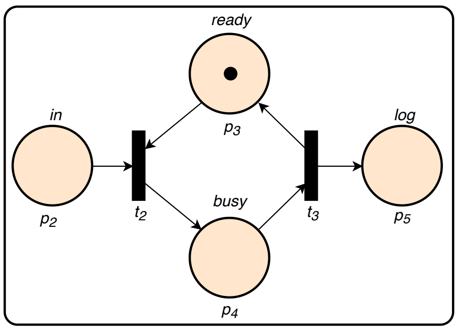


(a) Tokens possess colors modeling information entered by a user, e.g., *password*. Transition t_1 is enabled but has not yet fired.

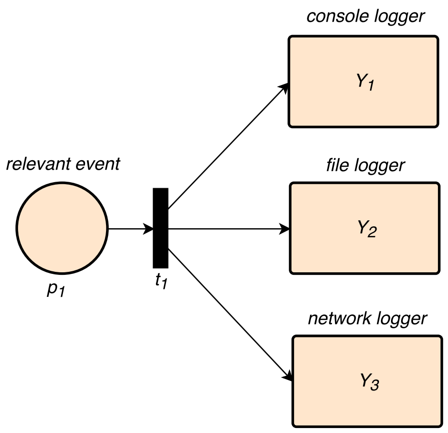
(b) Transition t_1 has already fired and, based on the color of its input tokens, it has produced a token containing a different color.

Figure 2.4: A high–level Petri Net showing tokens with color. Information contained in tokens is represented by keys and values using the $key = value$ format.

Nevertheless, even after utilizing concepts of time and color, models easily grow in size and complexity. As introduced by Peterson[28], Van der Aalst[34], the usage of hierarchies greatly increases the expressiveness of high–level Petri Nets. This is achieved by allowing aggregation of places, transitions, and possibly other systems (see Figure 2.5).



(a) Aggregation of places and transitions using hierarchies to form a basic, generic logger, Y.



(b) A system using three loggers. Depiction is kept more compact by reusing system Y.

Figure 2.5: Using hierarchies in high–level Petri Nets. Figures adapted with permission from[34]. Copyright 1994 Elsevier B.V.

Since high–level Petri Nets are an augmentation of classical Petri Nets, they inherently offer the same precise mathematical foundation and are suited to a big number of analysis methods[34]. High–level Petri Nets provide a rigorous language to formally model complex flows of information in a compact, manageable manner. Nonetheless, most software tools able to design and orchestrate the execution of workflows do not explicitly utilize Petri Nets. One can only speculate about the nature of this design decision, but it is reasonable to believe that this departure from Petri Nets obeys the fact that such tools aim to facilitate their usage and strive to widen their user base.

2.1.3 Common Representation

Workflows are commonly represented as unweighted directed acyclic graphs (DAG)[40–44]. Each vertex represents a system from a high–level Petri Net, typically containing a place together with its pre– and postconditions, i.e., the preceding and following transitions from the high–level Petri Net from which they originate[45]. Vertices are labeled and contain unique identifiers.

Edges are not weighed, and, akin to Petri Nets, they determine the logical sequence to follow: an edge between two vertices represents the *channeling* of an output from a task into another.

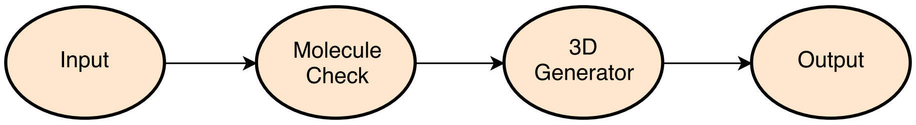


Figure 2.6: An unweighted DAG representing the abstract layer of a workflow. After validating input data, a three–dimensional representation of the input molecule is generated.

Tasks modeled by vertices are executed once all of their inputs can be resolved. Vertices are commonly referred to as *nodes*, *jobs* or *tasks*. In this work we will use these terms interchangeably.

Figure 2.6 shows an example of a workflow using this high–level representation featuring a further level of simplification compared to the high–level Petri Net presented in Figure 2.5. This workflow is composed of four tasks and three edges. The task *Input* has no predecessors and will be the first one to be executed. In comparison, the task labeled *3D Generator* depends on the completion of *Molecule Check*, which in turn depends on the completion of *Input*.

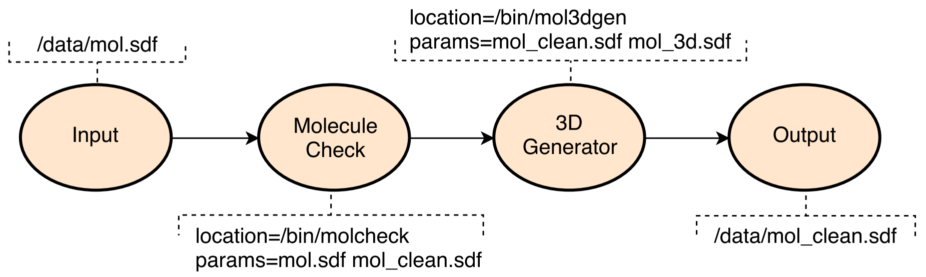


Figure 2.7: A possible concrete layer of the workflow depicted in Figure 2.6. Information required to execute tasks composing this workflow is included in this layer.

In contrast to Figure 2.6, Figure 2.7 shows a possible concrete representation of the presented abstract layer, in which each vertex has been annotated with the information needed to actually execute the corresponding tasks. While this varies across platforms and architectures, recall that abstracts are constrained exclusively to the application domain and are thus independent of the underlying technical dependencies. On the concrete layer, inputs and outputs are typically files, e.g., the *3D Generator* task receives an input file from its predecessor, *Molecule Check*, and generates an output file that will be channeled to the *Output* task.

2.2 Workflow Management Systems

Workflow management systems, or workflow engines, are software tools designed to create, manage and execute workflows. The execution order is driven by a computer representation of the workflow structure. Running a single workflow might take up to several months, depending on its complexity, parameter settings and amount of processed data[46].

While there is no fixed recipe to implement workflow engines, these typically require an array of information technologies and communications infrastructure, and are able to operate in environments ranging from a single–user computer to remotely distributed architectures, yet they exhibit certain common characteristics—providing a basis for integration and interoperability capabilities between different implementations, namely[46]:

- Build–time functions to define workflows and their constituting tasks: during the build–-time phase, a real–world business or information process is transcribed into a formal, computer–processable definition by the use of one or more analysis, modeling and system definition techniques. The product of these functions is a workflow model, typically including both the abstract and concrete layers.
- Runtime control features handling orchestration of workflow execution in an operational environment: these are in charge of sequencing the various activities to be handled as part of each execution. During the runtime phase, the workflow model is interpreted by software responsible for creating and controlling operational instances of the workflow, as well as scheduling the several tasks comprising it.
- Runtime interactions with users and external software: tasks within a workflow could depend upon further input from users or external applications (e.g., fill out an electronic form). These interactions are required to process the various individual tasks.

Reproducibility is a desirable feature in workflow management systems. Parameters, inputs, and outputs should be permanently recorded: analyses could then be precisely and independently repeated. In other words, each independent run of a workflow (i.e., each instance on the case layer) should be recorded in order to guarantee reproducibility.

2.2.1 The KNIME Analytics Platform

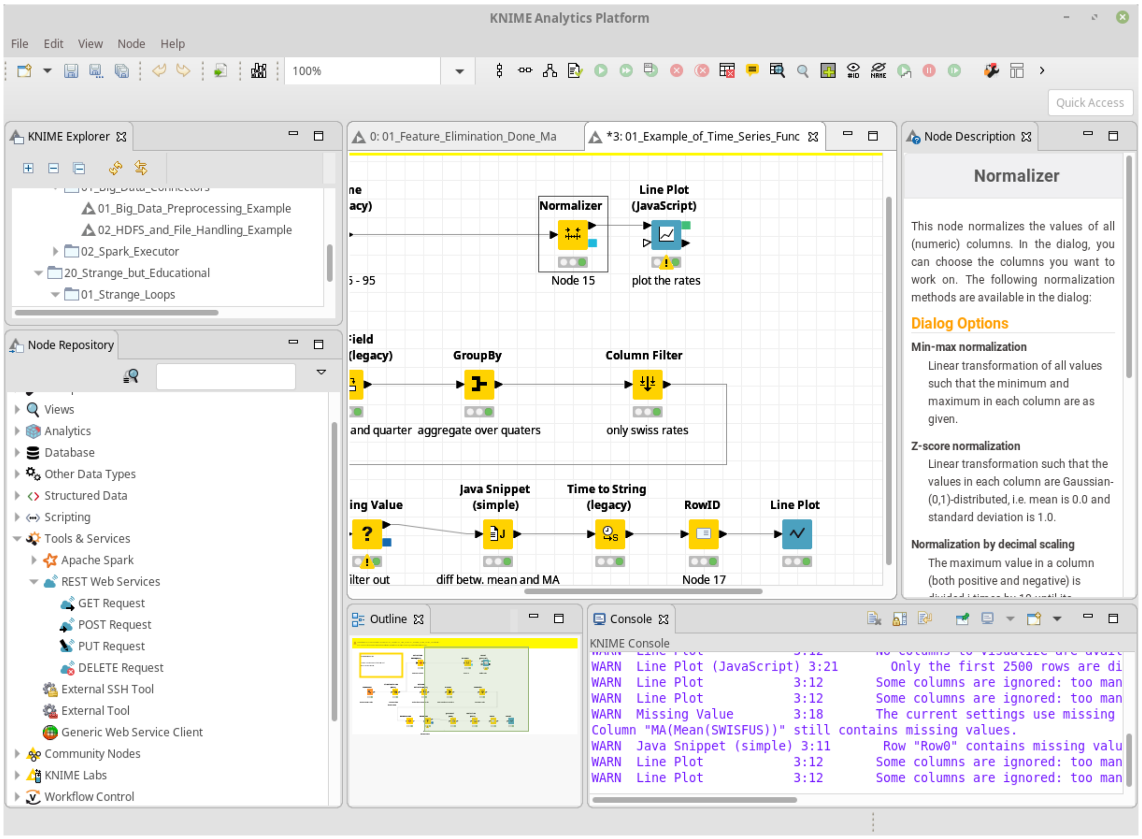
The KNIME Analytics Platform is a desktop–based workflow engine featuring a powerful and accessible GUI with hundreds of ready–to–run examples[40]. Users can choose among more than two thousand KNIME Nodes that serve as the building blocks of a workflow. Nodes can be added to the KNIME Analytics Platform by either downloading ready–to–use nodes or by developing new ones using a well–documented Application Programming Interface (API) that is accessible to inexperienced developers available at KNIME's website[2].

The KNIME community is very active, meets at periodic KNIME–organized events and supplies nodes under the KNIME Community Contributions program. The domain of these contributions, which are easily obtainable from `https://www.knime.com/community`, range from image processing to chemo– and bioinformatics.

Workflows executed on the KNIME Analytics Platform are limited to run on the same personal computer on which it has been installed, rendering it unsuitable for tasks with high–-memory or high–performance requirements. Two royalty–based variants to execute workflows on distributed HPC resources are available: KNIME Cluster Executor and KNIME Server[3].

[2]`https://www.knime.com/developer/documentation/wizard`

[3]Further information available at `https://www.knime.org/knime-cluster-executor` and `https://www.knime.com/knime-software/knime-server`, respectively.



Normalizer

This node normalizes the values of all (numeric) columns. In the dialog, you can choose the columns you want to work on. The following normalization methods are available in the dialog:

Dialog Options

Min-max normalization
Linear transformation of all values such that the minimum and maximum in each column are as given.

Z-score normalization
Linear transformation such that the values in each column are Gaussian-(0,1)-distributed, i.e. mean is 0.0 and standard deviation is 1.0.

Normalization by decimal scaling
The maximum value in a column (both positive and negative) is

KNIME Console

```
WARN  Line Plot (JavaScript) 3:21       Only the first 2500 rows are di
WARN  Line Plot              3:12      Some columns are ignored: too man
WARN  Line Plot              3:12      Some columns are ignored: too man
WARN  Missing Value          3:18      The current settings use missing
Column "MA(Mean(SWISFUS))" still contains missing values.
WARN  Java Snippet (simple) 3:11       Row "Row0" contains missing valu
WARN  Line Plot              3:12      Some columns are ignored: too man
WARN  Line Plot              3:12      Some columns are ignored: too man
```

Figure 2.8: Screenshot of the KNIME Analytics Platform. Through its friendly GUI, users drag and drop nodes from the *Node Repository* (bottom left) and use them to build and execute workflows using the *Workflow Manager* (top middle). A summary of the currently selected node is displayed on the *Node Description* section (top right). Figure adapted with permission from[40]. Copyright 2009 ACM.

Users build workflows by dragging and dropping visual representations of nodes. Inputs and outputs of each of these nodes, commonly referred to as *ports*, can be connected by using the intuitive GUI. Ports are assigned data types, thus enforcing compatibility between input and output ports: only ports with compatible types can be connected.

Users obtain immediate feedback upon running a workflow. It is possible to resume a faulty, canceled execution after issues have been resolved: the KNIME Analytics Platform correctly determines which tasks require re–execution. Output data can be visualized in the KNIME Analytics Platform by right–clicking a node and selecting the desired action on the displayed pop–up menu.

2.2.2 WS–PGRADE and gUSE

Together, WS–PGRADE and gUSE make up a web–based engine that taps into several DCI and HPC infrastructures[44]. WS–PGRADE, offered as a *Liferay portal*, acts as the front end and offers *portlets* for workflow creation, execution and monitoring, as well as for supporting tasks (e.g., usage statistics). Interaction with gUSE is realized through its remote API—based on Web Services Description Language (WSDL) requests, allowing gUSE to interact with WS–PGRADE and other workflow engines[44,47]. Its layered architecture enables administrators to distribute a setup across resources: WS–PGRADE can be installed on a dedicated front end server, while gUSE's components can be deployed on shared resources.

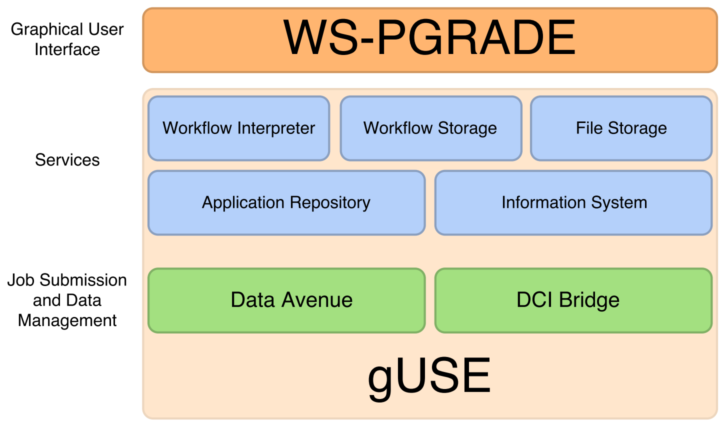
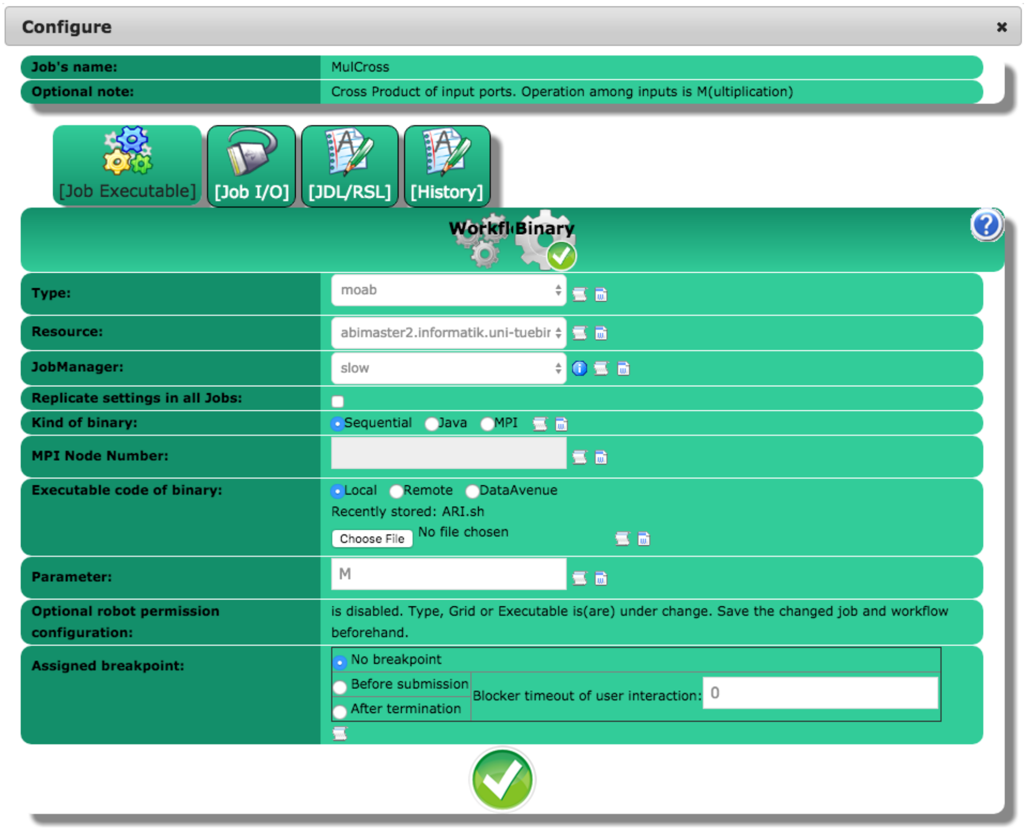


Figure 2.9: WS–PGRADE/gUSE's architecture. The *Services* layer handles tasks such as storage and execution. The *Job Submission and Data Management* layer contains the *DCI Bridge*, which is responsible to access DCIs. Adapted with permission from[48]. Copyright 2014 MTA SZTAKI LPDS, Budapest.

WS–PGRADE splits creation of abstracts and concretes. Abstracts are created via a *Java WebStart* application, *Graph Editor*: users create nodes from scratch, providing details such as number of input and output ports. Due to an update in security policies denying automatic execution of Java WebStart applications[49], latest versions ship with a browser–based Graph Editor.

Successfully building and saving an abstract is followed by creation of concretes and their subsequent node–per–node configuration, a step where users provide a script that will be submitted to the respective DCI, along with a command line providing any associated parameters. The potential complexity of WS–PGRADE workflows is reflected in the many available fields in the concrete configuration portlets, as depicted in Figure 2.10.



Job's name:	MulCross
Optional note:	Cross Product of input ports. Operation among inputs is M(ultiplication)

Type:	moab
Resource:	abimaster2.informatik.uni-tuebir
JobManager:	slow
Replicate settings in all Jobs:	☐
Kind of binary:	[✓ Sequential] Java MPI
MPI Node Number:	
Executable code of binary:	[✓ Local] Remote DataAvenue Recently stored: ARI.sh Choose File No file chosen
Parameter:	M
Optional robot permission configuration:	is disabled. Type, Grid or Executable is(are) under change. Save the changed job and workflow beforehand.
Assigned breakpoint:	[✓ No breakpoint] Before submission After termination Blocker timeout of user interaction: 0

Figure 2.10: Concrete node configuration in WS–PGRADE. Users must specify details such as the resource on which a task will execute and the command line. A script to be submitted on the corresponding DCI must also be provided. Adapted with permission from[50]. Copyright 2015 MTA SZTAKI LPDS, Budapest.

WS–PGRADE offers no real–time feedback regarding the state of a workflow: users must *poll* the execution status. Email notifications for changes in the execution status of a workflow are available, however, these do not cover status updates of single tasks.

There is a steep learning curve associated with WS–PGRADE, and this might intimidate users lacking technical experience. Nevertheless, gUSE's main strength lies in that it offers a uniform platform to access several DCIs. Additionally, development of application–domain specific portlets to facilitate creation and execution of workflows is supported.

2.2.3 The Galaxy Project

Galaxy is a web–based workflow management system that seeks to address the disruptive changes biomedical research has been subjected to after the introduction of high–throughput data production technologies, featuring several pre–installed tools for data–intensive biomedical research. By providing a common platform to share and publish results, Galaxy simplifies collaboration via transparent data analyses that can be inspected at every level of detail or even replicated and extended[41].

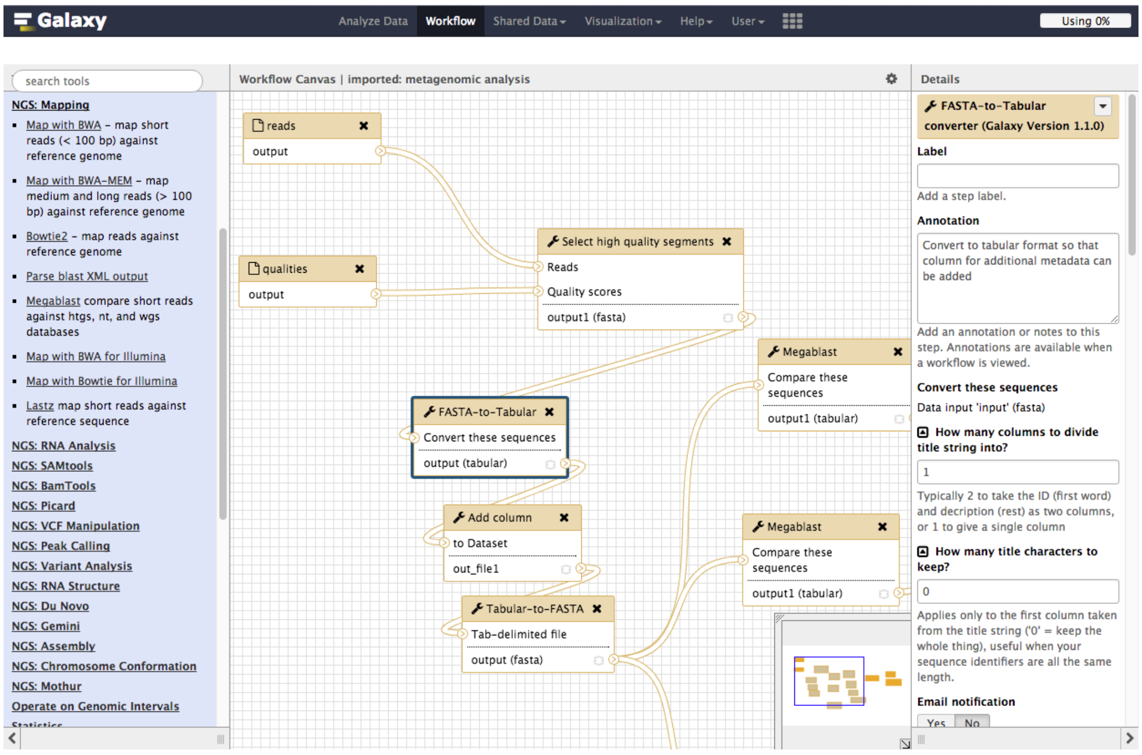
Through its intuitive browser–based GUI, shown in Figure 2.11, users can seamlessly upload data from their workstations, access data from public databases, choose among analysis toolsets, set workflow parameters, execute workflows, etc. Galaxy also features a user–friendly workflow editor, the *Workflow Canvas*, where users create workflows by dragging and dropping elements[41].

Installing additional nodes is achieved by using the *Tool Shed*'s accessible web–based GUI. There are, at the time of writing, more than 6,700 valid tools available in the public Tool Shed[51].

Galaxy offers a free–to–use public instance, the Galaxy Public Server[4], which provides significant computing power and disk space, making it feasible to analyze large datasets. It features common analysis tools and data sources, and supports hundreds of thousands task executions for thousands of users per month[41].

It is also possible to install Galaxy on–site by using the Galaxy Software Framework: administrators and tool developers maintain their own Galaxy instances and integrate external tools into Galaxy. The Galaxy Software Framework is an open–source, Python–based, highly–customizable application that can be installed on *Unix–like* systems as a Galaxy instance. It executes jobs locally by default, but it can be configured to submit jobs on DCIs. However, setting up a shared file system between the resources hosting Galaxy instances and computing resources is required. Furthermore, all jobs are submitted under a single user account, *galaxy_user*, complicating administrative tasks such as individual quota enforcement and usage statistics[41,52].

[4]Available at `https://usegalaxy.org`



NGS: Mapping

- Map with BWA – map short reads (< 100 bp) against reference genome
- Map with BWA-MEM – map medium and long reads (> 100 bp) against reference genome
- Bowtie2 – map reads against reference genome
- Parse blast XML output
- Megablast compare short reads against htgs, nt, and wgs databases
- Map with BWA for Illumina
- Map with Bowtie for Illumina
- Lastz map short reads against reference sequence

NGS: RNA Analysis
NGS: SAMtools
NGS: BamTools
NGS: Picard
NGS: VCF Manipulation
NGS: Peak Calling
NGS: Variant Analysis
NGS: RNA Structure
NGS: Du Novo
NGS: Gemini
NGS: Assembly
NGS: Chromosome Conformation
NGS: Mothur
Operate on Genomic Intervals

FASTA-to-Tabular converter (Galaxy Version 1.1.0)

Label

Add a step label.

Annotation

Convert to tabular format so that column for additional metadata can be added

Add an annotation or notes to this step. Annotations are available when a workflow is viewed.

Convert these sequences

Data input 'input' (fasta)

How many columns to divide title string into?

1

Typically 2 to take the ID (first word) and decription (rest) as two columns, or 1 to give a single column

How many title characters to keep?

0

Applies only to the first column taken from the title string ('0' = keep the whole thing), useful when your sequence identifiers are all the same length.

Email notification

Yes [✓ No]

Figure 2.11: Screenshot of the Galaxy Public Server showing an imported workflow. The left column displays a list of all available tools in the Galaxy Public Server. The middle section contains the *Workflow Canvas*, where users can drag and drop tools to build workflows. The rightmost section allows users to provide parameters for the selected tool. Screenshot by author. Copyright 2005–2015 Pennsylvania State University.

2.3 Workflow Languages

Workflow engines rely on different underlying technologies and focus on particular capabilities, resulting in *incompatible islands* of process automation: it is not expected that workflows built in certain engine could be executed on a different one without some sort of preprocessing. Ensuring certain degree of interoperability is critical for the success of large–scale workflow management, where distinct application domains, platforms and engines are involved[46,53].

We have introduced how high–level Petri Nets provide a rigorous workflow representation, but they still lack expressiveness to store designs for their execution on workflow management systems. Workflow languages were conceived to represent real processes for their execution by engines, addressing key issues such as task coordination requirements, specification of tasks and their execution states, and workflow execution requirements to ensure application–domain standards[54,55]. Even though they contain elements such as control structures, computational completeness is not an area of interest for workflow languages[55].

2.3.1 Interoperable Workflow Intermediate Representation

The SHIWA Project, an initiative whose focus is to develop workflow coarse–grained interoperability, utilizes the Interoperable Workflow Intermediate Representation (IWIR) as its workflow language. IWIR was designed to create a standard that sufficiently describes most existing workflow constructs using a lower level of abstraction, enabling portability of abstract layers and decoupling itself from concretes. It is specified using Extensible Markup Language (XML) documents containing graph–based structures that represent data flows, parallel and sequential controls[14,56].

IWIR Bundles were introduced by Plankensteiner et al.[57] to describe concretes. They contain an IWIR definition of a workflow bundled with a set of files containing binary dependencies and templates in the Job Submission Description Language (JSDL) describing computational tasks. Integration developers can extend workflow engines by creating *IWIR front–ends* able to convert IWIR definitions—included in IWIR Bundles—into compatible representations.

2.3.2 Common Workflow Language

CWL is a platform–independent standard that has its origins in *Make* and other similar build automation tools that determine the execution order based on dependencies between tasks. It is the joint effort of individuals, organizations and vendors collaborating to facilitate sharing of data–intensive analysis pipelines among scientists. It observes the principles of the *OpenStand* movement, a group that encourages the development of global, market–driven standards for the benefit of humanity[58,59].

CWL documents are written in a mixture of JavaScript Object Notation (JSON) and *YAML Ain't Markup Language* (YAML), and they are able to describe large–scale workflows in HPC environments where tasks are commonly scheduled in parallel across many computing resources[58].

Chapter 3

Conversion of Workflow Nodes

3.1 Introduction

The computational tasks comprising concrete layers of workflows often result in invocations of command line tools. Therefore, conversion of individual workflow nodes must consider not only the disparities between the origin and target node representations but also the underlying architectural differences among the involved platforms, if any.

Files where inputs, parameters and outputs of a task are formally described greatly simplify their automatic conversion to other formats. Galaxy, CWL documents and CTDs offer well–structured file formats able to represent all aspects of workflow nodes (with varying degrees of expressiveness). A file parser could take any such descriptor file as an input, extract relevant information, and generate an output in a different format.

There is an intrinsic value in the usage and conversion of tool description files: software libraries able to provide and interact with such descriptors are easier to integrate into workflow engines, thus paving the way for increasing interoperability.

3.2 Methods

3.2.1 Definition of Workflow Nodes and Their Conversion

Files describing workflow nodes contain information easily pliable into *key–value pairs*, regardless of the format (e.g., *name*, *output_location* could be keys that map to values representing the name and output location of a certain tool, respectively). Workflow nodes can thus be represented using key–value pairs, which programming languages commonly implement as *dictionary* data structures.

Nodes can be formally represented as a function $n\colon k \to v$, where k, v represent sets holding keys and values, respectively. This, akin to colors in high–level Petri Nets, provides a basis to store and manipulate information in a more concise fashion.

```
<ITEM name="lig_chain" type="string"
      description="Ligand chain name" />
-----------------------------------------------

lig_chain:
    type: string
    label: Ligand chain name
    inputBinding:
        prefix: -lig_chain
-----------------------------------------------

<param name="lig_chain" type="text"
       label="Ligand chain name" />
```

$name = lig\_chain$
$type = string$
$desc = Ligand\ chain\ name$
$bind\_to = -lig\_chain$

Figure 3.1: Representation of workflow nodes using dictionaries. The left side shows (top to bottom) possible depictions of the same input parameter using a CTD, a CWL document and a Galaxy representation, respectively. The right side displays a dictionary containing the same information using a "$key = value$" notation.

The first step of our proposed conversion procedure is to parse a descriptor file with the purpose of extracting information into a dictionary, as briefly depicted in Figure 3.1. Since node representation formats might contain platform–specific information not pertinent for the conversion (e.g., inputs in Galaxy allow a `size` attribute indicating the length of a text field on which a value can be entered in Galaxy's GUI), only certain attributes are extracted. What follows is the generation of an output file using a different format. Nowadays, there are software libraries available with the purpose of parsing and generating structured files, e.g., XML parsers. Algorithm 1 details the procedure to convert a workflow node.

Algorithm 1 Single workflow node conversion. An input representation, i_{f_s} (in the source format f_s), is first parsed into an in–memory data structure, p_{f_s}, dependent on its format. Relevant information from p_{f_s} is then extracted into a dictionary representing the node to convert, i.e., a function, $n\colon k \to v$ (k, v being the key and value sets, respectively). This dictionary, which is independent of the source and target formats, is then serialized to a file, o_{f_t}, using the target format f_t. Implementations for the $Parse$ and $Write$ routines are commonly available as software libraries, e.g., XML parsers.

```
 1: procedure CONVERTNODE($i_{f_s}, o_{f_t}$)    ▷ source, target files, each having its own format
 2:     $p_{f_s} \leftarrow Parse(i_{f_s})$    ▷ parse source file
 3:     $n \leftarrow EmptyDictionary()$
 4:     for all relevant $(k, v)$ in $p_{f_s}$ do    ▷ visit all relevant entries in parsed input
 5:         $n[k] \leftarrow From_{f_s}(v)$    ▷ translate property from the source format
 6:     end for
 7:     for all $(k, v)$ in $n$ do
 8:         $Write_{f_t}(k, To_{f_t}(v), o_{f_t})$    ▷ write transformed property to the target format
 9:     end for
10: end procedure
```

3.2.2 Main Challenges

Algorithmically speaking, conversion of workflow nodes is clearly not a complicated process. Proper translation of elements is trivial if they represent a basic type, e.g., integers, floats or strings: a simple mapping relating data types among formats would be sufficient. This translation process is summarized by the $From_{f_s}$ and To_{f_t} routines introduced in Algorithm 1. However, workflow management systems represent single tasks using custom formats, each having its own strengths and caveats. Although all formats describe workflow nodes, some are capable to convey more information than others through the use of special sections.

A successful conversion of workflow nodes requires a thorough understanding of each involved format: even though these have overlapping goals, each was designed to fulfill specific needs, resulting in discrepancies that must be properly dealt with. This also means that any implementation must consider an adequate direction of the conversion in order to avoid loss of information throughout the process.

Representation of Galaxy Nodes

Galaxy natively interacts with external command line tools via *ToolConfig* files. These are XML documents that describe the inputs, outputs and parameters of tools and act as proxies between Galaxy instances and command line tools. ToolConfigs, together with the required dependencies to execute the represented tools, can be imported into a public or a local Tool Shed[51,60].

ToolConfigs are not fully platform–independent: including Python–based snippets (i.e., *Cheetah* templates) for text generation is allowed[60,61]. When a section containing Cheetah code is invoked, it is first compiled to Python code and is then executed by the Galaxy engine. Basic error handling is also supported via *stdout/stderr* scanning. Additionally, it is possible to include *macros* in order to import content from external XML documents in order to reuse sections across several ToolConfig files.

Even though ToolConfig files can contain Python–like code, users without software development experience can also easily generate them[60]. Refer to Appendix B.0.4 for a sample ToolConfig file.

Representation of Nodes using CWL

CWL is not a workflow management system proper, rather, it is a set of specifications developed to represent data–intensive workflows across a variety of platforms supporting the CWL standard. It offers a single format to define both complete workflows and their underlying tasks. To achieve this, CWL documents are categorized by document classes: workflows are specified using the *Workflow* document class, while command line tools are assigned the *CommandLineTool* document class (see Appendix B.0.1). The standard provides a CWL interpreter as part of the reference implementation[1].

Contrasting to Galaxy's ToolConfigs, CWL representation of workflow nodes is inherently platform–independent. Nevertheless, the CWL standard includes frequently–available features present in modern workflow engines, e.g., usage of environment variables, support for Docker containers, capture of *stdout/stderr*, parameter manipulation via JavaScript expressions, file staging and definition of nested workflows[58].

Command Line Tool Representation Using CTDs

We briefly introduced CTDs as platform–independent tool representations in Chapter 1. CTDs are XML documents that were originally used inside bioinformatics libraries to avoid using long command line arguments when executing complex pipelines. Tools featuring native support to generate and interpret CTD files (e.g., SeqAn[62], OpenMS[63] and CADDSuite[64]) are said to be *CTD–enabled*. Appendix B.0.2 briefly exemplifies usage of CTD files.

CTDs provide a concise format that limits itself to the representation of inputs, outputs, and parameters that a task requires to be executed. While ToolConfigs and CWL documents also contain these details, they also include additional information not supported by CTDs. One of the main motivations—and advantages—of CTDs is to provide a platform–independent representation of workflow nodes easy to work with: the simpler a format is kept, the smoother it will be to integrate with third–party command line tools and workflow engines.

It is possible to couple arbitrary command line tools with CTDs via *CTDopts*, a Python module available on the Anaconda Cloud that enables tools to interact with CTDs[65].

CTDopts provides a data structure, *CTDModel*, containing the model of a tool. General information about a tool (e.g., name, version), parameters, inputs and outputs can be added to CTDModels, which CTDopts can not only serialize into CTDs but also load them from CTD files[65]. Refer to Appendix B.0.3 for sample CTDopts usage.

Conversion of Individual Parameters

Input and output files in CTDs are declared as parameters in the `<PARAMETERS>` section, receiving a specific parameter type, i.e., `input-file` and `output-file`, respectively. However, both CWL and ToolConfigs dedicate separate sections for inputs and outputs. File handling in ToolConfigs and CWL documents drastically differs from how files are declared in CTDs:

- ToolConfigs handle output files separately by using the `<outputs>` section, while input files are declared as normal parameters under the `<inputs>` section (see Listing B.11).

[1]Available at `https://github.com/common-workflow-language/cwltool`

- Definition of an output file in CWL requires an input of type `string`, representing the file path. Additionally, an output of type `File` needs to be present and it must be *bound* to its corresponding file path. CWL treats the output file path as an input parameter, while the file itself is seen as a true output. Refer to Listing B.1 for an example.

Files certainly receive different handling across workflow node representations: a simple mapping is not sufficient to successfully convert this kind of elements. These sections must be identified and proper dedicated routines for their handling must be implemented.

Direction of the Conversion

The `<command>` section in ToolConfigs shown in Listing B.11 contains two flow control sections. It would indeed be possible to program full sub–routines in this section, or even invoke other programs, elevating the complexity of ToolConfigs. Although limited in scope compared to the expressiveness achievable in ToolConfig files, the permitted JavaScript fragments in CWL documents offer support to create complex expressions to manipulate input parameters.

Loss of information could take place if the source format of the conversion, i.e., f_s, is more expressive than the target format, i.e., f_t. For instance, conversion of ToolConfigs and CWL documents into CTDs cannot be guaranteed for all scenarios: CTD files were not designed to provide support for scripting languages. On the other hand, given the scope of the CTD format, a full conversion from CTD files into other formats can be, in general, guaranteed.

3.3 Results

3.3.1 CTDConverter

We developed *CTDConverter*, an easily extensible Python framework able to parse CTD files and convert them to other formats. Extending CTDConverter to add supplementary output formats requires significantly less effort than writing a stand–alone script able to parse a CTD and produce a file in the appropriate format.

Common features such as validation of input CTD files against a schema, blacklisting parameters, using fixed values for specific parameters and processing several CTD files in a single invocation are offered as core features.

We designed CTDConverter to be extensible. To support an additional format, tool developers provide a Python script containing a module defining parameters and options without having to modify other components. Each extension module has access to the core functionalities, cutting down development time. Furthermore, CTDConverter can easily be integrated with other external systems, e.g., scripts in continuous integration build systems.

CTDConverter first scans all provided command line parameters and options, then makes use of CTDopts to parse all input CTD files and convert them into their corresponding CTD-Models. Once a CTDModel has been obtained[2], each conversion script is responsible to iterate through the contained data structures to effectively convert them and generate files in the desired output format. Tool developers extending CTDConverter interact with CTDModels without the need to write code to neither explicitly parse nor generate CTDs.

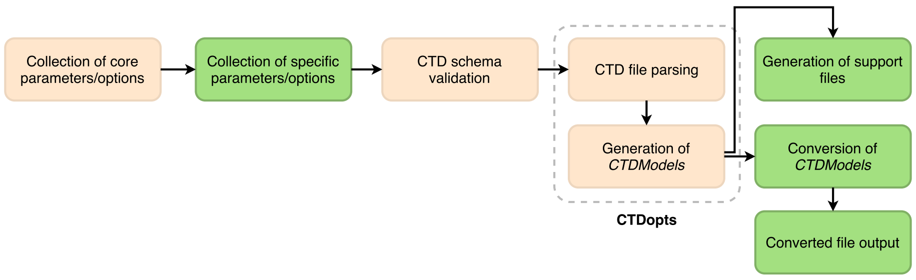


Figure 3.2: Schematic view of the overall process to convert CTD documents using the CTD-Converter framework. The conversion relies on CTDopts to parse input CTD files and to generate CTDModels, in–memory representations of CTD files. These models are then used by the specific converter modules to output files in the desired format. Boxes shown in beige represent common features shared across all supported converters, while green–colored boxes represent specific actions that are determined by each of the converter modules.

Design

CTDConverter has a modular design that allows developers to easily extend its functionality by adding independent, supplementary converter modules. Developers extending CTDConverter are required to implement three functions that will be invoked from the entry point. Each module is able to specify the preferred extension for generated files, register its own parameters and options, and convert a CTDModel without parsing or validating any input CTD file. Minimal changes need to be done in the entry point module. Figures 3.2 and 3.3 depict CTDConverter's overall design and its detailed component diagram, respectively.

[2]CTDModels implement the introduced formal representation of workflow nodes, i.e., n, as detailed in Algorithm 1.

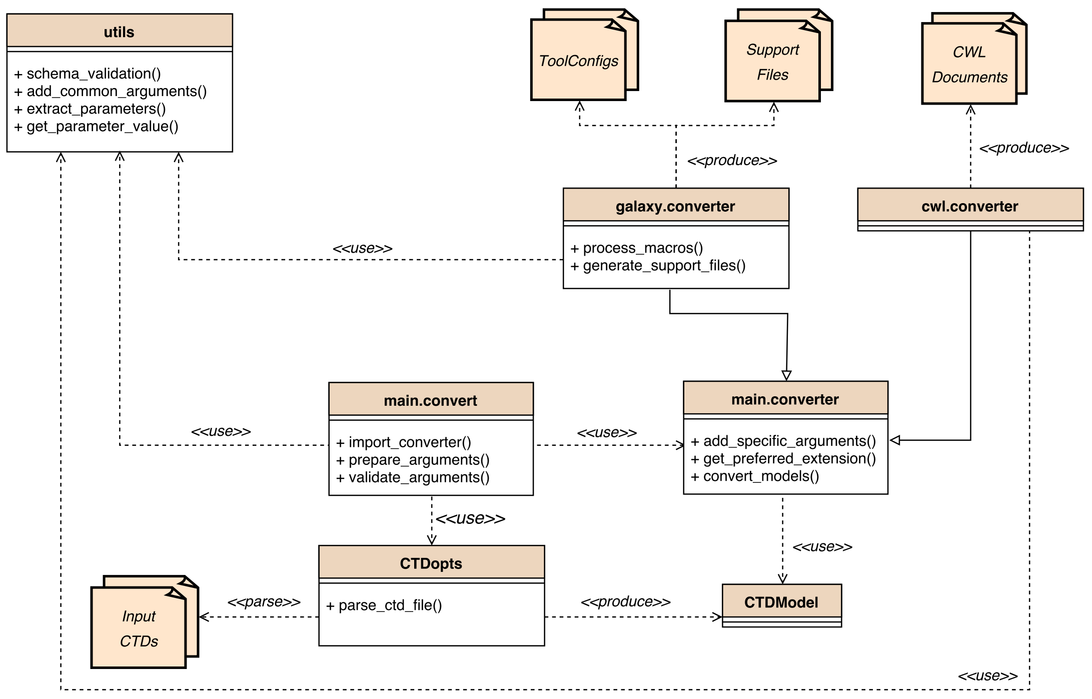


Figure 3.3: CTDConverter component diagram. Additional formats can be added by providing an extra Python module implementing the functions in *main.converter*. No intrinsic knowledge of the implementation details of the specific converters is included in the entry point, *main.convert*. Specific converters, here shown as the *galaxy.converter* and *cwl.converter* modules, receive in–memory representations of CTD documents (i.e., *CTDModel*), which they can later utilize to generate output files, and have access to utility functions via the *utils* module.

Common Features

All converter modules can utilize the core CTDConverter features, which can be controlled via command line parameters, namely:

- Converting several CTD files at once: instructs CTDConverter to convert all of the provided CTD files. CTDConverter will automatically generate an adequate file name for each of the converted CTDModels. This feature is specially useful in build systems, where a build step could generate a set of CTD files and, in a separate step, they can be converted using a single command.

- Blacklisting parameters: workflow developers might want to hide certain flags or commands from final users. Some parameters present in command line tools are not to be exposed in workflow engines (e.g., a typical `-help` parameter to print usage of a tool to *stdout*).

- Validation of input CTD files against a schema: CTDopts requires correct CTD files in order to parse them to produce CTDModels. In order to guarantee the validity of CTD files, a schema document can be provided—CTDConverter will stop its execution if any of the provided CTD files is invalid, providing information about incorrect inputs.

- *Hard–coding* parameters: inexperienced end users might provide inadequate values to options such as `-threads` or `-debug`, which could lead to errors difficult to track down. CTDConverter allows to set fixed values for this kind of parameters.

Galaxy–Specific Features

CTDConverter converts arbitrary CTD files into proper ToolConfigs that can be integrated into any Galaxy Tool Shed, offering the following features to facilitate tool integration:

- Generation of macros files: CTDConverter is able to include macro definitions from any of the provided input macros. It also ships with a sample XML including default macro definitions, as shown in Listing B.13. Users can customize or completely override the included XML macros file and adapt it to their needs.

- Generation of other support files: Galaxy instances utilize ancillary files to define data types (i.e., *datatypes_conf.xml*), include tools and assign them a category for display purposes (i.e., *tool_conf.xml*). Even though these support files are incompatible with CTD files, CTDConverter can generate them to fully integrate command line tools in a Galaxy Tool Shed.

3.3.2 Availability

CTDConverter is distributed under the MIT license and can be obtained directly from its repository located at `https://github.com/WorkflowConversion/CTDConverter`.

3.3.3 Use Cases

Integration With the Biochemical Algorithms Library

We integrated CTDConverter in the nightly builds of the Biochemical Algorithms Library (BALL). At the time of writing, BALL offers more than 60 command line tools designed to perform the most common tasks in the field of computer–aided drug design, such as molecular docking, retrieval and processing of protein data bank (PDB) files, and generation of conformational isomers[64]. These tools, distributed as the Computer Aided Drug Design Suite (CADDSuite), are already CTD–enabled and were specifically designed to be integrated into workflows, an example of this is *BALLaxy*[66], which integrates BALL with Galaxy.

Previously, the generation of ToolConfigs was performed by the BALL codebase. We modified the existing build scripts to utilize CTDConverter. Since the CADDSuite tools can produce their own CTDs, we were able to seamlessly *plug in* CTDConverter and generate the required files for BALLaxy, i.e., ToolConfig files, *tool_conf.xml* and *datatypes_conf.xml*. The content of the generated Galaxy files before and after integrating CTDConverter did not change.

Integration With OpenMS

Featuring more than 90 command line tools, OpenMS caters to a broad audience of developers, mass spectrometry laboratories, single users as well as consortia with access to HPC infrastructures. It offers versatile, CTD–enabled command line tools for file handling, signal (pre–)processing, visualization, database searching, quantification, as well as peptide identification, which can be used individually or as part of complex pipelines for data analysis[63].

Developers and enthusiasts of both OpenMS and Galaxy have been manually generating ToolConfig files to integrate OpenMS into Galaxy's Tool Shed. We worked closely with the team whose responsibility was to maintain the OpenMS ToolConfig files. After several iterations, the result was a stable CTDConverter version that, without breaking compatibility with previous versions, delivered ToolConfigs for the Galaxy/OpenMS community. Given the needed parameters and input files, CTDConverter is now used to automate a task that used to be a manual, tedious effort. Our efforts translated into configurable features so other communities could benefit from our work.

CTDConverter as a Conversion Framework

In its current version at the time of writing (2.1), CTDConverter presumes the existence of CTD files describing workflow nodes. In order to benefit from CTDConverter, integrators of non–CTD–enabled must somehow generate CTDs. Suites lacking usage of any structured format to describe and document their tools could undergo a refactoring to become CTD–enabled: integration with CTDConverter would be automatic.

Large toolsets that utilize other tool description formats could also benefit from CTD-Converter without any modification of the codebase. For instance, the latest release of the European Molecular Biology Open Software Suite (EMBOSS) features more than 250 tools that utilize AJAX Command Definition (ACD) files[67,68], which could be parsed into CTDModels for their conversion into CTDs. Because of its modular design, CTDConverter offers a development platform for integrators: it could be easily be extended to allow for additional input formats.

3.3.4 Sample Usage

To exemplify CTDConverter, we manually created a CTD for a non–interactive command line tool, shown in Listing 3.1.

Listing 3.1: CTD file for *wget*, a non–CTD–enabled internet downloader tool. Only selected parameters are shown for brevity.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tool version="1.17.1" name="wget" category="Data" ctdVersion="1.7">
  <description>Wget - The non-interactive network downloader.</description>
  <executableName>wget</executableName>

  <PARAMETERS version="1.7">
      <NODE name="wget" description="Parameters for wget">
          <ITEM name="outputDocument" type="output-file" required="true" value=""
                description="Location of the output file" />
          <ITEM name="noCertificate" type="bool" required="false" value="false"
                description="Skip checking of certificate" />
          <ITEM name="noVerbose" type="bool" required="false" value="true"
                description="Print important information" />
          <ITEM name="url" type="string" required="true" value="" position="0"
                description="URL to download" />
      </NODE>
  </PARAMETERS>

  <cli>
    <clielement optionIdentifier="--output-document">
      <mapping referenceName="outputDocument" />
    </clielement>
    <clielement optionIdentifier="--no-check-certificate">
      <mapping referenceName="noCertificate" />
    </clielement>
    <clielement optionIdentifier="--no-verbose">
      <mapping referenceName="noVerbose" />
    </clielement>
    <!-- positional argument, identifier is left blank on purpose -->
    <clielement optionIdentifier="">
      <mapping referenceName="url"/>
    </clielement>
  </cli>
</tool>
```

We then fed this CTD (stored under *wget.ctd*) into CTDConverter to generate a ToolConfig and a CWL document as depicted in Listing 3.2.

Listing 3.2: Commands required to convert the file shown in Listing 3.1 into CWL and ToolConfig representations, respectively. The used *macros.xml* file is included in CTDConverter and is displayed under Listing B.13.

```
$ python convert.py cwl --input wget.ctd --output wget.cwl
$ python convert.py galaxy --input wget.ctd --output wget.xml --macros galaxy/macros.xml
```

These invocations produced the output files shown in the next two pages.

Listing 3.3: Sample CWL output for the CTD presented in Listing 3.1.

```
#!/usr/bin/env cwl-runner
# This CWL file was automatically generated using CTDConverter.
# Visit https://github.com/WorkflowConversion/CTDConverter for more information.

baseCommand: wget
class: CommandLineTool
cwlVersion: v1.0
label: Wget - The non-interactive network downloader.
inputs:
- id: param_outputDocument_filename
  doc: Filename for outputDocument output file
  inputBinding:
    prefix: --output-document
  label: Filename for outputDocument output file
  type: string
- id: param_noCertificate
  default: 'False'
  doc: Skip checking of certificate
  inputBinding:
    prefix: --no-check-certificate
  label: Skip checking of certificate
  type: ['null', boolean]
- id: param_noVerbose
  default: 'True'
  doc: Print important information
  inputBinding:
    prefix: --no-verbose
  label: Print important information
  type: ['null', boolean]
- id: param_url
  doc: URL to download
  inputBinding:
    position: 0
  label: URL to download
  type: string
outputs:
- id: param_outputDocument
  doc: Location of the output file
  label: Location of the output file
  outputBinding:
    glob: $(inputs.param_outputDocument_filename)
  type: File
```

Listing 3.4: Sample ToolConfig output for the CTD presented in Listing 3.1. Contents of the referenced macros file are shown in Listing B.13.

```
<?xml version=’1.0’ encoding=’UTF-8’?>
<!--This is a configuration file for the integration of a tools into Galaxy
    (https://galaxyproject.org/).
    This file was automatically generated using CTDConverter.-->
<!--Proposed Tool Section: [Data]-->
<tool id="wget" name="wget" version="1.17.1">
  <description>Wget - The non-interactive network downloader.</description>
  <macros>
    <token name="@EXECUTABLE@">wget</token>
    <import>macros.xml</import>
  </macros>
  <expand macro="stdio"/>
  <expand macro="requirements"/>
  <command>wget
#if $param_outputDocument:
  --output-document $param_outputDocument
#end if
#if $param_noCertificate:
  --no-check-certificate
#end if
#if $param_noVerbose:
  --no-verbose
#end if
#if $param_url:
       "$param_url"
#end if
</command>
  <inputs>
    <param name="param_noCertificate" type="boolean" help="(-noCertificate) "
           label="Skip checking of certificate" />
    <param name="param_noVerbose" type="boolean" checked="true" help="(-noVerbose) "
           label="Print important information" />
    <param name="param_url" type="text" size="30" help="(-url) "
           label="URL to download" />
      <sanitizer>
        <valid initial="string.printable">
          <remove value="’"/>
          <remove value="&quot;"/>
        </valid>
      </sanitizer>
    </param>
  </inputs>
  <expand macro="advanced_options"/>
  <outputs>
    <data name="param_outputDocument" format="data"/>
  </outputs>
  <help></help>
</tool>
```

3.4 Discussion

There are other well–established formats that achieve the same result such as CWL task definitions and Galaxy ToolConfigs. In the end, both CWL files and ToolConfigs are structured documents that, generally speaking, could be converted to CTD files or to other formats. Of course, implementations would be required to properly handle sections that are CWL–specific or Galaxy–specific.

Since CTDs were designed to be a concise, platform–independent representation of tasks, it makes sense to use CTDs as a source format to export descriptors into other workflow–specific formats: one of our goals is to achieve integration of single tasks in workflow engines. The most sensible solution to achieve this would be to push for platform–independent formats, such as the one provided by CTD, and convert them across workflow engines.

Development of CTD converters is, in essence, not a complex task, but it certainly widens the range of workflow engines on which the described tools can be executed. Furthermore, seamless integration of tools in workflow engines is clearly a feature that surely will be appreciated by users lacking technical skills.

Software libraries that are usable across several workflow engines have value in the scientific community. The *bio.tools* application registry currently includes more than 12,300 tools and offers a custom descriptor that, similar to CTD, is not natively supported by workflow engines[69]. The Tool Description Generator (ToolDog) simplifies the inclusion of tools from the bio.tools registry into workflow engines and also features generation of CWL files and ToolConfigs[70].

CTDConverter, similar to ToolDog, is able to greatly simplify the inclusion of software libraries into workflow engines. Furthermore, the modular design of CTDConverter minimizes the effort needed to include new output formats.

Chapter 4

Conversion of Complete Workflows

Parts of this chapter were published in:

Schubert, B., **de la Garza, L.**, Mohr, C., Walzer, M., Kohlbacher, O., May 2017.
ImmunoNodes: Graphical Development of Complex Immunoinformatics Workflows.
BMC Bioinformatics, 18(1):242.

de la Garza, L., Aicheler, F., Kohlbacher, O., March 2017.
From the Desktop to the Grid and Cloud: Conversion of KNIME Workflows to WS–PGRADE.
In 8th International Workshop of Science Gateways, PeerJ Preprints, 5:e2849v1.

de la Garza, L., Veit, J., Szolek, A., Röttig, M., Aiche, S., Gesing, S., et al., March 2016.
From the Desktop to the Grid: Scalable Bioinformatics via Workflow Conversion.
BMC Bioinformatics, 17(1):127.

de la Garza, L., Krüger, J., Schärfe, C., Röttig, M., Aiche, S., et al., June 2013.
From the Desktop to the Grid: Conversion of KNIME Workflows to gUSE.
In 5th International Workshop on Science Gateways, CEUR Workshop Proceedings, 993:9.

4.1 Introduction

Improving existing workflow engines by adding features already present in other systems leads to effort duplication and to the allocation of resources that could otherwise be utilized in more advantageous efforts. This rationale is not an attempt to encourage stagnation in workflow management systems, rather, it calls for focusing efforts on interoperability between them[71].

It is possible to combine different workflow management systems via workflow conversion techniques, dealing directly with the differences across engines, in particular with the disparities of how the orchestration of task execution is ultimately implemented. In Section 2.2 we argued that, although limited in computing power, the KNIME Analytics Platform's most salient features are its intuitive workflow editor and its extensibility, while gUSE and WS–PGRADE constitute an adaptable back end engine offering interfaces to multiple DCIs, but its workflow editor imposes a steep learning curve on inexperienced users. Combining these two engines comes with an intrinsic substantial challenge: gUSE remotely invokes command line tools via wrapper scripts and uses data staging routines to pass files between tasks, while, in sharp contrast, KNIME Nodes are Java objects residing within the KNIME Analytics Platform that utilize in–memory data structures to transfer information to other nodes.

A proper conversion procedure must deal not only with the correct translation of workflow patterns (e.g., loops, conditionals, and parameter sweep sections) but also with the conversion of edges, which model how outputs of nodes are channeled as inputs to subsequent tasks. Furthermore, because workflows are composed of nodes, these must also be converted in a similar fashion as previously detailed in Chapter 3.

Efforts spent on combining workflow engines to raise their level of interoperability will directly benefit the scientific community: an automatic workflow conversion mechanism able to leverage prominent features of different engines with the purpose of curtailing their individual shortcomings can greatly improve usability and reduce computing times.

4.2 Methods

4.2.1 Definition of Workflows and Their Conversion

As discussed in Section 2.1.3, workflows are depicted as unweighted DAGs. Formally:

A workflow W is a pair, $W = (V, E)$, where:	
$V = \{v_1, v_2, ..., v_i\}$	is a non–empty, finite set of vertices (or nodes),
$E \subseteq \{(v_j, v_k) \mid v_j, v_k \in V \land j \neq k\}$	is a binary relation on V specifying directed edges between vertices
$(v, v) \notin E^+ \forall v \in V$	and the transitive closure of E is irreflexive

Definition 4.1: Formal definition of a workflow as an unweighted DAG. Adapted with permission from[72]. Copyright 2013, Springer–Verlag Berlin Heidelberg.

Furthermore, each vertex is assigned a set of properties identified by name (e.g., *id*, *name*). More formally, we model each vertex v as a function $v\colon k \to v$, as previously introduced in Section 3.2.1.

Algorithm 2 Conversion of complete workflows. The source workflow w_{f_s} will be exported to the target file o_{f_t} using an output format f_t. The first step is to convert nodes, extracting all relevant properties from each of the input nodes, after which edges can be added to construct an intermediate workflow representation, w. This can then be exported to a file, a process that depends on how the target platform represents nodes, edges and workflows, but is still captured here as an explicit serialization of nodes and edges.

```
procedure CONVERTWORKFLOW($w_{f_s}, o_{f_t}$)                 ▷ input workflow, output file
    $w \leftarrow EmptyWorkflow()$                        ▷ empty intermediate representation
    for all $n_{f_s}$ in $w_{f_s}.Nodes$ do                ▷ convert nodes from the input workflow
        $n \leftarrow FindOrConvert(n_{f_s}, w)$
        for all $e_{f_s}$ in $w_{f_s}.GetOutputsFor(n_{f_s})$ do
            $t \leftarrow FindOrConvert(e_{f_s}.Target, w)$
            $w.AddEdge(n, t)$                        ▷ add directed edge from $n$ towards $t$
        end for
    end for
    $ExportWorkflow(w, o_{f_t})$                           ▷ use a format–specific exporter
end procedure
function FINDORCONVERT($n_{f_s}, w$)
    $id \leftarrow n_{f_s}["id"]$                                ▷ retrieve unique identifier
    $n \leftarrow w[id]$                                  ▷ find node $n$ identified by $id$
    if $n$ is NULL then
        $n \leftarrow EmptyDictionary()$
        for all relevant $(k, v)$ in $n_{f_s}$ do
            $n[k] \leftarrow From_{f_s}(v)$                  ▷ translate property from input format
        end for
        $w[id] \leftarrow n$                      ▷ store and identify $n$ by its unique identifier
    end if
    return $n$
end function
function EXPORTWORKFLOW($w, o_{f_t}$)
    for all $n$ in $w.Nodes$ do
        $Write_{f_t}(n, o_{f_t})$                         ▷ write node using output format
    end for
    for all $e$ in $w.Edges$ do
        $Write_{f_t}(e, o_{f_t})$                         ▷ write edge using output format
    end for
end function
```

Our proposed method to convert complete workflows (summarized in Algorithm 2) extends a *source* workflow management system by modifying its runtime behavior with the purpose of generating a valid representation of the same workflow. This output can later be imported into a *target* engine (a feature available in the workflow engines we have studied), as previously summarized in Figure 1.1. Workflows and nodes in the source and target engines are represented using platform–specific formats, i.e., f_s and f_t, respectively.

All nodes and edges are first converted to constitute a platform–independent, intermediate workflow representation, i.e., w. Here, conversion of single nodes, although conceptually identical to Algorithm 1, is operationally different: runtime access to the input workflow and its constituting nodes is readily available. This intermediate representation contains the abstract layer from the source workflow as well as pertinent information to build concrete layers. During the last stage, a concrete layer for the target engine is built during the $ExportWorkflow$ routine, a process ultimately determined by the target engine and its workflow representation format, f_t.

Using an intermediate representation has a direct advantage: support for additional output formats can be included by implementing converters whose input is independent of the source platform. This not only facilitates development and automated testing but it also shields developers from changes in the source engine.

4.2.2 Workflow Interoperability

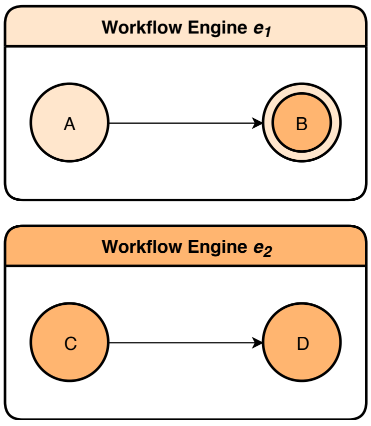
Converting complete workflows across workflow management systems will add certain degree of interoperability between them. Thus, grasping the nature and limitations of workflow interoperability is critical to the success of any conversion approach. The varying degrees of workflow interoperability, as summarized in Figure 4.1, can be categorized in the following levels[71]:

- Workflow–task level interoperability: certain workflow engines might feature tasks specifically designed for that engine (e.g., KNIME Nodes). This level of interoperability covers engines being capable of coordinating workflow tasks that were explicitly created for other workflow management systems.

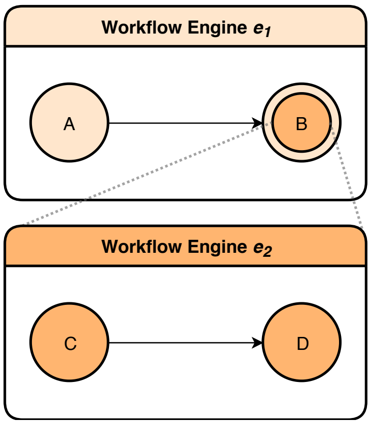
- Sub–workflow–task level: at this level of interoperability, workflow engines are able to share sub–workflows between them. This can be achieved by either including sub–workflows as black boxes or by replicating certain functionalities of systems. This is often referred to as coarse–grained interoperability[14].

- Complete workflow–task level: often referred to as fine–grained interoperability[14], this level includes scenarios where workflows designed for one system can be executed on another engine in an indistinguishable way.

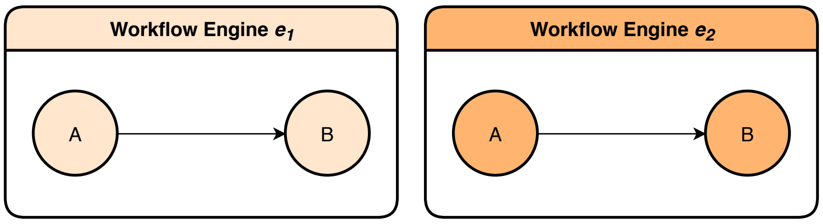
Achieving higher levels of interoperability requires greater efforts and poses new challenges, but this provides a more precise control over the execution of workflows, easing optimization of the involved computations (e.g., by executing several independent tasks in parallel). However, limits on the performance due to higher overhead costs can arise due to an extremely fine granularity. Too coarse–grained tasks can reduce performance due to a reduction in concurrency as the execution can no longer be split into smaller sections that could be executed independently[71]. Proper conversion of complete workflows must therefore deal with these questions.



(a) Workflow–task level interoperability. Through feature implementation, engine e_1 is able to execute tasks designed for other systems (i.e., task B).



(b) Sub–workflow level interoperability. Engine e_1 executes sub–workflows from other engines. Execution of task B is actually realized in the engine e_2.



(c) Complete workflow–task level interoperability. Engine e_1 is able to execute complete workflows designed for other systems.

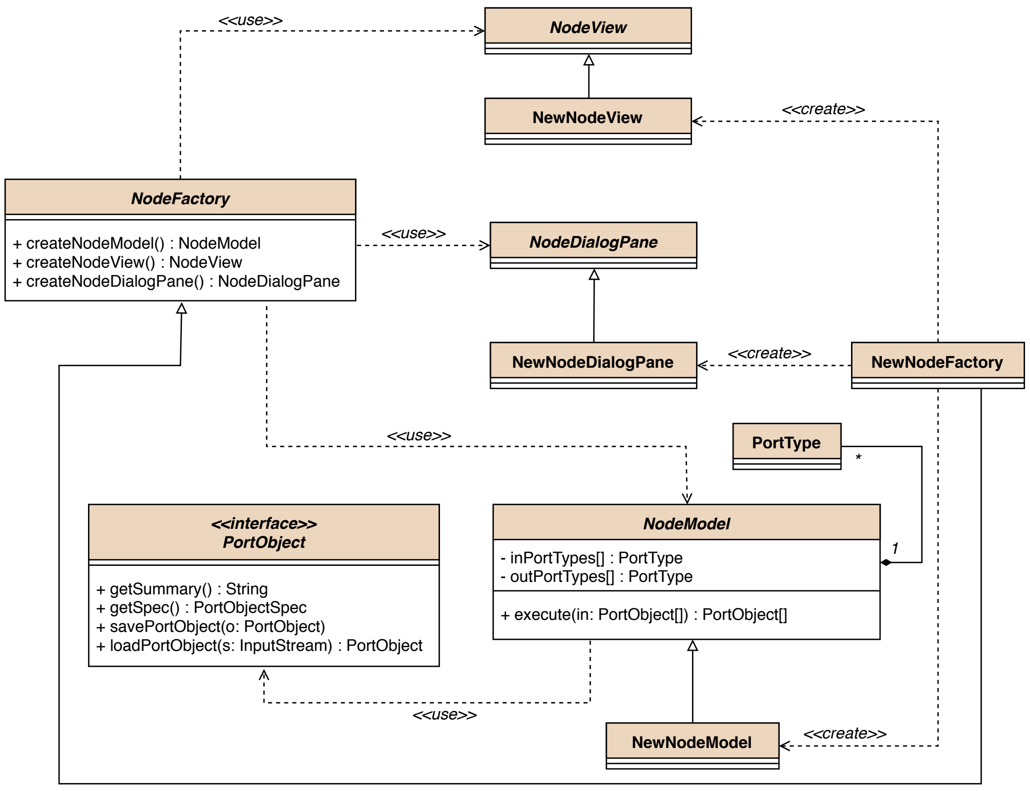
Figure 4.1: Workflow interoperability levels. Figures adapted with permission from[71]. Copyright 2009 Elsevier B.V.

4.2.3 Main Challenges

Discerning relevant discrepancies in the execution of workflow nodes, as performed by the involved engines, is the foremost action when combining systems to provide fine–grained interoperability. Converting KNIME workflows for their execution on other engines accessing DCIs will likely result in command line invocations: a proper implementation of a conversion method must devise the means to execute single KNIME Nodes from a workflow using a command line invocation. Furthermore, considering that our proposed method directly extends the functionality of running workflow management systems, any implementation must carefully study how features can be added to any involved engine. In addition to converting workflow nodes and properly recreating edges, translation of patterns such as parameter sweep must also be properly handled in an automatic workflow conversion procedure.

Execution of Workflow Nodes in the KNIME Analytics Platform

KNIME Nodes are composed of extensions of four abstract Java classes: *NodeModel*, *NodeDialog-Pane*, *NodeView* and *NodeFactory*[40]. These encapsulate the basic functionality, configuration, graphical representation, and instantiation of KNIME Nodes, respectively.

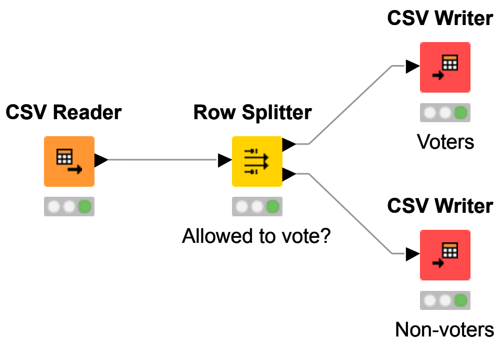


<<use>>
NodeView
NewNodeView
<<create>>
NodeFactory
+ createNodeModel() : NodeModel
+ createNodeView() : NodeView
+ createNodeDialogPane() : NodeDialogPane
<<use>>
NodeDialogPane
NewNodeDialogPane
<<create>>
NewNodeFactory
<<use>>
PortType
*
<<interface>> PortObject
+ getSummary() : String
+ getSpec() : PortObjectSpec
+ savePortObject(o: PortObject)
+ loadPortObject(s: InputStream) : PortObject
NodeModel
- inPortTypes[] : PortType
- outPortTypes[] : PortType
+ execute(in: PortObject[]) : PortObject[]
1
<<use>>
NewNodeModel
<<create>>

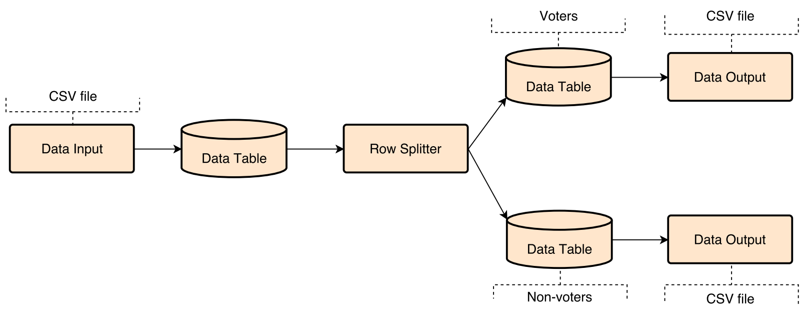
Figure 4.2: Reduced class diagram of a KNIME Node. Classes extending *NodeModel* implement computations on input data, as well as generation of output data. The class *NodeDialogPane* provides a basic framework to allow node configuration through a GUI, while *NodeView* handles data visualization. *NodeFactory* is responsible for the instantiation of needed components. Figure adapted with permission from[40]. Copyright 2009 ACM.

Nodes that follow the KNIME development guidelines seamlessly interact with other KNIME Nodes. This makes KNIME Community Contributions a vibrant community: once a node is part of the main repository, it can easily be imported into a workflow and interact with other nodes. However, KNIME Nodes suffer from certain lack of portability: nodes are instances of Java classes spawned by the process under which the KNIME Analytics Platform is executed.

Data consumed and produced by nodes via their ports in the KNIME Analytics Platform are organized in rows and columns, and are contained in *Data Tables*. Each row is automatically assigned a unique identifier and has an inherent index. Nodes access data in Data Tables by iterating over each individual row. This allows to process large amounts of data without having to store the complete contents of Data Tables in memory at once[40]. Figure 4.3 depicts a typical data flow between KNIME Nodes.



(a) Simple KNIME workflow. Data are imported using the *CSV Reader* node and serialized using *CSV Writer* nodes. The *Row Splitter* node is designed to separate data into two configurable categories. Rows are split based on whether the age column contains a number equal or greater than 18.



(b) Data flow from the simple workflow shown above. Here we see how KNIME Nodes exclusively interact with Data Table objects, even if, from the end user's perspective, they seem to interact with files. Adapted with permission from[40]. Copyright 2009 ACM.

Figure 4.3: Relationship between KNIME Nodes and Data Tables.

Interaction with Command Line Tools The Generic KNIME Nodes (GKN) extension, part of the KNIME Community Contributions, offers an easy way to create KNIME Nodes that interact with external command line tools[73,74]. GKN is comprised of two components: a standalone *Node Generator* and a plug–in for the KNIME Analytics Platform. Nodes generated and managed by GKN can interact with other KNIME Nodes and are virtually indistinguishable from other nodes. The Node Generator (offered as an Apache Ant script) automates node creation, while the plug–in resides within the KNIME Analytics Platform and manages all execution, visualization and configuration aspects of the generated nodes. GKN supports both CTD–enabled and non–CTD–enabled tools: CTD files are used only as a vehicle to describe command line tools that will ultimately be invoked from the KNIME Analytics Platform.

GKN users must provide a descriptor declaring media types allowed for inputs and outputs, along with a CTD file for each command line tool that will be integrated. A simple text editor and no programming experience are required to generate these files.

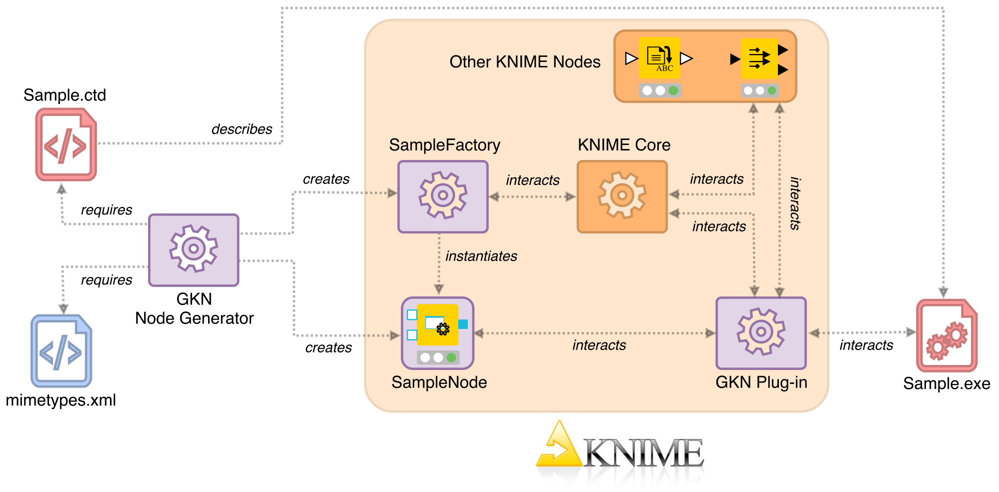


Figure 4.4: Interaction between GKN's components—shown in purple—and the KNIME Analytics Platform. The KNIME Core components are able to interact with the generated nodes via the GKN plug–in. Generation of a KNIME Node requires a CTD describing an external command line tool (e.g., *Sample.ctd* and *Sample.exe*, respectively). Media types are provided in the *mimetypes.xml* file. The Node Generator creates and compiles Java code *on the fly*.

Although nodes managed by GKN are virtually indistinguishable from native KNIME Nodes for end users, these clearly operate in such a different manner that an automatic conversion mechanism must acknowledge these disparities and foresee adequate procedures for their conversion, as illustrated by Figure 4.4.

Executing the KNIME Analytics Platform as a Command Line Tool Since our aim is to provide fine–grained interoperability, conversion of the individual tasks that comprise a KNIME workflow must be implemented. As we have seen, execution of KNIME Nodes departs from the common command line approach through which tasks are invoked on HPC resources. Nevertheless, it is possible to execute whole workflows in the KNIME Analytics Platform without a GUI by invoking the KNIME Analytics Platform in *headless mode* (often referred to as *batch mode*)[75]. This allows users to provide the name and location of a workflow to execute along any needed parameter values. This functionality provides immediate coarse–grained workflow interoperability (i.e., workflows are treated as a whole), so this solution alone is not enough to reach our aim of providing fine–grained workflow interoperability, but it provides a solid foundation to execute individual KNIME Nodes from a workflow.

Installing the KNIME Analytics Platform as an application on HPC resources is a procedure no different than to install it on a desktop computer. However, depending on the security settings of the HPC resource in question, this step must be performed by a system administrator as a one–time only action. One important aspect to keep in mind is that all involved KNIME Analytics Platform instances in a workflow conversion process must contain the same nodes, otherwise, difficult to trace runtime errors might arise during the execution of converted workflows. Furthermore, certain KNIME Nodes require external dependencies, e.g., the *R Snippet* KNIME Node depends on an R server (*Rserve*)[76].

Execution of Workflow Nodes in gUSE

As introduced in in Section 2.2.2, the DCI Bridge manages all task–related requests and is fully compatible with JSDL, enabling other workflow engines to interact with it[44,77]. Interaction between DCIs and the DCI Bridge happens through *DCI Submitters*. DCI Submitters are ultimately responsible for the execution of tasks in gUSE. They are a collection of Java classes that directly interact with computing resources in order to submit, monitor and cancel jobs (as depicted in Figure 4.5). This communication can be realized via, e.g., encrypted Secure Shell (SSH) or a vendor–provided API[78].

A DCI Submitter is accompanied by the necessary web pages to extend WS–PGRADE in order to offer a graphical interface to configure tasks that will be executed by said DCI Submitter, as previously shown in Figure 2.10. Development of DCI Submitters and their companion web pages requires a medium to advanced level of Java and Java Server Pages (JSP) knowledge. These JSPs are where WS–PGRADE users provide custom scripts that will ultimately submit jobs on DCIs[78]. A proper conversion mechanism must automatically generate similar scripts able to invoke remote command line tools for the respective workflow nodes.

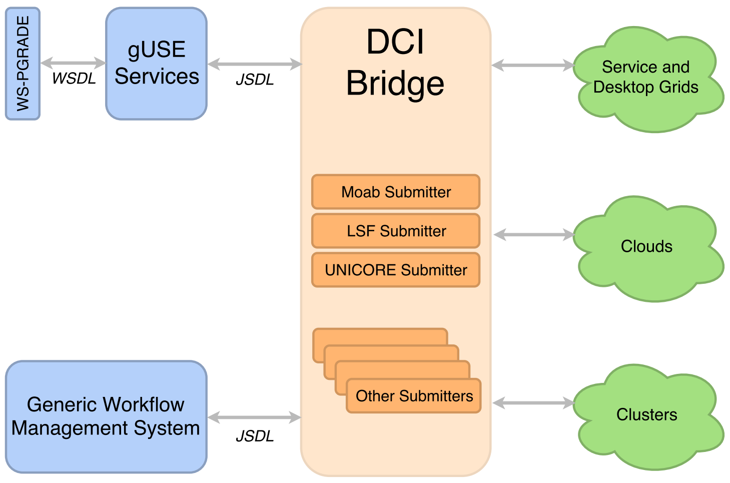


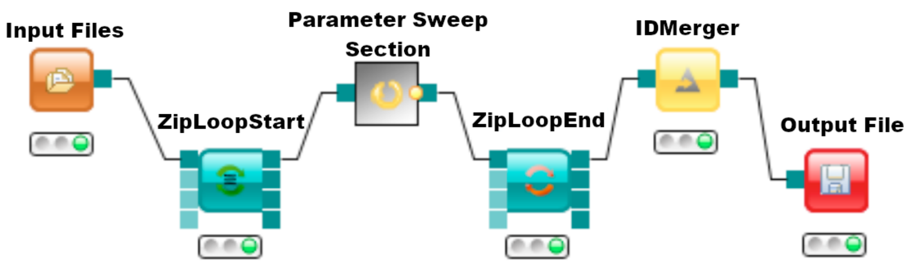
Figure 4.5: Schematic representation DCI Bridge's interaction with gUSE and other workflow management systems. Adapted with permission from[78]. Copyright 2015 MTA SZTAKI LPDS, Budapest.

Conversion of Workflow Patterns

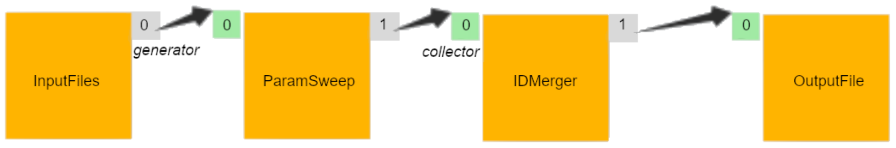
The *ZipLoopStart* and *ZipLoopEnd* KNIME Nodes perform a parametric sweep execution of the enclosed nodes. A list of uniform resource identifiers (URI), each pointing to an input file, is provided by *ZipLoopStart*. Given an input of n URIs, the encircled sub–workflow will be executed n times (each iteration receives a different URI). *ZipLoopEnd* will collect all files produced during the iterations, delivering a list containing the URIs of these files to any subsequent KNIME Nodes[74].

In contrast to the KNIME Analytics Platform, gUSE expects input and output files to be named after their corresponding ports. Since the number of ports a node contains is determined during the design phase[1], lists of files are not natively supported by gUSE. Furthermore, parametric sweep sections are not enclosed by specialized nodes, rather, by generator and collector ports[50]. Any port can be configured to function as such in the concrete node configuration page in WS–PGRADE shown in Figure 2.10. For instance, in the case of a port named *out.txt*, the process associated with its parent node must generate files named *out.txt_0*, *out.txt_1*, . . . , *out.txt_n*: gUSE will concurrently execute the enclosed section $n + 1$ times (using a different file for each iteration). Collector ports represent the inverse operation performed by generator ports. Workflow developers are thus responsible for the generation of these files and their proper naming[50]. These differences are summarized in Figure 4.6.

[1]This is a typical behavior of the workflow engines we have studied and not a limitation of gUSE.



(a) The section enclosed between the *ZipLoopStart* and *ZipLoopEnd* nodes will be invoked once for every input file. *ZipLoopEnd* will collect all results of the enclosed section and pass them to *IDMerger* as a list of URIs.



(b) *InputFiles*' output port has been configured as a generator port while *IDMerger*'s input port has been declared as a collector port. The sub–workflow between these ports will be executed once for every file associated with the generator port. Resulting files will be collected by *IDMerger*.

Figure 4.6: Implementation of the parameter sweep pattern in the KNIME Analytics Platform (top) and WS–PGRADE.

Extending the KNIME Analytics Platform

The KNIME Analytics Platform's *look and feel* might remind users of Eclipse, the integrated development environment (IDE), since both were built with the Eclipse Modeling Framework (EMF). This positions the KNIME Analytics Platform as an extensible, adaptable platform to which complex features such as plug–ins can be added[40,79].

Extensions for the KNIME Analytics Platform have access to the *WorkflowManager* class (part of the KNIME Core components), through which developers can query the current state of workflows and their comprising nodes. This class also contains methods to manipulate all aspects of a workflow, e.g., add and remove nodes, start and cancel the execution, and edit edges.

Extending WS–PGRADE

Users who desire to execute workflow tasks on any of the supported batch queueing systems in gUSE (e.g., Sun Grid Engine, Portable Batch System and Moab) are required to provide file paths of remote binaries to execute, along with their corresponding command lines, as previously shown in Figure 2.10. Since conversion of complete workflows will invariably result in generation of concrete layers (see Figure 1.1), an instrument to associate applications with DCIs to perform a mapping between tasks comprising the concrete layers must also be devised.

Importing Workflows in WS–PGRADE WS–PGRADE offers a workflow import mechanism through its standard GUI, where users upload compressed archives. These contain a directory structure in which input files and executable scripts are placed along an XML file (named *workflow.xml*) where both the abstract and concrete layers are defined, as depicted in Figure 4.7. Entries in these archives refer to elements in the *workflow.xml* file. A sample *workflow.xml* file is provided in Listing B.14.

```
<sample_worfklow.zip>
        ├── workflow.xml
        │
        └── nodes/
              ├── MoleculeCheck/
              │           ├── execute.bin
              │           └── 0/
              │               └── 0
              │
              └── 3DGenerator/
                          ├── execute.bin
                          ├── 0/
                          │   └── 0
                          └── 1/
                              └── 0
```

Figure 4.7: Contents of a compressed file containing a WS–PGRADE workflow similar to the one introduced in Section 2.1.3 (the *3DGenerator* job is depicted here with two input ports for illustration purposes). An XML file describing both the abstract and the concrete layers, *workflow.xml*, is required. Each node receives its own folder, under which an executable script with the fixed name *execute.bin* is provided. Ports are identified by their 0–based index, their inputs—regardless of content or type—must be named *0*.

Resource Management in gUSE Managing access to DCIs, a task typically carried out by administrators, is performed using the DCI Bridge component. Workflow tasks can only interact with already existent, active DCIs. Programmatic access to this data is achieved via a gUSE component that acts as a proxy to the DCI Bridge, *Information System*, briefly introduced in Section 2.2.2. Additionally, because UNICORE's remote API offers read–only access to its application registry, WS–PGRADE is able to display a read–only view of these applications in the node configuration pages introduced in Section 2.2.2, so a proper implementation must be able to differentiate between these two types of remote computing resources.

Portlet Development WS–PGRADE is offered as a full–fledged Liferay portal: complex, responsive GUI elements such as *Vaadin widgets* are available for portlet developers. Given that it is distributed as a set of web applications along Apache Tomcat (a web server that is able to act as a *Servlet Container*[80]), any portlet has access to core initialization routines through Tomcat's *ApplicationContext* class[80].

Development and deployment of portlets for WS–PGRADE requires more than 40 external libraries contained in 10 different repositories. Apache Maven, via the shipped Project Object Model (POM) documents, perfectly handles each of the dependencies and their required versions when assembling, or *packaging*, the binaries that will ultimately be installed on a running WS–PGRADE instance, i.e., web application resource (WAR) files[81].

Manual installation of portlets on a running Liferay portal entails navigation through the portal's *Control Panel*, where a WAR file can be uploaded. Liferay will then deploy all included portlets, making them available for their use. After deploying a portlet that depends on any of the gUSE services, the Liferay portal on which the WS–PGRADE instance is hosted (i.e., the Apache Tomcat web server) must be restarted. This is due to the fact that the Information System component, upon start–up, initializes registered components by invoking web services and injecting credentials, that is, Information System acts as the *subject* to all registered *observer* components, as described by the *observer* design pattern[82]. Any component deployed after the Information System's loading routine has concluded will not be properly initialized and therefore will not be able to interact with other gUSE services.

WS–PGRADE/gUSE offers the Application Specific Module (ASM) Java library to simplify development of application–specific interfaces able to control all aspects of workflows. Developers can enrich their Liferay portlets with ASM to develop GUIs for their domain–specific solutions[83].

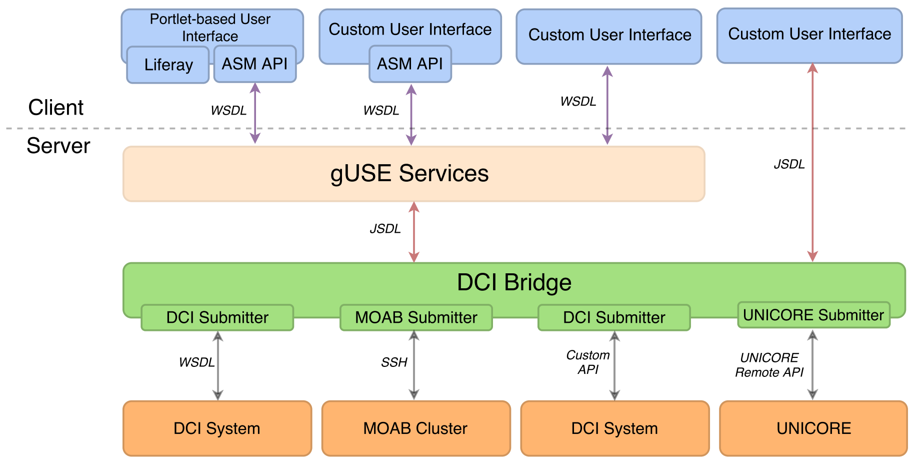


Figure 4.8: gUSE's development stack. ASM encapsulates the most common tasks in its API, but gUSE and DCI Bridge can still be accessed directly. DCI Submitters communicate with their corresponding DCI systems, each using a different communication protocol. Adapted with permission from[83]. Copyright 2013 Balaskó et al.[83].

ASM provides an extra layer of abstraction to access gUSE: domain–specific portlet developers can thus focus on tasks relevant to their application field. Scientific portals such as MoSGrid[84], the VisIVO Science Gateway[85], and HELIOGate[86] are just a few examples of development of such applications that benefit from ASM. ASM is, however, not required to access neither gUSE's services nor DCI Bridge's components, as summarized in Figure 4.8.

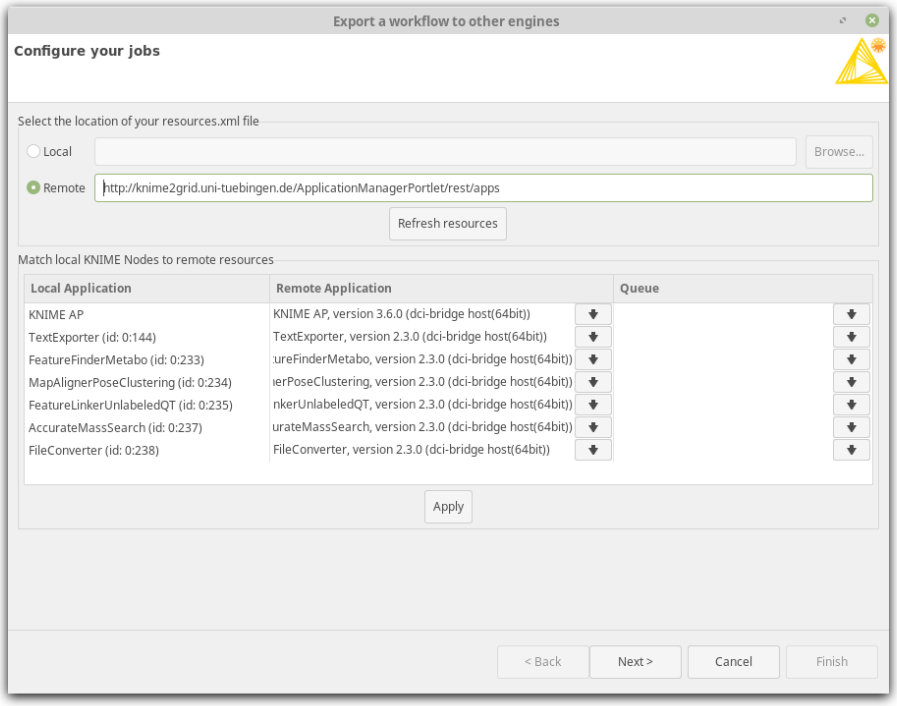
4.3 Results

4.3.1 The KNIME2Grid Extension for the KNIME Analytics Platform

We developed a plug–in extension for the KNIME Analytics Platform that, working together with GKN and the *Application Manager Portlet* (discussed in detail in Section 4.3.2), provides fine–grained interoperability between WS–PGRADE/gUSE and the KNIME Analytics Platform.

Users create and test their workflows in the KNIME Analytics Platform as any other KNIME workflow. Fully integrated into the KNIME Analytics Platform's GUI, our KNIME plug–in, KNIME2Grid, features a workflow export wizard that assists users to convert and configure workflows so they can be later imported into a WS–PGRADE portal (see Figures 4.9 and 4.10).

We have thus successfully combined prominent features of two workflow engines to provide a solution to design, create and test workflows using a user–friendly interface (KNIME Analytics Platform), while at the same time providing execution on HPC resources through the use of an extensible back–end (WS–PGRADE/gUSE).


Export a workflow to other engines

Configure your jobs

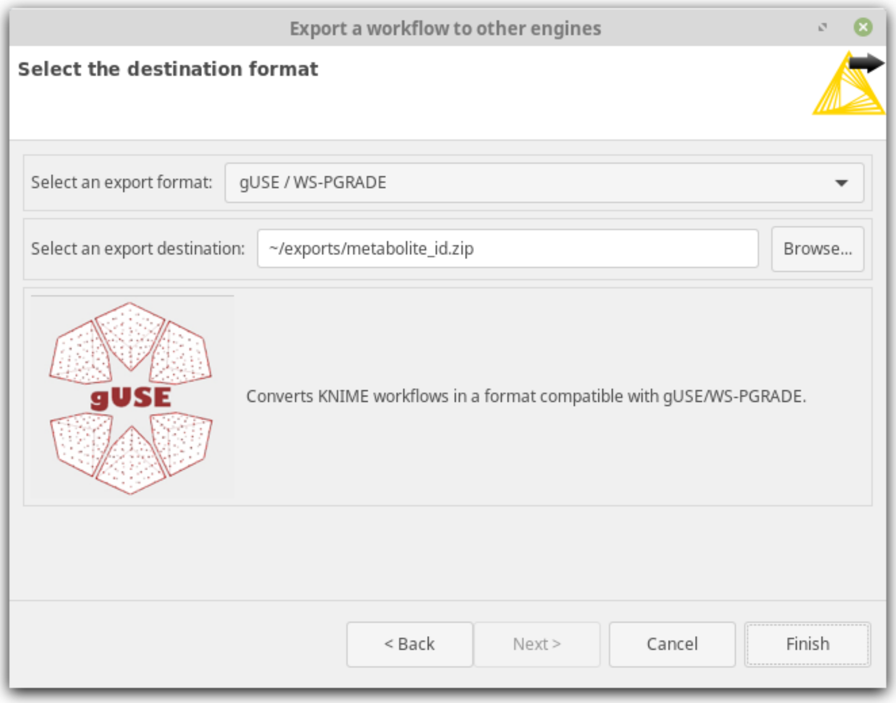
Select the location of your resources.xml file

Local [✓ Remote] http://knime2grid.uni-tuebingen.de/ApplicationManagerPortlet/rest/apps

Match local KNIME Nodes to remote resources

Local Application	Remote Application	Queue
KNIME AP	KNIME AP, version 3.6.0 (dci-bridge host(64bit))	
TextExporter (id: 0:144)	TextExporter, version 2.3.0 (dci-bridge host(64bit))	
FeatureFinderMetabo (id: 0:233)	:ureFinderMetabo, version 2.3.0 (dci-bridge host(64bit))	
MapAlignerPoseClustering (id: 0:234)	ıerPoseClustering, version 2.3.0 (dci-bridge host(64bit))	
FeatureLinkerUnlabeledQT (id: 0:235)	nkerUnlabeledQT, version 2.3.0 (dci-bridge host(64bit))	
AccurateMassSearch (id: 0:237)	urateMassSearch, version 2.3.0 (dci-bridge host(64bit))	
FileConverter (id: 0:238)	FileConverter, version 2.3.0 (dci-bridge host(64bit))	

Figure 4.9: Concrete configuration using our proposed workflow exporter included in KNIME2-Grid.


Export a workflow to other engines

Select the destination format

Select an export format: gUSE / WS-PGRADE

Select an export destination: ~/exports/metabolite_id.zip

Converts KNIME workflows in a format compatible with gUSE/WS-PGRADE.

Figure 4.10: Exporting a converted workflow using KNIME2Grid.

Overall Design

The interaction between the main components of KNIME2Grid during a typical workflow conversion are depicted in Figure 4.11. Furthermore, we relied on the *dependency injection* software development technique, as defined by Fowler[87], in order to provide a flexible, robust and extensible implementation. Figure 4.12 depicts the most relevant components of KNIME2Grid.

Having resilience and extensibility in mind, our method calls for an internal workflow representation, and we created a suitable implementation, as depicted in Figure 4.13. Our proposed internal workflow model closely mimics the abstract layer and contains information required to construct concrete layers: the model assumes that each *Job* can be executed as a command line tool that receives parameters and input files, and produces output files. In the end, our purpose is to export workflows to platforms that interact with command line tools, so we deem this decision as a practical compromise. Nevertheless, no further assumptions about target workflow engines are made. We created this model to add an extra layer of abstraction and to isolate components from changes in the KNIME Analytics Platform.

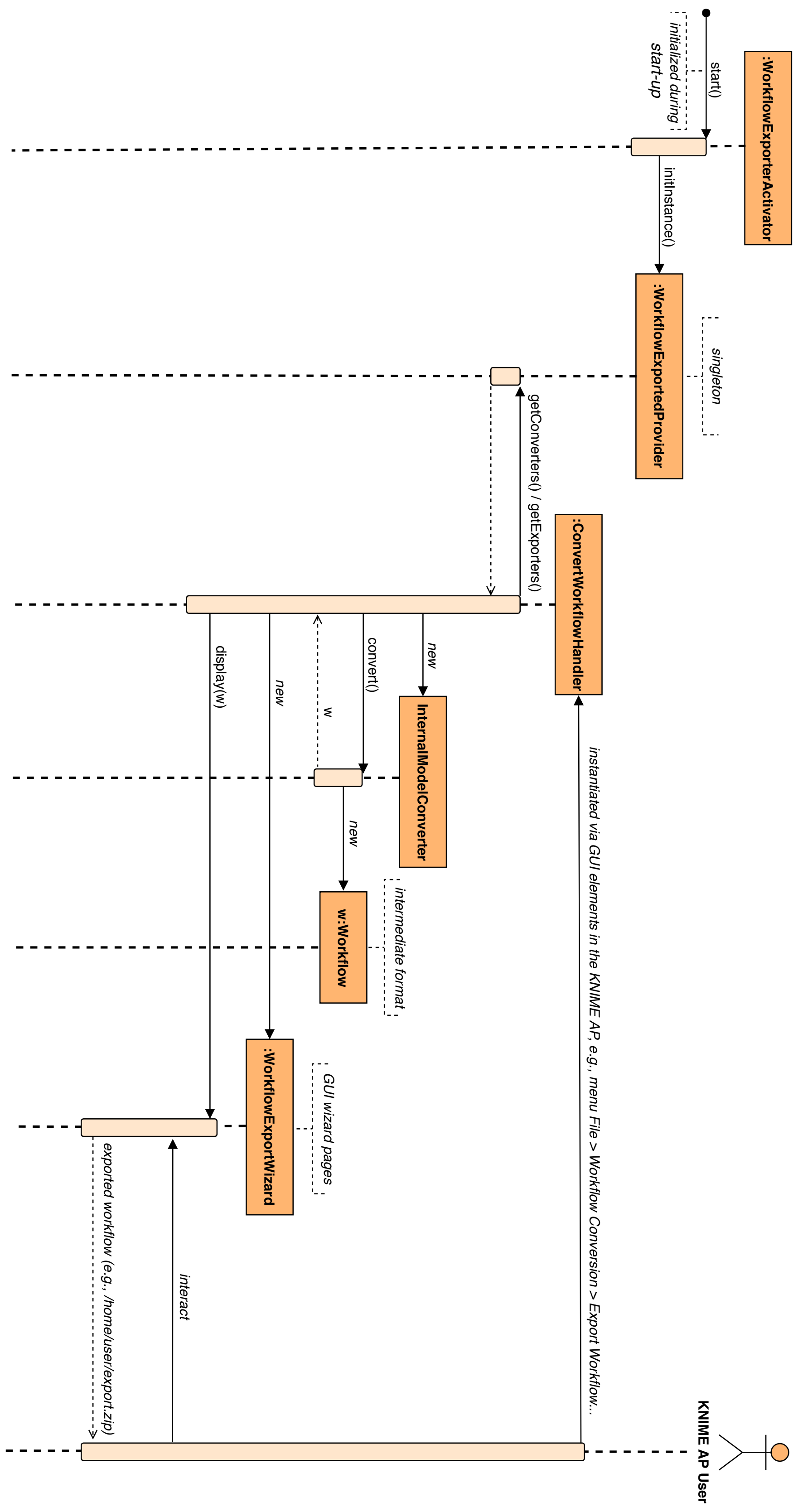
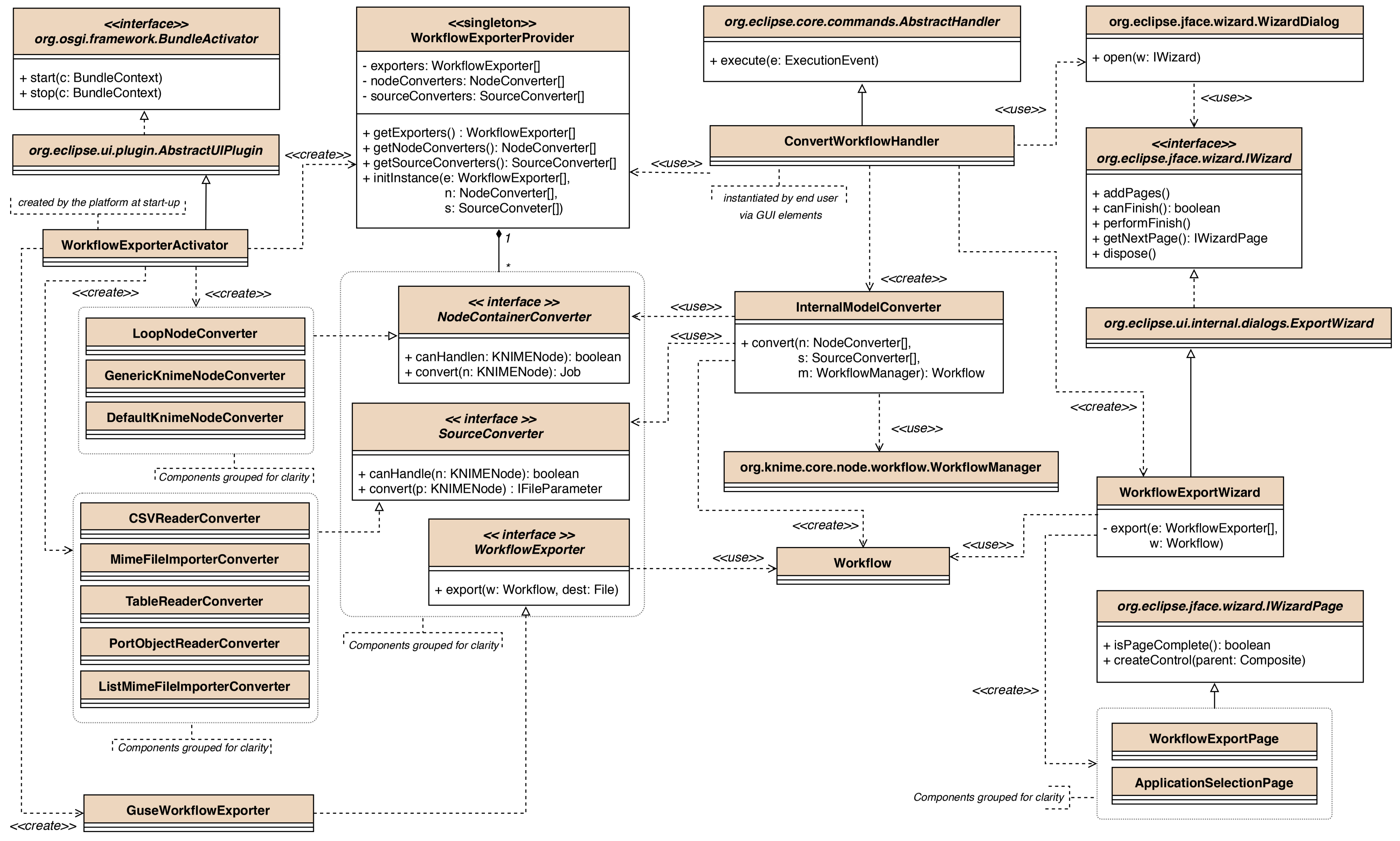

:WorkflowExporterActivator
start()
initialized during start-up
initInstance()
:WorkflowExportedProvider
singleton
getConverters() / getExporters()
:ConvertWorkflowHandler
new
:InternalModelConverter
convert()
w
new
w:Workflow
intermediate format
new
display(w)
:WorkflowExportWizard
GUI wizard pages
interact
exported workflow (e.g., /home/user/export.zip)
instantiated via GUI elements in the KNIME AP, e.g., menu File > Workflow Conversion > Export Workflow...
KNIME AP User

Figure 4.11: Sequence diagram of a conversion performed using the workflow export wizard shown in Figures 4.9 and 4.10.



<<interface>>
org.osgi.framework.BundleActivator
+ start(c: BundleContext)
+ stop(c: BundleContext)
org.eclipse.ui.plugin.AbstractUIPlugin
created by the platform at start-up
WorkflowExporterActivator
<<create>>
<<create>>
<<create>>
LoopNodeConverter
GenericKnimeNodeConverter
DefaultKnimeNodeConverter
Components grouped for clarity
CSVReaderConverter
MimeFileImporterConverter
TableReaderConverter
PortObjectReaderConverter
ListMimeFileImporterConverter
Components grouped for clarity
GuseWorkflowExporter
<<create>>
<<create>>
<<singleton>>
WorkflowExporterProvider
- exporters: WorkflowExporter[]
- nodeConverters: NodeConverter[]
- sourceConverters: SourceConverter[]
+ getExporters() : WorkflowExporter[]
+ getNodeConverters(): NodeConverter[]
+ getSourceConverters(): SourceConverter[]
+ initInstance(e: WorkflowExporter[], n: NodeConverter[], s: SourceConveter[])
1
*
<< interface >>
NodeContainerConverter
+ canHandlen: KNIMENode): boolean
+ convert(n: KNIMENode): Job
<< interface >>
SourceConverter
+ canHandle(n: KNIMENode): boolean
+ convert(p: KNIMENode) : IFileParameter
<< interface >>
WorkflowExporter
+ export(w: Workflow, dest: File)
Components grouped for clarity
org.eclipse.core.commands.AbstractHandler
+ execute(e: ExecutionEvent)
ConvertWorkflowHandler
<<use>>
instantiated by end user via GUI elements
<<use>>
<<create>>
InternalModelConverter
+ convert(n: NodeConverter[], s: SourceConverter[], m: WorkflowManager): Workflow
<<use>>
<<use>>
<<use>>
org.knime.core.node.workflow.WorkflowManager
<<create>>
<<use>>
Workflow
<<use>>
org.eclipse.jface.wizard.WizardDialog
+ open(w: IWizard)
<<use>>
<<interface>>
org.eclipse.jface.wizard.IWizard
+ addPages()
+ canFinish(): boolean
+ performFinish()
+ getNextPage(): IWizardPage
+ dispose()
org.eclipse.ui.internal.dialogs.ExportWizard
<<create>>
WorkflowExportWizard
- export(e: WorkflowExporter[], w: Workflow)
org.eclipse.jface.wizard.IWizardPage
+ isPageComplete(): boolean
+ createControl(parent: Composite)
<<create>>
WorkflowExportPage
ApplicationSelectionPage
Components grouped for clarity

Figure 4.12: Class diagram of the KNIME2Grid plug–in.

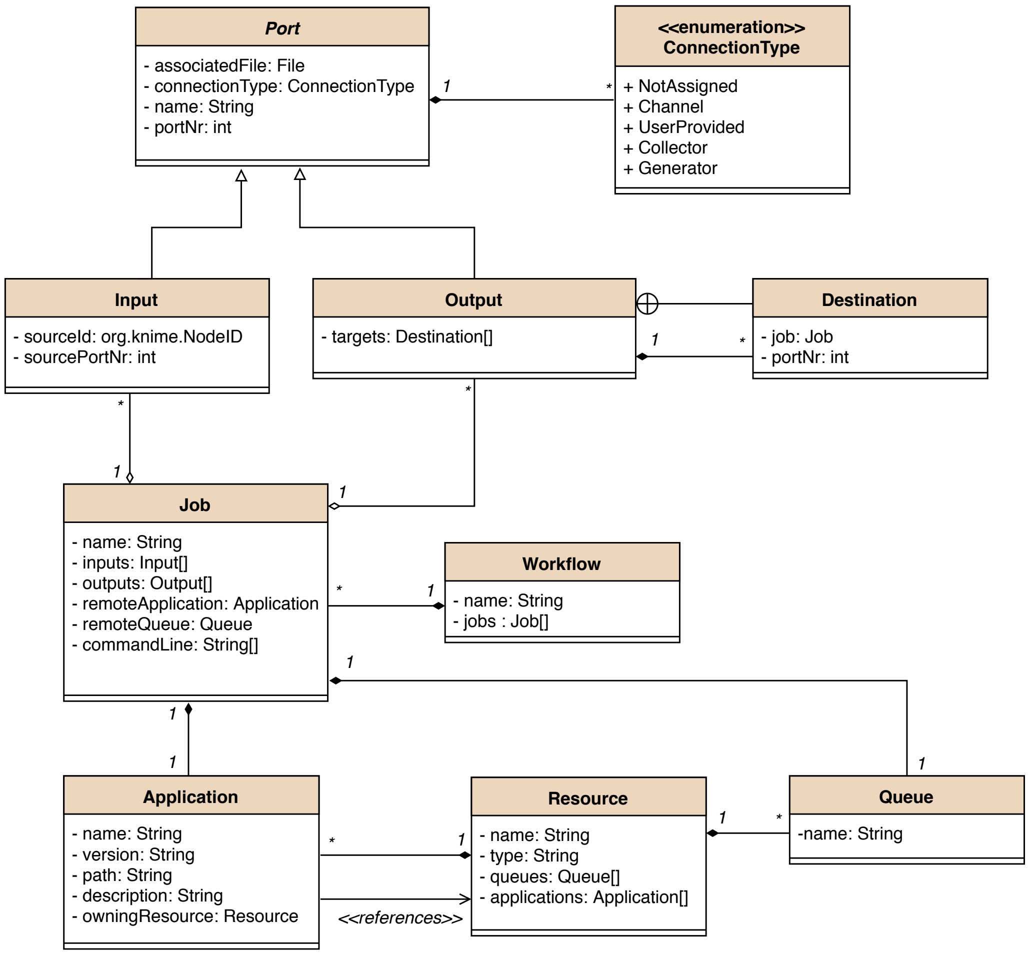


Figure 4.13: Class diagram of our internal workflow model. Using this generic representation of a workflow provides stability and flexibility to our proposed extension. Note: *setter* and *getter* methods were omitted for brevity.

Categorization of KNIME Nodes Before embarking on complete workflow conversion, our implementation makes a clear distinction of KNIME Nodes based on their operation. This is an important aspect since a successful conversion must ultimately execute KNIME Nodes as individual tasks within the context of a different workflow engine. KNIME2Grid categorizes KNIME Nodes as follows:

- Loop nodes: sections enclosed by *ZipLoopStart* and *ZipLoopEnd* KNIME Nodes correspond to parameter sweep operations, as such, implementation varies across platforms.
- KNIME Nodes imported via the GKN extension: these nodes represent an external command line tool that is independent of the KNIME Analytics Platform.
- Other KNIME Nodes: this category represents nodes that require a running KNIME Analytics Platform to execute.

Implementations of our *NodeContainerConverter* interface convert specific KNIME Nodes to *Job* instances. Supplementary KNIME Node converters, if ever needed, can be added without modifying any core components. Similarly, the *SourceConverter* interface defines the contract to transform input data files into instances of *IFileParameter*, an interface that represents files and lists of files, along with their allowed media types. These interfaces protect KNIME2Grid against future changes in the KNIME Analytics Platform.

Implementation of Workflow Exporters Although KNIME2Grid currently only supports WS–PGRADE/gUSE as a target platform via the *GuseWorkflowExporter* class, we created an interface with the purpose of providing additional export formats. This interface, *Workflow-Exporter*, defines a single method receiving an instance of our intermediate workflow model and a destination file. There are no required KNIME dependencies: developers can build and test converters outside the context of any KNIME component.

Initialization and Dependencies

KNIME2Grid is an extension offered as an EMF *bundle*. As such, activation and initialization routines occur within the context of the EMF: the platform invokes the *start* method of all registered bundles during start–up.

The class *WorkflowExporterActivator*, through its inheritance chain, is an implementation of the *BundleActivor* interface offered by the EMF. Our activator acts as the entry point of KNIME2Grid, injecting required dependencies into a singleton class, *WorkflowExporterProvider*, acting as a bridge between the entry point and the rest of the components, shielding most parts of our code from implementation details: whether this singleton was created within the context of a platform or during testing is irrelevant to classes depending on it.

Generation of an Intermediate Workflow Representation

The *InternalModelConverter* class produces intermediate workflow representations (refer to Figure 4.13), directly interacting with KNIME's *WorkflowManager*, which is used to extract information about the workflow being converted. *WorkflowManager* is, in fact, contained within a *WorkflowEditor*, which is the abstraction of the graphical editor users interact with. Based on the idea behind our internal workfow model, *InternalModeConverter* separates KNIME implementation details from the user interface comprising our workflow export wizard.

Configuring Concrete Layers Using the Workflow Export Wizard

Our *WorkflowExportWizard* class interacts with end users. Internally, it receives an instance of our internal workflow model and a list of workflow exporters. It derives from the *ExportWizard* abstract class contained in the EMF. Wizards in the EMF are defined as a sequence of *IWizardPages*. Each of these pages models a screen in the EMF's GUI, as shown in Figures 4.9 and 4.10.

Before being able to export a workflow to WS–PGRADE, users need to configure the concrete layer. Our workflow export wizard assists users to configure each of the nodes comprising the exported workflow. The purpose of our proposed wizard is not to replace WS–PGRADE's configuration pages (introduced in Section 2.2.2), rather, to leverage KNIME Analytics Platform's user–friendly GUI to offer a better user experience.

Our wizard is fully integrated into the KNIME Analytics Platform's user interface via menu and toolbar elements. Users are first prompted for an XML file containing a resource descriptor file (see sample file in Listing 4.4). If no local copy is available, users can enter a URL to obtain one via the Representational State Transfer (REST) API offered by the Application Manager Portlet[2], which we will introduce in Section 4.3.2. Next, users press the *Refresh resources* button to fill out the *Remote Application* and *Queue* columns. The table associates KNIME Nodes with applications available on the desired target gUSE installation. This action automatically preselects remote applications based on the Levenshtein distance[88] between the names of KNIME Nodes and remote applications to accelerate the configuration process (users can override this behavior to choose a more suitable remote application). Nodes that represent an external command line tool will be individually displayed in the *Local Application* column. However, since all native KNIME Nodes exist only within the context of a running KNIME Analytics Platform, they will be represented as a single item in this column.

Pressing the *Apply* button saves changes and advances users to the next screen. Here, users provide the location on which the exported workflow will be saved. Our extension is able to generate archives that can be imported into any WS–PGRADE portal.

Execution of Converted Nodes in gUSE

We previously introduced how gUSE utilizes user–provided scripts to execute command line tools on DCIs. *GuseWorkflowExporter* automatically generates these scripts during the export process, performing in–place substitution of variables in template scripts using placeholders, e.g., `@@EXECUTABLE@@`. Conversion of ports associated with a list of files is realized via the use of compressed archives.

Loop Nodes Listings 4.1 and 4.2 present the template scripts to realize the conversion of KNIME loop nodes in gUSE. *GuseWorkflowExporter* will additionally mark the corresponding ports as generator/collector in the generated *workflow.xml* file containing the converted workflow.

[2]e.g., `http://portal.org/ApplicationManagerPortlet/rest/apps`

Listing 4.1: Script template implementing *ZipLoopStart* in gUSE.

```
#!/usr/bin/env bash

# we know that the port name refers to an archive (e.g., foo.tar.gz)
INPUT_PORT_NAME="@@INPUT_PORT_NAME@@"
OUTPUT_BASE_NAME="@@OUTPUT_BASE_NAME@@"

# gUSE expects files from a generator to be named, e.g., bar_0, bar_1, ...
FILENAME_INDEX=0
for input_file in `tar tfz ${INPUT_PORT_NAME}`; do
  tar xvfOz ${INPUT_PORT_NAME} ${input_file} > \
                       ${OUTPUT_BASE_NAME}_${FILENAME_INDEX}
  FILENAME_INDEX=$(expr ${FILENAME_INDEX} + 1)
done
```

Listing 4.2: Script template implementing *ZipLoopEnd* in gUSE.

```
#!/usr/bin/env bash

# gUSE will provide files named after this port name, e.g., foo_0, foo_1, ...
INPUT_BASE_NAME="@@INPUT_BASE_NAME@@"
OUTPUT_PORT_NAME="@@OUTPUT_PORT_NAME@@"

tar cvfz ${OUTPUT_PORT_NAME} ${INPUT_BASE_NAME}_*
rm ${INPUT_BASE_NAME}_*
```

Other Nodes and List of Files Other types of converted nodes in gUSE will be executed via the wrapper template script shown in Listing 4.3.

Listing 4.3: Wrapper script template capable to handle dynamic file lists on gUSE.

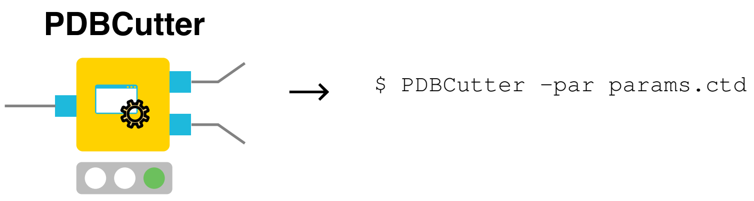
```
#!/usr/bin/env bash

# contains names of input ports that take filelists, separated by whitespace
INPUT_PORTS_WITH_FILELIST="@@INPUT_PORTS_WITH_FILELIST@@"

##### start filename translation variables
@@INPUT_FILENAME_TRANSLATION@@
##### end

# contains names of output ports that generate filelists, separated by whitespace
OUTPUT_PORTS_WITH_FILELIST="@@OUTPUT_PORTS_WITH_FILELIST@@"
EXECUTABLE="@@EXECUTABLE@@"
COMMAND_LINE_PARAMETERS="@@COMMAND_LINE_PARAMETERS@@"

ARCHIVE_INDEX=0
if [ -n "$INPUT_PORTS_WITH_FILELIST" ]; then
  for input_port in ${INPUT_PORTS_WITH_FILELIST}; do
    echo "expanding ${input_port}"
    # extract files individually and rename them
    FILE_INDEX=0
    BASENAME_VARIABLE_NAME="KNIME2GRID_VAR_${FILE_INDEX}"
    for input_file in `tar tfz ${input_port}`; do
      # use basename and index to rename the file as its written to stdout
      tar xOfz ${input_port} ${input_file} > ${FILE_INDEX}_${!BASENAME_VARIABLE_NAME}
      FILE_INDEX=$(expr ${FILE_INDEX} + 1)
    done
    ARCHIVE_INDEX=$(expr ${ARCHIVE_INDEX} + 1)
  done
fi

${EXECUTABLE} ${COMMAND_LINE_PARAMETERS}

# compress the multi-file outputs
if [ -n "$OUTPUT_PORTS_WITH_FILELIST" ]; then
  for output_port in ${OUTPUT_PORTS_WITH_FILELIST}; do
    echo "compressing outputs for ${output_port}"
    tar cfz ${output_port} *_${output_port:0:(-7)}
  done
fi
```

Conversion of KNIME Nodes

In order to split the execution of a KNIME workflow into individual tasks, we devised a procedure that benefits from KNIME's batch mode, introduced in Section 4.2.3. The first step is to generate a command line invocation for each of the nodes comprising the workflow. The generated command line invocations can later be used to execute the same task on a DCI. In order to generate these per–node commands, KNIME2Grid first determines whether the node in question requires a running instance of the KNIME Analytics Platform to execute.

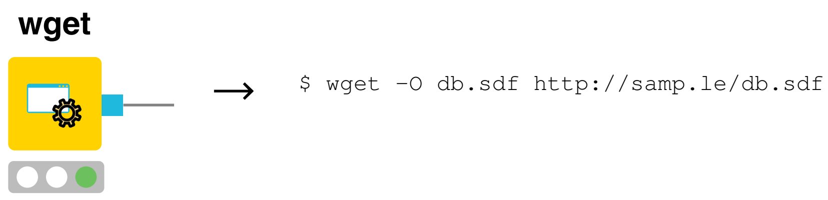
Conversion of Nodes Generated With GKN Even though KNIME Nodes created by the GKN extension interact with the KNIME Core components, they rely on an external binary: their execution is possible without the KNIME Analytics Platform. In this case, generation of an equivalent command line is somewhat trivial. However, conversion of a single node interacting with CTD–enabled tools requires two extra steps: an automatic generation of a CTD file containing the runtime parameters, and its inclusion as an input file to the converted node. This process is summarized in Figures 4.2 and 4.3. For a detailed example of how CTD–enabled tools interact with CTD files, refer to Appendix B.0.2.

Conversion of Native KNIME Nodes In order to execute a single native KNIME Node using the command line, its inputs and outputs must first be available as files, not as Data Tables. Since KNIME's batch mode is able to execute only whole workflows, KNIME2Grid automatically generates a KNIME workflow for each of the converted native KNIME Nodes. These generated workflows contain a copy of the native KNIME Node in question, whose configuration settings are replicated, along with utility KNIME Nodes to deserialize any incoming inputs and to serialize any produced outputs. The major trade–off of this approach is that there is some inherent overhead in the serialization and deserialization process. Figure 4.4 depicts this conversion mechanism.



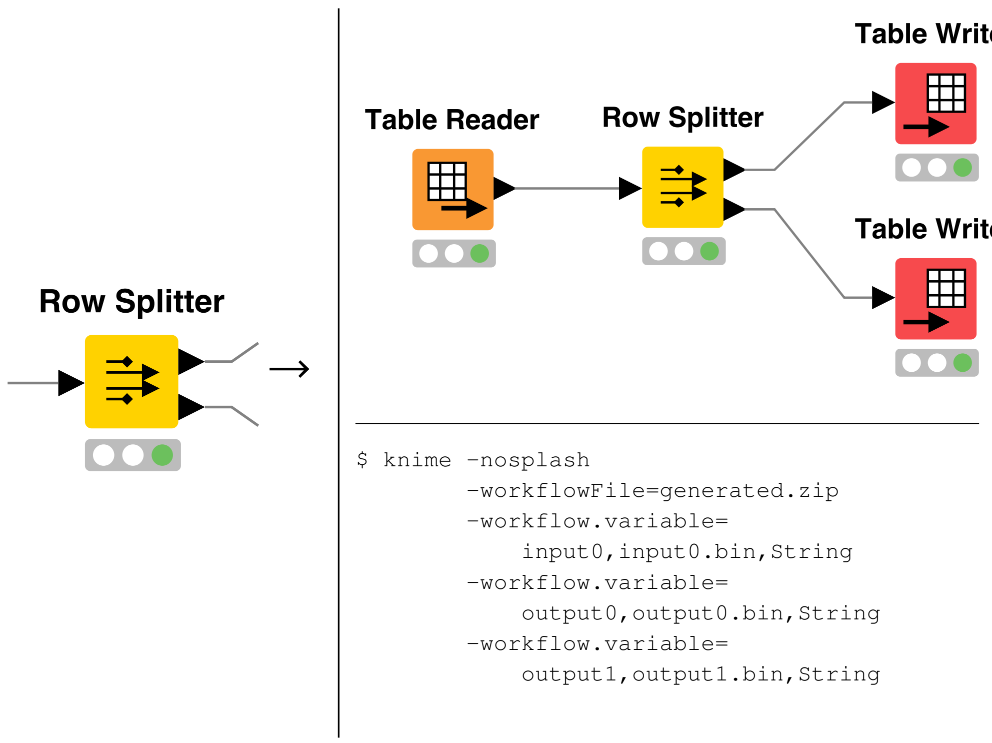
```
$ PDBCutter -par params.ctd
```

Figure 4.2: Generation of a command line to execute a CTD–enabled tool. Conversion requires the automatic creation of *params.ctd*, which must be included as an input file. Figure adapted with permission from[13].



```
$ wget -O db.sdf http://samp.le/db.sdf
```

Figure 4.3: Generation of the command line needed to execute an non–CTD–enabled tool. In this case, the full command line must be generated. Generation of the required command line relies on a CTD representing the tool. Since the tool *wget* was imported into the KNIME Analytics Platform via GKN, it can be assumed that a CTD exists. Figure adapted with permission from[13].



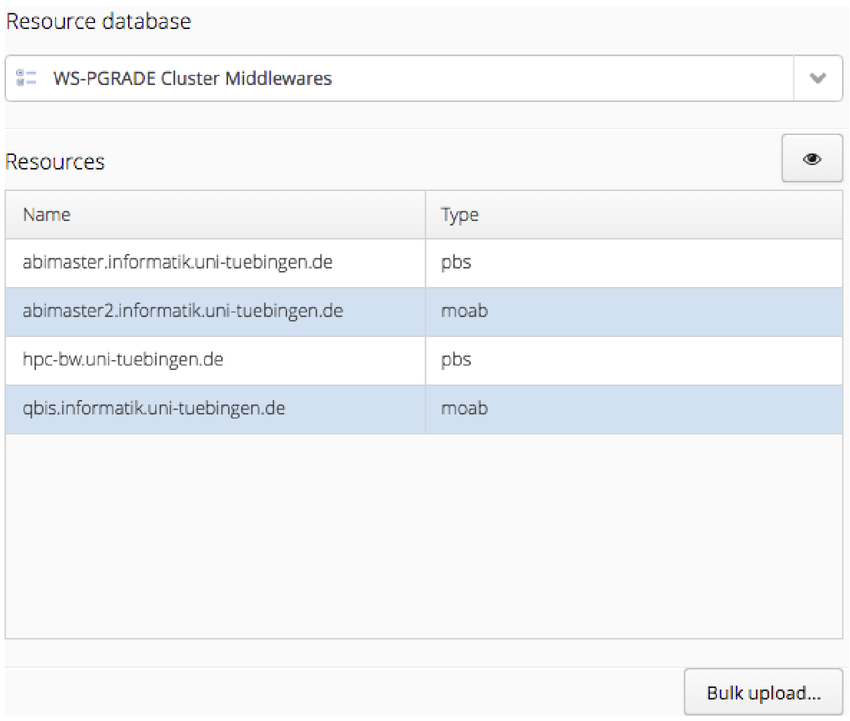
```
$ knime -nosplash
        -workflowFile=generated.zip
        -workflow.variable=
            input0,input0.bin,String
        -workflow.variable=
            output0,output0.bin,String
        -workflow.variable=
            output1,output1.bin,String
```

Figure 4.4: Conversion of native KNIME Nodes. The KNIME Analytics Platform can be started in batch mode by providing the *-nosplash* flag. KNIME2Grid generates a small workflow (top–right) able to read its inputs (i.e., *input0.bin*) into Data Tables using a *Table Reader* node. Once all inputs have been loaded, a copy of the KNIME Node (whose settings have been duplicated from the source node) will be executed. Outputs will be serialized into files (i.e., *output0.bin* and *output1.bin*) using *Table Writer* nodes. A sample command line execution of the generated workflow is shown on the bottom–right. Figure adapted with permission from[13].

4.3.2 The WS–PGRADE Extensions

Rather to limit our efforts to create a single *do–it–all* portlet, we created a basic development platform to extend WS–PGRADE/gUSE. Building upon it, we developed an application database portlet that allows users to manage applications which they can later use when converting workflows, the Application Manager Portlet.

Our proposed portlet does not concern itself with proper installation of binaries, rather, it limits itself to be a registry of applications available to gUSE, as displayed in Figures 4.14 to 4.17. It also features a REST API to provide read–only remote access to the application registry. This web service, when invoked, generates a resource file descriptor (see Listing 4.4) that can be used in external applications, e.g., in the KNIME2Grid extension.



Resource database

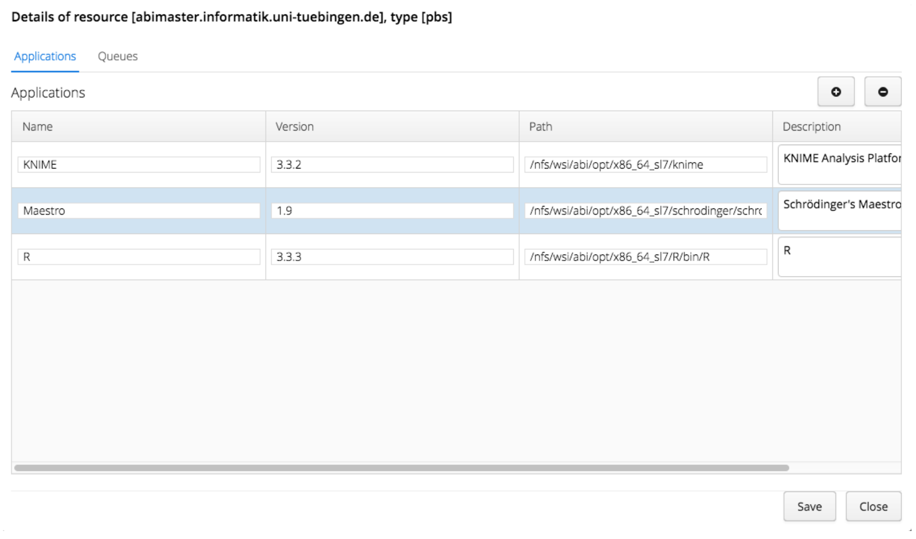
WS-PGRADE Cluster Middlewares

Resources

Name	Type
abimaster.informatik.uni-tuebingen.de	pbs
abimaster2.informatik.uni-tuebingen.de	moab
hpc-bw.uni-tuebingen.de	pbs
qbis.informatik.uni-tuebingen.de	moab

Bulk upload...

Figure 4.14: View of the available batch queueing systems. Pressing the top–right button will display a dialog where users can manage applications associated with the currently selected resource and see a list of the available queues (see Figure 4.15). The button located on the bottom–right allows users to add or edit several applications at once by uploading an XML file similar to the one presented in Listing 4.4.



Details of resource [abimaster.informatik.uni-tuebingen.de], type [pbs]

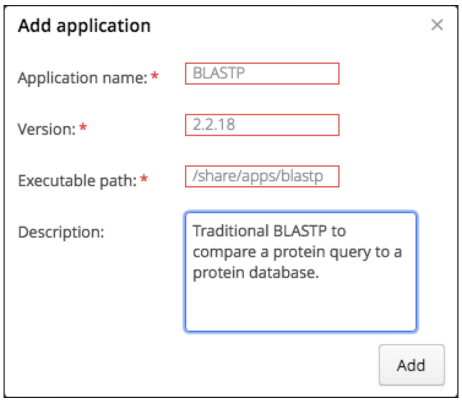
Applications Queues

Applications

Name	Version	Path	Description
KNIME	3.3.2	/nfs/wsi/abi/opt/x86_64_sl7/knime	KNIME Analysis Platfor
Maestro	1.9	/nfs/wsi/abi/opt/x86_64_sl7/schrodinger/schrc	Schrödinger's Maestro
R	3.3.3	/nfs/wsi/abi/opt/x86_64_sl7/R/bin/R	R

Save Close

Figure 4.15: View of the applications and queues associated with a specific batch queueing system. Users can edit, insert, and delete applications using the intuitive controls.



Add application

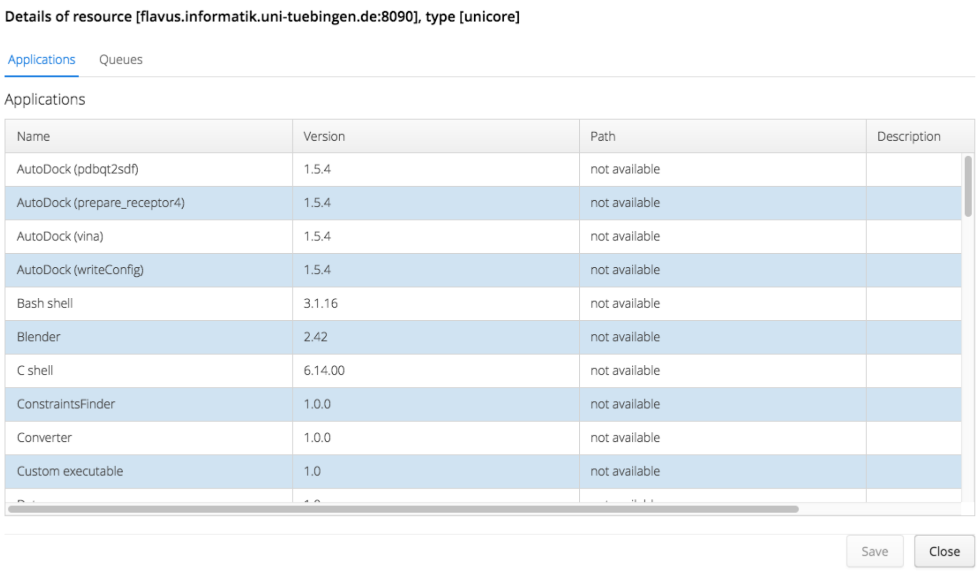
Application name: * BLASTP

Version: * 2.2.18

Executable path: * /share/apps/blastp

Description: Traditional BLASTP to compare a protein query to a protein database.

Figure 4.16: Dialog that permits users to manually associate an application to a specific batch queueing system. Optionally, users can *bulk add* applications via uploading an XML file, as depicted in Figure 4.14.



Details of resource [flavus.informatik.uni-tuebingen.de:8090], type [unicore]

Applications

Name	Version	Path	Description
AutoDock (pdbqt2sdf)	1.5.4	not available	
AutoDock (prepare_receptor4)	1.5.4	not available	
AutoDock (vina)	1.5.4	not available	
AutoDock (writeConfig)	1.5.4	not available	
Bash shell	3.1.16	not available	
Blender	2.42	not available	
C shell	6.14.00	not available	
ConstraintsFinder	1.0.0	not available	
Converter	1.0.0	not available	
Custom executable	1.0	not available	

Figure 4.17: A view of available UNICORE applications using the application database portlet. The controls to perform editions are not available due to UNICORE exposing its application registry as read–only.

Listing 4.4: Sample XML resources file containing available resources, queues and applications. To improve usability, we use a single format across all related operations (i.e., bulk uploads and REST API output as described in this section, as well as the job configuration screen included in the workflow export wizard previously shown in Figures 4.9 and 4.10).

```xml
<resources>
  <resource name="abimaster2.informatik.uni-tuebingen.de" type="moab">
    <application name="Maestro" version="1.9" description="Maestro"
                 path="/share/schroedinger/bin/maestro" />
    <application name="R" version="3.3.3" description="R"
                 path="/share/R/bin/R" />
    <queue name="default"/>
    <queue name="fast"/>
  </resource>
  <resource name="abimaster.informatik.uni-tuebingen.de" type="pbs">
    <application name="KNIME" version="3.3.1" description="KNIME"
                 path="/share/knime/knime" />
    <queue name="default"/>
  </resource>
</resources>
```

Overall Structure

Even though the Application Manager Portlet is currently the only portlet based on our proposed framework, developers of supplementary portlets would benefit from the components comprising our framework. We structured our proposed development framework as follows:

- *core–lib*: contains the core components, utility methods, and data structures that are used throughout the extensions and the *ui–components* library.
- *ui–components*: our custom Vaadin widgets on which the WS–PGRADE portlets depend on are contained in this module.
- *application–manager–portlet*: contains the code specific for Application Manager Portlet and the REST API; future, supplementary portlets would be at this same level on the development stack.

The relationship between these projects and other libraries is shown in Figure 4.18. Each of the components we created is delivered as a separate Apache Maven artifact, allowing simplification of two tasks that, were they not automated, development could quickly be stunted: assembly and dependency management.

Overall Design

Similar to the implementation provided in KNIME2Grid, we also followed the dependency injection technique to resolve dependencies only during construction time. In order to further raise the level of abstraction, each relevant component has been modeled by an interface and a set of separate implementations. Configurable dependencies are defined in the *Settings*

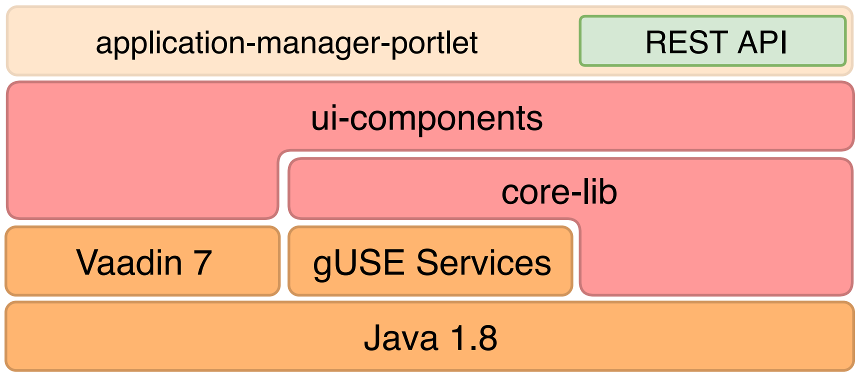
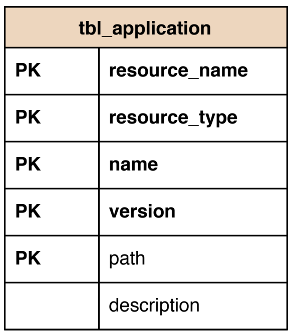


Figure 4.18: Dependencies between the different software components that comprise our WS–PGRADE extensions. *core-lib* depends on the gUSE services and standard Java classes (e.g., for database access), but is independent on Vaadin. The elements contained in *ui-components*, on the other hand, depend on Vaadin and also on *core-lib*. *application-manager-portlet* builds upon *core-lib* and *ui-components*, including a REST API.

singleton class, which is then used to provide the required dependencies of components at construction time.

Since WS–PGRADE/gUSE already requires MySQL for its proper operation, we chose to implement our application registry using MySQL (refer to Figure 4.19 for its schema). During its initialization, the Application Manager Portlet will automatically create this database if needed. Additionally, users are not required to provide additional credentials: our portlet queries the Information System component to obtain these. We include the REST API as part of Application Manager Portlet to provide consistency for users, developers and portal administrators: access to both the REST API and our the Application Manager Portlet can be done using the same base URL.



tbl_application	
PK	**resource_name**
PK	**resource_type**
PK	**name**
PK	**version**
PK	path
	description

Figure 4.19: Schema of the application database. DCI Bridge maintains a registry for resources and their associated queues. A single table is sufficient to add an application database. All fields but *description* comprise the primary key.

As a feature to speed–up development and automated testing, we created *mock* components containing in–memory data structures whose content is independent of gUSE services. These data structures are automatically populated with synthetic data during initialization. To enable the usage of these mock components, a single property on the deployment descriptor file needs to be modified (as depicted on the top–right corner of the diagram shown in Figure 4.20). After a successful assembly and deployment, any portlet using these mock components is ready to use, thus bypassing the need to restart the Liferay portal.

Deployment Script

Development or maintenance of our WS–PGRADE extensions can become a stultifying task after a handful of cycles of coding, WAR file deployment, and manual restarting of the Liferay portal. In order to expedite development efforts, we created a support Apache Ant script that automates building and deployment of WAR files on a running WS–PGRADE instance. This script depends on a simple configuration file to read values such as location and credentials of the server on which portlets will be deployed. This procedure, along a sample configuration file and the support script, are detailed in Appendix B.0.7.

Initialization

Our WS–PGRADE extensions define *hook–ups* to perform initialization and clean–up tasks by registering an implementation of the *ServletContextListener* interface in the deployment descriptor file (i.e., the *web.xml* file). Our implementation is realized in *WorkflowConversion-ContextListener*, a class that properly initializes the singleton *Settings* class.

Once the single instance of the *Settings* class has been created, the Servlet Container proceeds to instantiate all pertinent *Servlets* as declared in the deployment descriptor. We created *WorkflowConversionUI*, an abstract class designed to provide uniform access to basic Vaadin mechanisms and other initialization routines. The Application Manager Portlet provides a custom extending class, *ApplicationManagerUI*, and is responsible to prepare the graphical content that will ultimately be displayed. Tasks requiring a *lazy initialization* are handled in the *WorkflowConversionUI* abstract class we provide.

Following the guidelines for multithreading programming detailed by Goetz and Peierls[89] (i.e., immutable objects are always thread–safe), instances of the *Settings* class are immutable objects. This is an important aspect in multi–user, concurrent web–based systems such as WS–PGRADE. We thus applied a modified version of the *builder* design pattern as described by Gamma et al.[82] in the *SettingsBuilder* class to avoid possible race conditions.

Figures 4.20 and 4.21 show the class and sequence diagrams of the Application Manager Portlet, respectively, highlighting the components involved during the initialization.

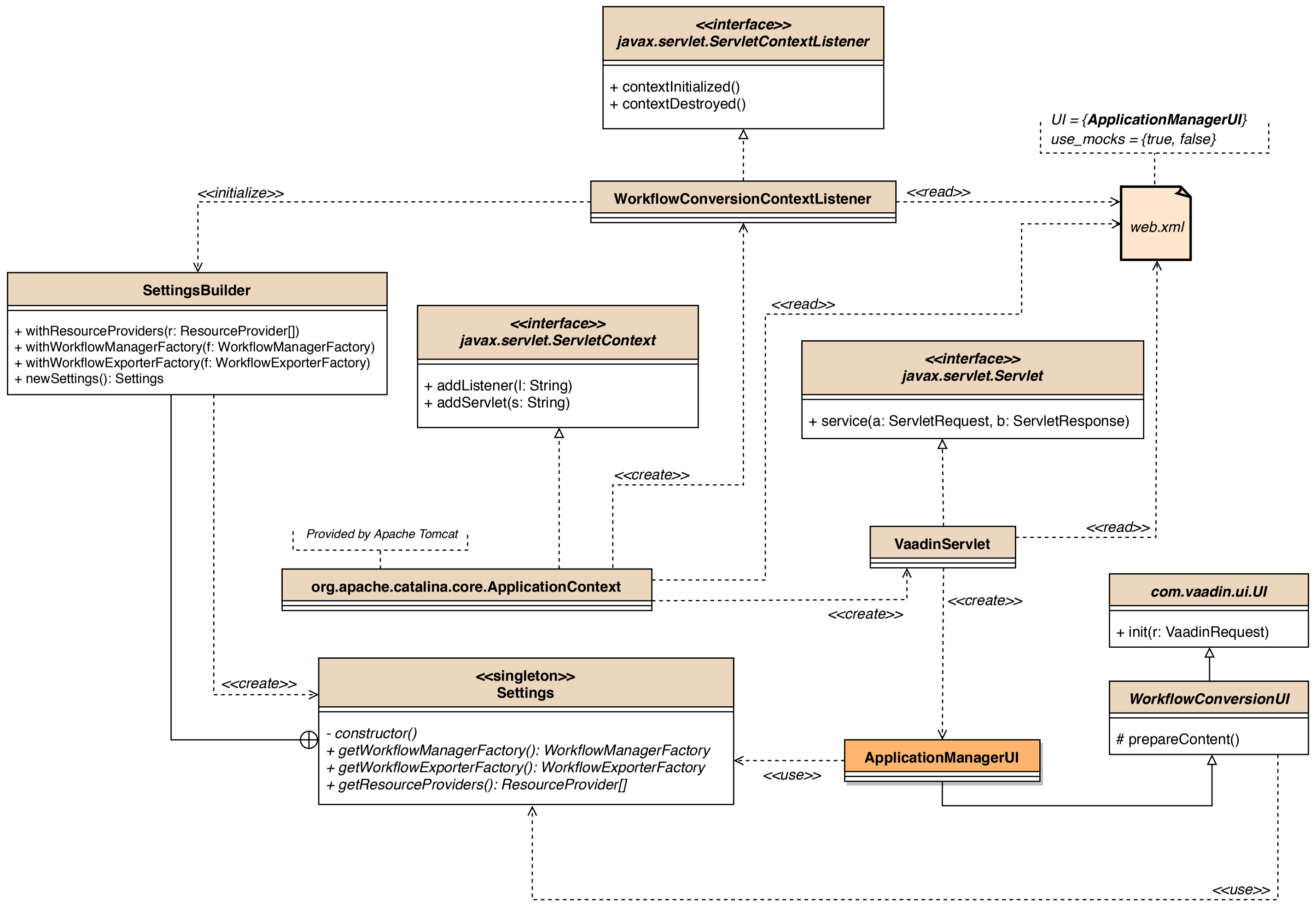

<<interface>>
javax.servlet.ServletContextListener
+ contextInitialized()
+ contextDestroyed()
UI = {ApplicationManagerUI}
use_mocks = {true, false}
<<initialize>>
WorkflowConversionContextListener
<<read>>
web.xml
SettingsBuilder
+ withResourceProviders(r: ResourceProvider[])
+ withWorkflowManagerFactory(f: WorkflowManagerFactory)
+ withWorkflowExporterFactory(f: WorkflowExporterFactory)
+ newSettings(): Settings
<<interface>>
javax.servlet.ServletContext
+ addListener(l: String)
+ addServlet(s: String)
<<read>>
<<interface>>
javax.servlet.Servlet
+ service(a: ServletRequest, b: ServletResponse)
<<create>>
Provided by Apache Tomcat
VaadinServlet
<<read>>
org.apache.catalina.core.ApplicationContext
<<create>>
<<create>>
com.vaadin.ui.UI
+ init(r: VaadinRequest)
<<create>>
<<singleton>>
Settings
- constructor()
+ getWorkflowManagerFactory(): WorkflowManagerFactory
+ getWorkflowExporterFactory(): WorkflowExporterFactory
+ getResourceProviders(): ResourceProvider[]
WorkflowConversionUI
# prepareContent()
ApplicationManagerUI
<<use>>
<<use>>

Figure 4.20: Class diagram of the Application Manager Portlet highlighting the interactions between components needed during initialization. The portlet can be set on *development mode* by modifying a single property on the *web.xml* file, as previously described in Section 4.3.2.

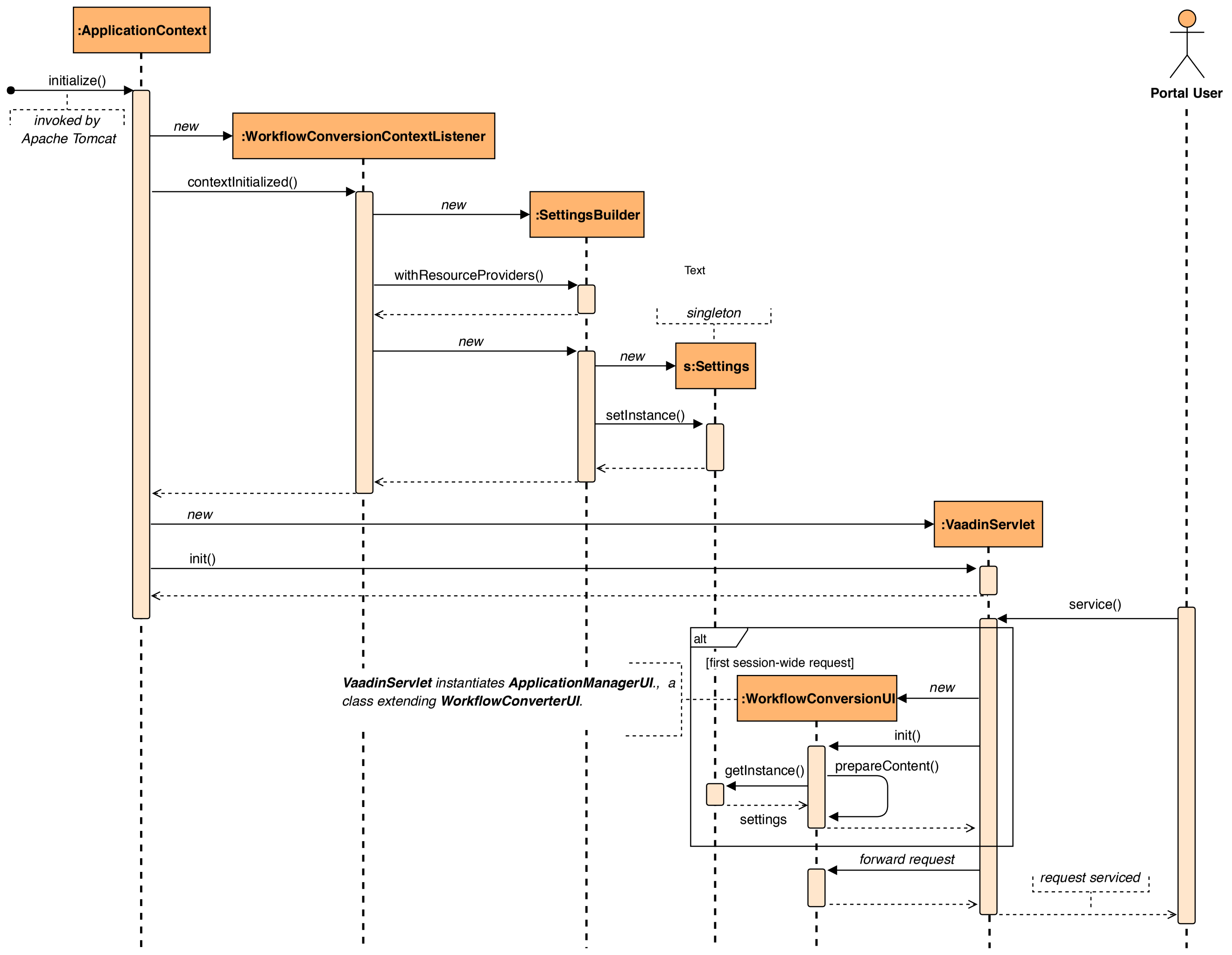


Figure 4.21: Sequence diagram of the initialization routine of the Application Manager Portlet. *ApplicationContext* is instructed to initialize the portlet. This process includes creating an instance of the *WorkflowConversionContextListener* class, who in turn is responsible to initialize the *Settings* class and to set the singleton instance to be used throughout the life cycle of the portlet.

Abstraction of Applications: *Resources* and *ResourceProviders*

We modeled each of the available computing resources using the class *Resource* (see Figure 4.22). Individual instances of the *Resource* class represent computing resources configured in gUSE, each containing a number of applications and queues (modeled by the *Application* and *Queue* classes, respectively).

Similar to the usage of the *Settings* singleton, instances of these classes must also be immutable, due to the fact that they will be shared among concurrent threads. We also applied the same modified version of the *builder* pattern.

In order to offer a clean and consistent access to the available computing resources, we defined an interface, *ResourceProvider*, defining interaction with components that are able to manage applications. Implementations of this interface specify whether it is possible to edit applications, and they must provide a name to identify the provider, an initialization method, and a list of *Resources*. The *UNICOREResourceProvider* implementation, for instance, uses the UNICORE remote API and displays a read–only view of resources and applications (refer to Section 4.2.3 for an explanation of this behavior). The *ClusterResourceProvider* class, on the other hand, queries the Information System component to obtain the list of the active batch job processing computing resources and their associated queues. It then uses this set of resources to associate applications using our application registry. The REST API benefits from this design: it accesses all available *ResourceProviders* to generate an XML document containing all applications visible to the gUSE without having to directly rely on gUSE components or the UNICORE remote API.

Generic Graphical Components

We implemented a series of configurable graphical components using Vaadin widgets to allow any portlet to easily integrate acquire the look and feel of the hosting Liferay portal. The Application Manager Portlet's GUI was put together using these components, the most important of them being the generic tables on which users can interact with our application registry. These tables contain configurable, additional controls to add, edit, and remove displayed elements. This sort of situation perfectly aligns itself with the intent of *abstract factories*: provide an interface for creating families of related objects without specifying their concrete class[82].

In order to implement this design pattern we created both an interface and a direct implementation in the form of an abstract class as generic components (*TableWithControls* and *AbstractTableWithControls*, respectively). We used Java generics to encapsulate the common functionalities and to increase the application domain of our graphical components. The *AbstractTableWithControls* generic abstract class contains the implementation of the core features allowing elements to be inserted, edited, and removed in a table. Each row is the textual representation of an item, each cell displaying the value of a specific property of said item.

The concrete classes are only responsible to provide a proper bidirectional translation between the contents of a row and their represented elements (e.g., application, resource, queue), thus simplifying development efforts. Furthermore, in the interest of enforcing the *abstract factory* pattern, constructors of these tables were hidden as private methods, forcing usage of the provided builder factory static inner class.

Since these concrete classes are a transitive implementation of the *TableWithControls* generic interface, the *ApplicationManagerUI* will be able to further abstract itself from the concrete classes by interacting with an interface, rather than communicating with specific concrete classes.

The class diagram presented in Figure 4.23 highlights the relationships of the most relevant entities of our graphical components.

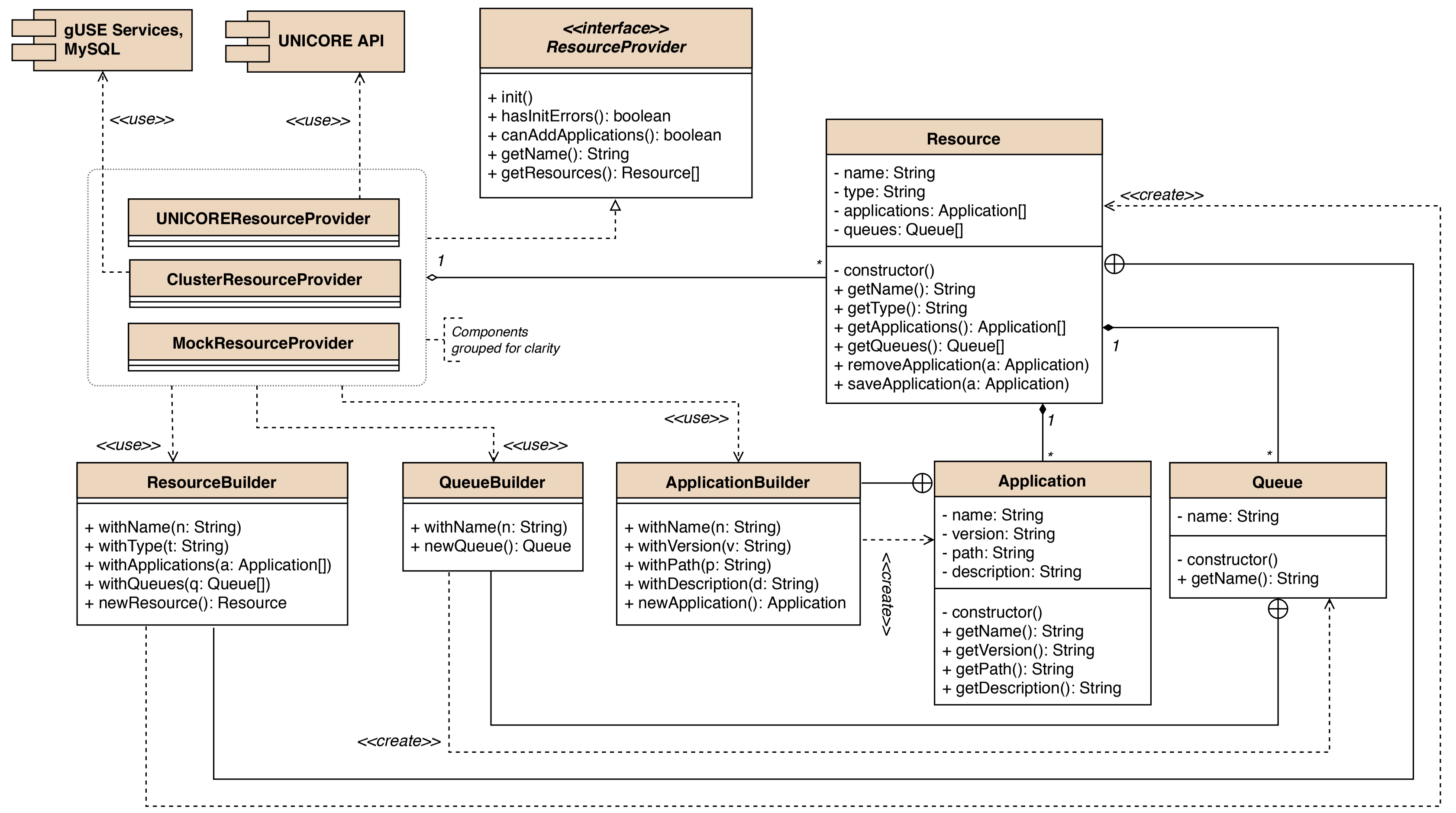


Figure 4.22: Component diagram highlighting interaction between *Resources* and *ResourceProviders*.

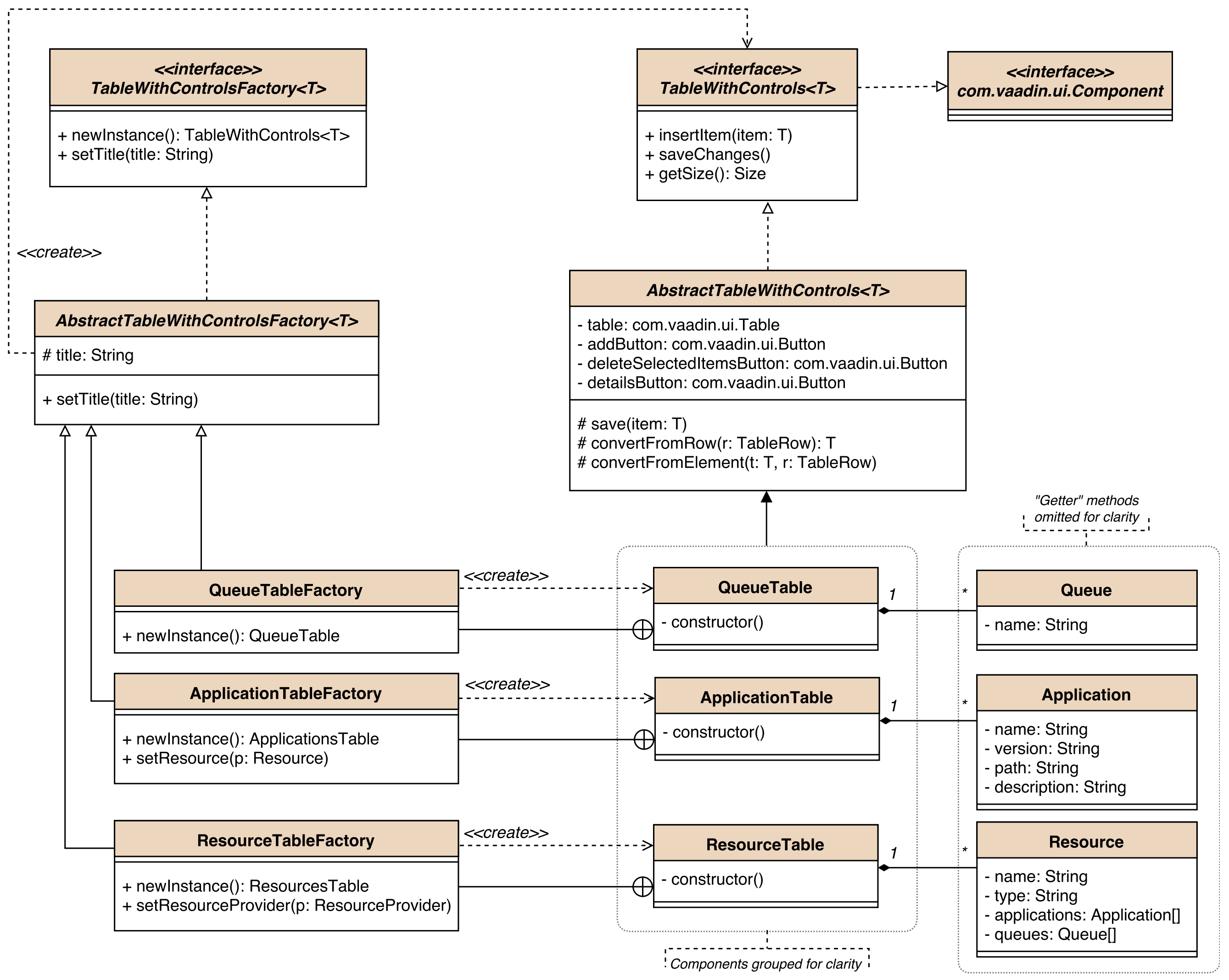

<<interface>>
TableWithControlsFactory<T>
+ newInstance(): TableWithControls<T>
+ setTitle(title: String)
<<interface>>
TableWithControls<T>
+ insertItem(item: T)
+ saveChanges()
+ getSize(): Size
<<interface>>
com.vaadin.ui.Component
<<create>>
AbstractTableWithControlsFactory<T>
# title: String
+ setTitle(title: String)
AbstractTableWithControls<T>
- table: com.vaadin.ui.Table
- addButton: com.vaadin.ui.Button
- deleteSelectedItemsButton: com.vaadin.ui.Button
- detailsButton: com.vaadin.ui.Button
# save(item: T)
# convertFromRow(r: TableRow): T
# convertFromElement(t: T, r: TableRow)
"Getter" methods
omitted for clarity
QueueTableFactory
+ newInstance(): QueueTable
<<create>>
QueueTable
- constructor()
1
*
Queue
- name: String
ApplicationTableFactory
+ newInstance(): ApplicationsTable
+ setResource(p: Resource)
<<create>>
ApplicationTable
- constructor()
1
*
Application
- name: String
- version: String
- path: String
- description: String
ResourceTableFactory
+ newInstance(): ResourcesTable
+ setResourceProvider(p: ResourceProvider)
<<create>>
ResourceTable
- constructor()
1
*
Resource
- name: String
- type: String
- applications: Application[]
- queues: Queue[]
Components grouped for clarity

Figure 4.23: Class diagram displaying the relationships between the concrete tables and their factories.

4.3.3 Availability

KNIME2Grid is available at `https://github.com/WorkflowConversion/KNIME2Grid` and is distributed under the MIT license. The presented WS–PGRADE extensions can be downloaded from `https://github.com/WorkflowConversion/WS-PGRADE-Extensions` (also distributed under the MIT license).

Furthermore, `http://workflowconversion.github.io` contains a summary we prepared to highlight the work presented both in this chapter and in Section 3.3.

4.3.4 Use Cases

We have argued that usage of HPC resources speeds up computation and retrieval of scientific results. Runtime of a converted workflow will depend on factors outside the scope of this work, e.g., latency of storage systems and hardware specifications. The work presented in Sections 4.3.1 and 4.3.2 is able to generate workflows fully compatible with WS–PGRADE/gUSE, but performance and running time of workflows ultimately depends on the capabilities and current load of available HPC resources.

Combined usage of our presented extensions to WS–PGRADE and the KNIME Analytics Platform adds value to the field of life sciences: workflows designed and tested on a desktop computer can be—with minimal required input—executed on HPC resources.

Each presented workflow was designed and tested using the KNIME Analytics Platform. Workflows were then converted using the workflow export wizard contained in KNIME2Grid. In order to obtain a resource descriptor file (refer to Listing 4.4 for a sample file), the REST API was remotely accessed by the introduced workflow export wizard. The generated workflows were then imported on a WS–PGRADE/gUSE instance, their concrete was then submitted using WS–PGRADE's standard user interface. The WS–PGRADE/gUSE installation had access to a batch queueing system with the Moab Workload Manager on which all mentioned command line tools, as well as the KNIME Analytics Platform, were installed.

Structural Bioinformatics

Molecular docking is a technique in structural bioinformatics with direct applicability to the drug discovery pipeline. Fundamentally, molecular docking algorithms attempt to *fit* molecules (ligands) in advantageous conformational isomers using their topographic features into binding sites of proteins (receptors), performing free energy calculations of the overall system with the purpose of finding a local minimum. The vast solution space and complexity of the operations do not allow for an exhaustive search for the best global conformation, rendering molecular docking a field where combinatorial optimization is often applied[84,90].

The presented workflow starts by reading an input PDB file containing a receptor whose binding site already holds a well–known ligand. *PDBCutter* then separates the input complex into its constituting elements. The extracted receptor, still requiring some processing, is forwarded to *ProteinProtonator*, where, based on a given pH value, protons (H^+) will be inserted[64].

Placement of the reference ligand is extremely relevant for docking algorithms. In this case, a tridimensional grid is built around the reference ligand to assist in the computation of the free energy of the system (a task performed by the collaboration of *PocketDetector* and *GridBuilder*)[64].

The second input is a structure data file (SDF) where a list of candidate ligands is provided. These compounds, if deemed favorable by the algorithm, could then be used in later phases of the drug discovery pipeline. This ligand database is received by *Ligand3DGenerator*, where each input compound will be transformed into a tridimensional conformation that is ready for docking[64].

IMGDock (short for iterative multi–greedy docking) consolidates the information about the binding site, along with the processed list of compounds. It will then *dock* each of the given ligands into the binding site of the receptor to compute a score. The list of scores is then collected, sorted and written to an output file by *DockResultMerger*[64].

The workflow was composed using the CTD–enabled CADDSuite tools, which were integrated into the the KNIME Analytics Platform using GKN (a process we described in Section 4.2.3). Conversion of these nodes was performed as shown in Figure 4.2. This scenario presents the most trivial type of conversion for our extensions: executing command line tools on DCIs does not require a running instance of the KNIME Analytics Platform. Figures 4.24 and 4.25 present the original and converted workflows, respectively.

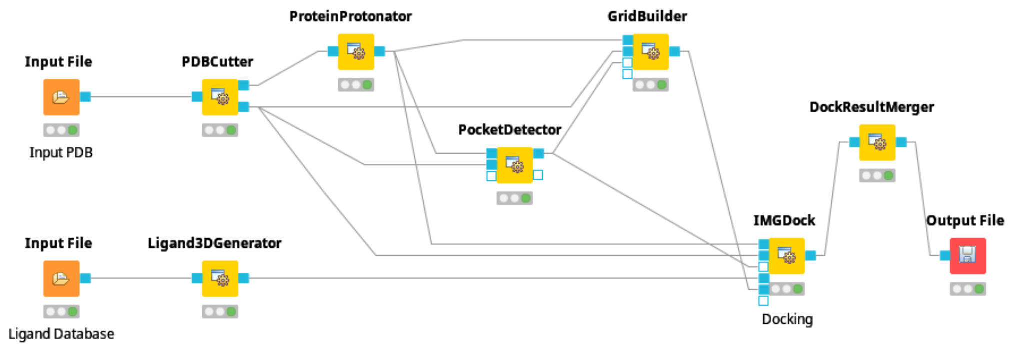


Figure 4.24: Docking workflow on the KNIME Analytics Platform. Adapted with permission from[13].

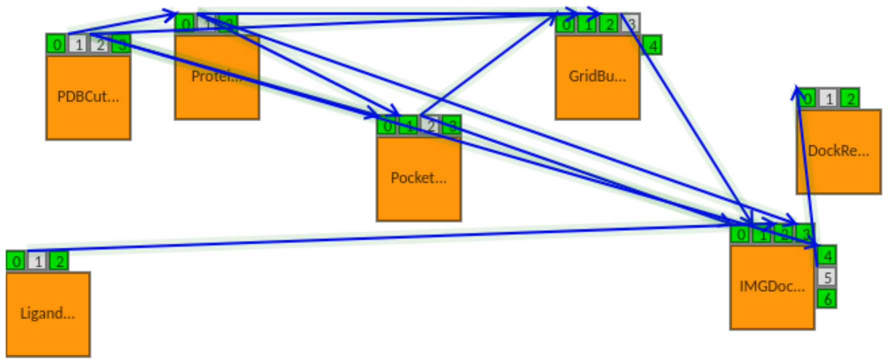


Figure 4.25: Converted docking workflow on WS–PGRADE. Since all tools are CTD–enabled, an additional input port was created on each node to provide a pre–configured CTD file as an input.

Immunoinformatics

The exponential growth of biological data relevant to immunology research, coupled with the rapid increase of clinical and epidemiologic information available in medical records and scientific literature, motivated scientists interested in pathogenesis and immune function to lay the foundations of immunoinformatics. With applications varying from basic immunological and translational research to oncological research, immunoinformatics methods have ever since become a vital part of biomedical research[91,92].

The complexity of the required methods, lack of standardized interfaces and data formats usually prohibits the use of different tools in the same workflows. Although this has prompted creation of web–based workbenches that provide access via unified interfaces, factors such as data volume and legal considerations (e.g., restrictions on sharing patient data) might render these approaches unusable. *ImmunoNodes*, part of the KNIME Community Contributions, was developed to offer researchers a unified platform consisting of a toolbox where each individual *ImmunoNode* carries out a specific analysis or computation in immunoinformatics[92].

The presented workflow in Figure 4.26 implements a population–based vaccine design pipeline. This process starts by reading specific geographical regions (or populations) of interest with the purpose of producing a list of human leukocyte antigen (HLA) alleles with their corresponding occurrence probability, a task performed by the *AlleleFrequency*. A further input, a file containing well–known pathogens in the form of protein sequences is provided to *Epitope-Prediction*. This node generates peptides off the given input and produces a file containing the predicted binding affinities of these, along with the selected HLAs. Finally, *EpitopeSelection*, given a user–defined number of epitopes from the candidate pool, writes these together with other statistics into an output file[92].

Each ImmunoNode was generated by GKN and uses Docker containerization to interact with arbitrary external command line tools contained in the Framework for Epitope Detection (FRED2)[92]. However, the toolset is not CTD–enabled, therefore conversion of each ImmunoNode was performed as shown in Figure 4.3. The converted workflow is shown in Figure 4.27.

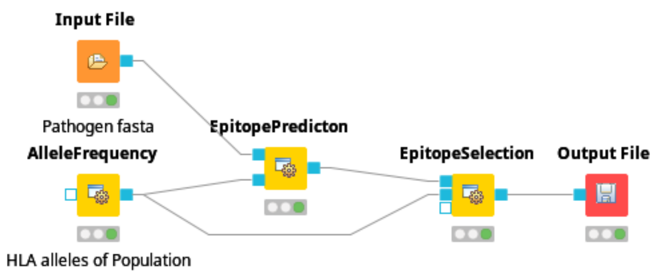


Figure 4.26: Population–based vaccine design workflow on the KNIME Analytics Platform using ImmunoNodes. Figure adapted with permission from[92]. Copyright 2017 Schubert et al.[92].

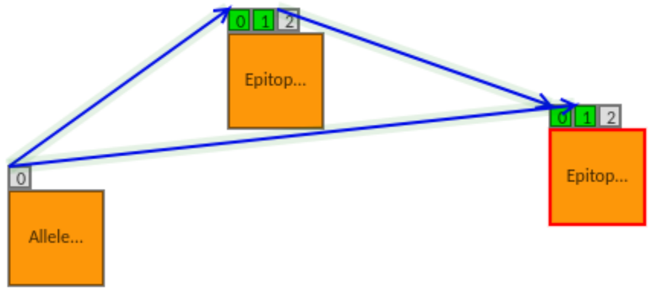


Figure 4.27: Converted vaccine design workflow on WS–PGRADE.

Metabolomics

Metabolomics is a set of methods based on mass spectrometry data with the intent of evaluating the entirety of a metabolite sample. Common applications of metabolomics include discovery of mechanisms behind diseases, analysis of chemicals and their byproducts in waste water, and cancer type identification. Compared to other so–called *omics* techniques (e.g., proteomics and transcriptomics), metabolomics finds itself closer to the actual biochemical processes, making it promising for development of biomarkers. Studies interested in comparative metabolite concentrations often resort to label–free quantification approaches: independence from chemical labels allows direct comparison of small molecules across an arbitrary number of samples. The need to concurrently evaluate considerable amounts of data (often hundreds of gigabyte–sized samples), while numbers and sizes of available data are steadily increasing, urges researchers to utilize distributed computing approaches.

The presented metabolomics workflow performs biomarker discovery using a detection label–free quantification method for small molecules using the OpenMS KNIME Nodes extension (also part of the KNIME Community Contributions)[63] together with other KNIME Nodes.

Previous to executing the workflow, some preparations must be performed: data reduction by means of *peak picking* and conversion from closed, vendor–specific formats to the open *mzML* data format.

The pipeline shown in Figure 4.28 starts with quantification, a process that consists of sample–specific feature detection (i.e., finding convex hulls and their respective centroids of analyte mass traces) followed by temporal alignment of samples and quantification of features across samples. These tasks are performed by the collaboration of *FeatureFinderMetabo*, *MapAlignerPoseClustering* and *FeatureLinkerUnlabeledQT*, all three part of OpenMS. Furthermore, feature detection is a process that is performed inside a parametric sweep section, i.e., it is enclosed between *ZipLoopStart* and *ZipLoopEnd* nodes.

Downstream small molecule identification was performed via mass–based search in the Human Metabolome Database by *AccurateMassSearch*, this being the last OpenMS–based task executed in the pipeline. Analytes whose abundances vary significantly after false discovery rate correction are annotated with the mass–based identifications and exported to a spreadsheet. This last analysis is performed by a combination of standard KNIME Nodes and R scripts (the *R Snippet* KNIME Node directly interacts with a local R installation).

Contrasting to the other presented use cases, this workflow contains a mixture of the three type of KNIME Nodes we identified, as presented in Figures 4.2 to 4.4, as well as a parametric sweep workflow pattern. Figure 4.29 depicts the converted workflow.

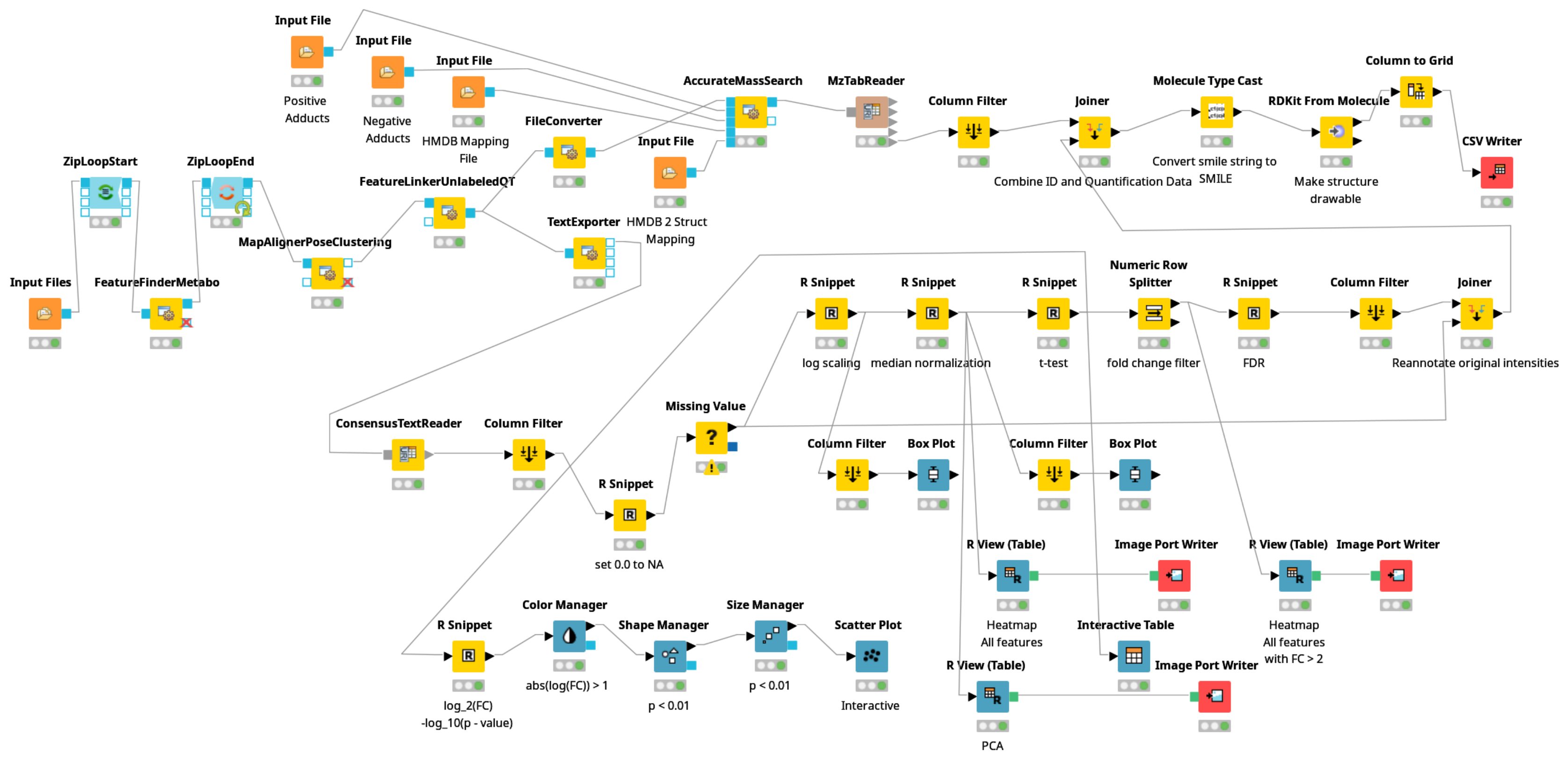


Figure 4.28: Biomarker discovery pipeline on the KNIME Analytics Platform. Figure adapted with permission from[13].

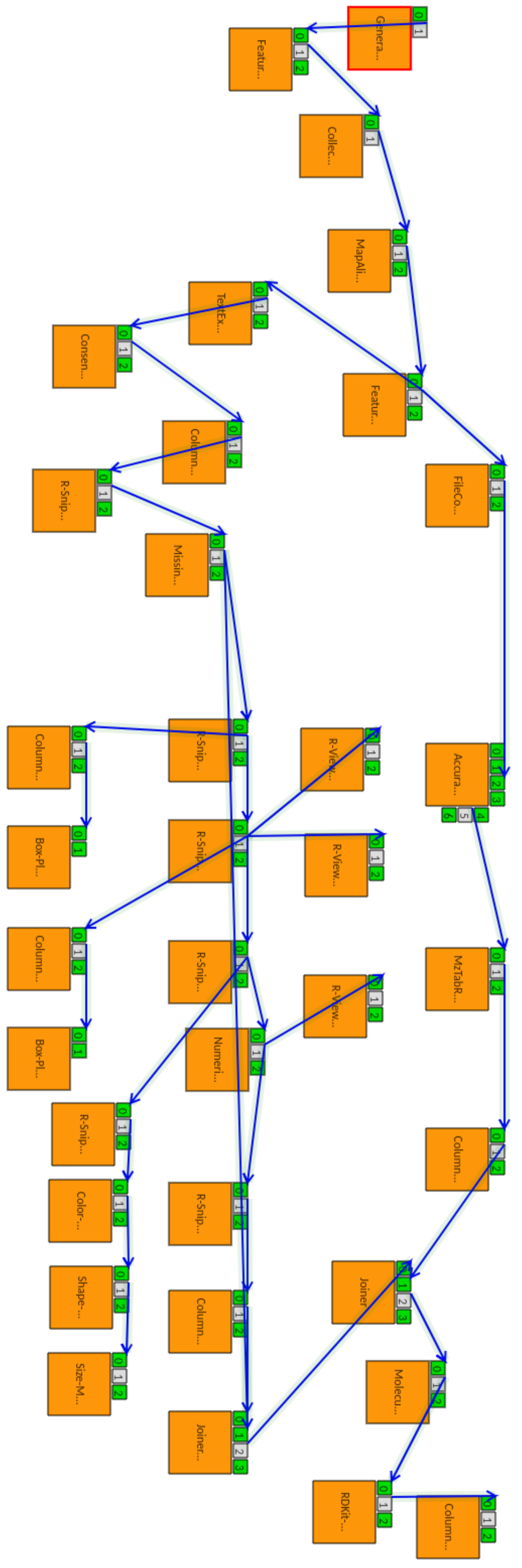


Figure 4.29: Converted biomarker discovery workflow on WS–PGRADE.

4.4 Discussion

Researchers and developers have made meaningful efforts to increase interoperability between other workflow management systems. As presented by Grunzke et al.[17], execution of KNIME workflows using UNICORE is a simple task: once users are satisfied with the design of their KNIME workflow, they use the standard KNIME Analytics Platform export dialog to save it in a submission directory that is readable by the UNICORE Server, where it will be automatically executed.

Taverna offers simplified access to freely available resources from life science institutions such as the European Bioinformatics Institute (EBI) and the National Center of Biotechnology Information (NCBI). Tavaxy and Taverna 2–Galaxy are initiatives that allow integration of Taverna workflows into Galaxy sub–workflows, offering support for cloud computing[15,16,93]. Similarly, the SHIWA Simulation Platform, comprised of several engines, offers an integrated environment on which users can share executable workflows using its workflow repository, allowing users to build *meta–workflows* composed of workflows for any of the supported engines[14].

These projects have one particular aspect in common: they only offer coarse–grained interoperability. Our fine–grained approach delivers a more flexible solution and produces a true node–by–node translation of a workflow. Fine–grained approaches allow to individually optimize each of the converted tasks, providing a higher degree of scalability. Another aspect that coarse–grained interoperability does not fully address is troubleshooting of faulty workflows: whole workflows are treated as nodes, so identifying a problem on a complex pipeline might be a tedious task. In contrast, our proposed extensions allow the target engine to pinpoint the source of an error. Furthermore, the SHIWA Simulation Platform, Galaxy, Taverna, and UNICORE offer different levels of support of various DCIs. However, WS–PGRADE/gUSE, via its DCI Bridge component, is able to communicate with most major DCIs. Also, as presented in Section 4.2.3, it is possible to develop new DCI Submitters for DCIs that are not supported.

Finally, using the KNIME Analytics Platform as a source engine on which users would design workflows offers the creation of complex pipelines on a user–friendly environment, as presented in Section 4.3.4. Integrating external command line tools into the KNIME Analytics Platform is a simple task, as discussed in Section 4.2.3, but integrating single KNIME nodes on other workflow engines is not a trivial task—as described in Section 4.3.1—yet our proposed solution performs this task automatically.

Chapter 5

Conclusion and Outlook

Each workflow management system we have studied addresses specific concerns of the community it was designed for. Throughout this work, we have argued that development efforts should be directed towards integration of already existent workflow engines with the intent of leveraging their combined features.

Recognizing the near ubiquity of command line tools in life sciences, we created CTD-Converter, a Python–based framework that boosts integration of such tools into workflow engines. This was achieved by usage and translation of platform–independent tool representations, CTDs. We also combined the features of WS–PGRADE/gUSE and the KNIME Analytics Platform to provide fine–grained workflow interoperability, enabling users to create and test workflows on a user–friendly platform, while transparent workflow execution occurs on distributed HPC resources.

The speed at which workflow technologies and distributed computing evolves can be overwhelming for workflow developers. Innovative solutions present at the commencement of our research are now posing themselves as mature, dominant platforms, e.g., containerization and cloud computing. Having this aspect in mind, our proposed implementations were built following software development techniques that lend flexibility and resilience: generation of task representations for additional workflow engines calls for insertion of a small Python module into CTDConverter, adding supplementary output format to the KNIME2Grid extension requires implementation of a single class.

Nevertheless, there are still areas of improvement. As presented, executing individual KNIME Nodes as command line tools has certain overhead associated to it: serialization and deserialization of Data Tables is an expensive input/output (I/O) operation. This could be alleviated by dynamically creating sub–workflows out of sequential sections composed exclusively of native KNIME Nodes. Instead of loading and writing the inputs and outputs of each converted node, these operations would occur once per identified sub–workflow section.

Similarly, conversion of KNIME nodes that loop over a list of input files could also be optimized. WS–PGRADE encloses parameter sweep sections between an output generator port and an input collector port. Having a dedicated job on a cluster to implement parametric sweeps is an approach that suffers of unnecessary I/O overhead.

A functionality in the KNIME Analytics Platform that our converter does not support is data streaming. The *KNIME Streaming Executor* feature allows for concurrent node execution by immediately providing partial outputs as inputs to downstream nodes. This feature cuts down I/O operations and reduces the memory footprint of nodes. Since gUSE workflows rely almost exclusively on channeling of files, implementation of a similar feature—while attractive—poses a paramount challenge.

There are some KNIME Nodes whose conversion, as currently implemented, poses no value for researchers. Visualizing results using KNIME Nodes on a desktop computer is intuitive: double–clicking a visualization node (e.g., *Box Plot*) displays a chart on the user's screen. In contrast, obtaining these graphical results off the computational output of a workflow executed on WS–PGRADE requires certain knowledge of the platform. KNIME2Grid could be extended to directly download and display the output of such nodes.

Transparently executing a set of KNIME Nodes on remote resources is an area worthy of study. KNIME offers the *KNIME Cluster Executor* node, but it has been designed to perform remote execution of single nodes and is offered as a royalty–based extension to the KNIME Analytics Platform. Our proposal would execute computationally demanding sub–workflows on a remote system, while nodes displaying results would still be executed on the KNIME Analytics Platform used to design the converted workflow. This, of course, does not lie within the theoretical boundaries of fine–grained workflow interoperability, however, it would greatly assist researchers by speeding up generation of scientific results.

Bibliography

[1] Roger D Peng, Francesca Dominici, and Scott L Zeger. Reproducible epidemiologic research. *Am. J. Epidemiol.*, 163(9):783–789, May 2006.

[2] Roger D Peng. Reproducible research in computational science. *Science*, 334(6060):1226–1227, December 2011.

[3] Monya Baker. Over half of psychology studies fail reproducibility test. *Nature News & Comment*, `https://doi.org/10.1038/nature.2015.18248` (Accessed Jun 2, 2019), August 2015.

[4] Marcia McNutt. Reproducibility. *Science*, 343(6168):229, January 2014.

[5] Jan Vitek and Tomas Kalibera. Repeatability, reproducibility and rigor in systems research. In *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, pages 33–38. IEEE, October 2011.

[6] Alexander Etz and Joachim Vandekerckhove. A bayesian perspective on the reproducibility project: Psychology. *PLOS ONE*, 11(2):1–12, February 2016.

[7] Ian Sample. Study delivers bleak verdict on validity of psychology experiment results. *The Guardian*, `https://www.theguardian.com/science/2015/aug/27/study-delivers-bleak-verdict-on-validity-of-psychology-experiment-results` (Accessed Jun 2, 2019), August 2015.

[8] Trouble at the lab. *The Economist*, 409(8858):26–30, October 2013.

[9] Let's just try that again: the scientific method. *The Economist*, 418(8975):74, February 2016.

[10] Lin Dai, Xin Gao, Yan Guo, Jingfa Xiao, and Zhang Zhang. Bioinformatics clouds for big data manipulation. *Biol. Direct*, 7(1):43, November 2012.

[11] Neil Savage. Big data versus the big C. *Nature*, 509(7502):S66–S67, May 2014.

[12] Jianwu Wang, Ilkay Altintas, Parviez R Hosseini, Derik Barseghian, Daniel Crawl, Chad Berkley, and Matthew B Jones. Accelerating parameter sweep workflows by utilizing ad–hoc network computing resources: an ecological example. In *2009 Congress on Services – I*, pages 267–274. IEEE, July 2009.

[13] Luis de la Garza, Fabian Aicheler, and Oliver Kohlbacher. From the desktop to the grid and cloud: Conversion of KNIME workflows to WS–PGRADE. In *8th International Workshop of Science Gateways*, volume 5, page e2849v1. PeerJ Preprints, March 2017.

[14] Gabor Terstyanszky, Tamas Kukla, Tamas Kiss, Peter Kacsuk, Ákos Balaskó, and Zoltan Farkas. Enabling scientific workflow sharing through coarse–grained interoperability. *Future Gener. Comp. Sy.*, 37:46–59, July 2014.

[15] Mohamed Abouelhoda, Shadi Alaa Issa, and Moustafa Ghanem. Tavaxy: Integrating Taverna and Galaxy workflows with cloud computing support. *BMC Bioinformatics*, 13(1):77, May 2012.

[16] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, et al. The Taverna workflow suite: Designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Res.*, 41(W1):W557–W561, July 2013.

[17] Richard Grunzke, Florian Jug, Bernd Schuller, René Jäkel, Gene Myers, and Wolfgang E Nagel. Seamless HPC integration of data–intensive KNIME workflows via UNICORE. In *European Conference on Parallel Processing*, pages 480–491. Springer, August 2016.

[18] Diimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distrib. Parallel. Dat.*, 3(2):119–153, April 1995.

[19] Frank B Gilbreth and LM Gilbreth. Process charts and their place in management. *Mech. Eng.*, 70: 38–41, January 1922.

[20] Raúl Medina-Mora, Terry Winograd, Rodrigo Flores, and Fernando Flores. The action workflow approach to workflow management technology. *Inform. Soc.*, 9(4):391–404, October 1993.

[21] Wil MP Van der Aalst. The application of Petri Nets to workflow management. *J. Circuit. Syst. Comp.*, 8(01):21–66, February 1998.

[22] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Kent Blackburn, Albert Lazzarini, Adam Arbree, Richard Cavanaugh, et al. Mapping abstract complex workflows onto grid environments. *J. Grid Comput.*, 1(1):25–39, March 2003.

[23] Jia Yu and Rajkumar Buyya. A taxonomy of workflow management systems for grid computing. *J. Grid Comput.*, 3(3–4):171–200, September 2005.

[24] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Technische Universität Darmstadt, June 1962.

[25] Tadao Murata. Petri Nets: Properties, analysis and applications. *P. IEEE*, 77(4):541–580, April 1989.

[26] Anatol W Holt. Information system theory project. Technical report, Applied Data Research, Inc., Princeton, NJ, September 1968.

[27] Chander Ramchandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*. PhD thesis, Massachusetts Institute of Technology, September 1973.

[28] James L Peterson. Petri Nets. *ACM Comput. Surv.*, 9(3):223–252, September 1977.

[29] Tilak Agerwala. Putting Petri Nets to work. *Computer*, 12(12):85–94, December 1979.

[30] Hartmann J. Genrich and Kurt Lautenbach. System modelling with high–level Petri Nets. *Theor. Comput. Sci.*, 13(1):109–135, January 1981.

[31] Kurt Jensen. High–level Petri Nets. In *Applications and Theory of Petri Nets*, pages 166–180. Springer, 1983.

[32] Kurt Jensen. Coloured Petri Nets and the invariant–method. *Theor. Comput. Sci.*, 14(3):317–336, January 1981.

[33] Kees M van Hee. *Information Systems Engineering: a Formal Approach*. Cambridge University Press, June 1994.

[34] Wil MP Van der Aalst. Putting high–level Petri Nets to work in industry. *Comput. Ind.*, 25(1): 45–54, November 1994.

[35] Jerre D Noe and Gary J Nutt. Macro e–nets for representation of parallel systems. *IEEE T. Comput.*, 100(8):718–727, August 1973.

[36] Cristian Radu Zervos and Keki B Irani. Colored Petri Nets: Their properties and applications. Technical report, Michigan University, Ann Arbor Systems Engineering Lab, August 1977.

[37] James L Peterson. A note on colored Petri Nets. *Inform. Process. Lett.*, 11(1):40–43, August 1980.

[38] Hartmann J Genrich and Kurt Lautenbach. The analysis of distributed systems by means of predicate/transition–nets. In *Semantics of Concurrent Computation*, pages 123–146. Springer, 1979.

[39] Wil MP Van der Aalst. *Timed Coloured Petri Nets and Their Application to Logistics*. PhD thesis, Technische Universiteit Eindhoven, September 1992.

[40] Michael R Berthold, Nicolas Cebron, Fabian Dill, Thomas R Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, and Bernd Wiswedel. KNIME–the Konstanz Information Miner: Version 2.0 and beyond. *SIGKDD Explor.*, 11(1):26–31, November 2009.

[41] Enis Afgan, Dannon Baker, Bérénice Batut, Marius Van Den Beek, Dave Bouvier, Martin Čech, John Chilton, Dave Clements, Nate Coraor, Björn A Grüning, et al. The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update. *Nucleic Acids Res.*, 46(W1): W537–W544, May 2018.

[42] Dassault Systèmes BIOVIA. BIOVIA Pipeline Pilot. `http://www.3dsbiovia.com/products/collaborative-science/biovia-pipeline-pilot/` (Accessed Jun 3, 2019), 2002.

[43] Johannes Junker, Chris Bielow, Andreas Bertsch, Marc Sturm, Knut Reinert, and Oliver Kohlbacher. TOPPAS: A graphical workflow editor for the analysis of high–throughput proteomics data. *J. Proteome Res.*, 11(7):3914–3920, May 2012.

[44] Péter Kacsuk, Zoltán Farkas, Miklós Kozlovszky, Gábor Hermann, Ákos Balaskó, Krisztián Karóczkai, and István Márton. WS–PGRADE/gUSE generic DCI gateway framework for a large variety of user communities. *J. Grid Comput.*, 10(4):601–630, December 2012.

[45] Khodakaram Salimifard and Mike Wright. Petri net–based modelling of workflow systems: an overview. *Eur. J. Oper. Res.*, 134(3):664–676, November 2001.

[46] David Hollingsworth. The workflow reference model. `http://www.wfmc.org/standards/docs/tc003v11.pdf` (Accessed Jun 4, 2019), January 1995.

[47] W3C. Web service definition language (WSDL). `http://www.w3.org/TR/wsdl` (Accessed Jun 2, 2019), June 2007.

[48] MTA SZTAKI Laboratory of Parallel and Distributed Systems. gUSE in a nutshell. `https://sourceforge.net/projects/guse/files/gUSE_in_a_Nutshell.pdf` (Accessed Jun 2, 2019), January 2015.

[49] Oracle. Java SE development kit 7, update 51, release notes. `http://www.oracle.com/technetwork/java/javase/7u51-relnotes-2085002.html` (Accessed Jun 2, 2019), January 2014.

[50] MTA SZTAKI Laboratory of Parallel and Distributed Systems. WS–PGRADE portal user manual, version 3.7.4. `https://sourceforge.net/projects/guse/files/3.7.4/Documentation/Portal_User_Manual_v3.7.4.pdf` (Accessed Jun 2, 2019), November 2015.

[51] Daniel Blankenberg, Gregory Von Kuster, Emil Bouvier, Dannon Baker, Enis Afgan, Nicholas Stoler, James Taylor, and Anton Nekrutenko. Dissemination of scientific software with Galaxy ToolShed. *Genome Biol.*, 15(2):403, February 2014.

[52] Running Galaxy tools on a cluster. `https://docs.galaxyproject.org/en/release_19.01/admin/cluster.html` (Accessed Jun 2, 2019), January 2019.

[53] Shi Meilin, Yang Guangxin, Xiang Yong, and Wu Shangguang. Workflow management systems: a survey. In *1998 International Conference on Communication Technology. Proceedings (IEEE Cat. No. 98EX243)*, volume 2, page 6. IEEE, October 1998.

[54] Marek Rusinkiewicz and Amit P Sheth. Specification and execution of transactional workflows. In *Modern Database Systems: The Object Model, Interoperability, and Beyond*, volume 1995, pages 592–620. ACM Press, January 1995.

[55] Mathias Weske and Gottfried Vossen. Workflow languages. In *Handbook on Architectures of Information Systems*, pages 359–379. Springer, 1998.

[56] Kassian Plankensteiner, Johan Montagnat, and Radu Prodan. IWIR: a language enabling portability across grid workflow systems. In *Proceedings of the 6th Workshop on Workflows in Support of Large–Scale Science*, pages 97–106. ACM, November 2011.

[57] Kassian Plankensteiner, Radu Prodan, Matthias Janetschek, Thomas Fahringer, Johan Montagnat, David Rogers, Ian Harvey, Ian Taylor, Ákos Balaskó, and Péter Kacsuk. Fine–grain interoperability of scientific workflows in distributed computing infrastructures. *J. Grid Comput.*, 11(3):429–455, September 2013.

[58] Peter Amstutz, Michael R Crusoe, Nebojša Tijanić, Brad Chapman, John Chilton, Michael Heuer, Andrey Kartashov, Dan Leehr, Hervé Ménager, Maya Nedeljkovich, et al. Common Workflow Language, v1.0. `https://www.commonwl.org/v1.0/Workflow.html` (Accessed Jun 3, 2019), July 2016.

[59] OpenStand: Principles for the modern standard paradigm. `http://open-stand.org` (Accessed Jun 2, 2019), August 2012.

[60] Galaxy tool XML file. `https://docs.galaxyproject.org/en/release_19.01/dev/schema.html` (Accessed Jun 2, 2019), January 2019.

[61] Tavis Rudd, Mike Orr, Ian Bicking, and C Esterbrook. Cheetah: the Python–powered template engine. In *10th International Python Conference*, February 2002.

[62] Knut Reinert, Temesgen Hailemariam Dadi, Marcel Ehrhardt, Hannes Hauswedell, Svenja Mehringer, René Rahn, Jongkyu Kim, Christopher Pockrandt, Jörg Winkler, Enrico Siragusa, et al. The SeqAn C++ template library for efficient sequence analysis: a resource for programmers. *J. Biotechnol.*, 261:157–168, November 2017.

[63] Hannes L Röst, Timo Sachsenberg, Stephan Aiche, Chris Bielow, Hendrik Weisser, Fabian Aicheler, Sandro Andreotti, Hans-Christian Ehrlich, Petra Gutenbrunner, Erhan Kenar, et al. OpenMS: a flexible open–source software platform for mass spectrometry data analysis. *Nat. Methods*, 13(9): 741, September 2016.

[64] Oliver Kohlbacher. CADDSuite — a workflow–enabled suite of open–source tools for drug discovery. *J. Cheminformatics*, 4(1):O2, December 2012.

[65] CTDopts. `https://github.com/WorkflowConversion/CTDopts` (Accessed Jun 3, 2019), March 2013.

[66] Anna Katharina Hildebrandt, Daniel Stöckel, Nina M Fischer, Luis de la Garza, Jens Krüger, Stefan Nickels, Marc Röttig, Charlotta Schärfe, Marcel Schumann, Philipp Thiel, et al. BALLaxy: Web services for structural bioinformatics. *Bioinformatics*, 31(1):121–122, September 2014.

[67] EMBOSS: the applications (programs). `http://emboss.sourceforge.net/apps/release/6.6/emboss/apps/` (Accessed Jun 2, 2019), July 2012.

[68] Peter Rice, Ian Longden, and Alan Bleasby. EMBOSS: the european molecular biology open software suite. *Trends Genet.*, 16(6):276–277, June 2000.

[69] Hervé Ménager, Matúš Kalaš, Kristoffer Rapacki, and Jon Ison. Using registries to integrate bioinformatics tools and services into workbench environments. *Int. J. Softw. Tools Te.*, 18(6): 581–586, November 2016.

[70] ToolDog – tool description generator. `http://github.com/bio-tools/ToolDog` (Accessed Jun 2, 2019), January 2017.

[71] Erik Elmroth, Francisco Hernández, and Johan Tordsson. Three fundamental dimensions of scientific workflow interoperability: Model of computation, language, and execution environment. *Future Gener. Comp. Sy.*, 26(2):245–256, February 2010.

[72] Marcelo Fiore and Marco Devesas Campos. The algebra of directed acyclic graphs. In *Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky*, pages 37–51. Springer, 2013.

[73] Marc Röttig. *Combining Sequence and Structural Information Into Predictors of Enzymatic Activity*. Verlag Dr. Hut, 2013.

[74] Generic KNIME Nodes. `https://github.com/genericworkflownodes/GenericKnimeNodes` (Accessed Jun 3, 2019), August 2011.

[75] KNIME AG. FAQ — is there any way to run KNIME in batch mode, i.e. only on command line and without the graphical user interface? `http://tech.knime.org/faq#q12` (Accessed Jun 2, 2019), 2018.

[76] KNIME AG. RSnippet KNIME Node. `https://nodepit.com/node/de.mpicbg.knime.scripting.r.RSnippetNodeFactory` (Accessed Jun 2, 2019), December 2018.

[77] Ali Anjomshoaa, Fred Brisard, Michel Drescher, Donal Fellows, An Ly, Stephen McGough, Darren Pulsipher, and Andreas Savva. Job submission description language (JSDL) specification, version 1.0. `https://www.ogf.org/documents/GFD.56.pdf` (Accessed Jun 2, 2019), November 2005.

[78] MTA SZTAKI Laboratory of Parallel and Distributed Systems. DCI Bridge administrator manual, version 3.7.4. `https://sourceforge.net/projects/guse/files/DCI_BRIDGE_MANUAL_v3.7.4.pdf` (Accessed Jun 2, 2019), January 2015.

[79] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison–Wesley Professional, 2nd edition, 2009.

[80] Aleksa Vukotic and James Goodwill. *Apache Tomcat 7*. Apress, 2011.

[81] The Apache Software Foundation. Apache Maven Project. `https://maven.apache.org/` (Accessed Jun 2, 2019), 2002.

[82] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Software Components*. Addison–Wesley Professional, 1995.

[83] Ákos Balaskó, Zoltán Farkas, and Péter Kacsuk. Building science gateways by utilizing the generic WS–PGRADE/gUSE workflow system. *Computer Science*, 14(2):307, 2013.

[84] Jens Krüger, Richard Grunzke, Sandra Gesing, Sebastian Breuers, André Brinkmann, Luis de la Garza, Oliver Kohlbacher, Martin Kruse, Wolfgang E Nagel, Lars Packschies, et al. The MoSGrid science gateway—a complete solution for molecular simulations. *J. Chem. Theory Comput.*, 10 (6):2232–2245, May 2014.

[85] Eva Sciacca, Marilena Bandieramonte, Ugo Becciani, Alessandro Costa, Mel Krokos, Piero Massimino, Catia Petta, Costantino Pistagna, Simone Riggi, and Fabio Vitello. VisIVO science gateway: a collaborative environment for the astrophysics community. In *5th International Workshop on Science Gateways*, volume 993, page 1. CEUR Workshop Proceedings, June 2013.

[86] Gabriele Pierantoni and Eoin Carley. HELIOGate: a portal for the heliophysics community. In *Science Gateways for Distributed Computing Infrastructures*, pages 195–207. Springer, 2014.

[87] Martin Fowler. Inversion of control containers and the dependency injection pattern. `https://martinfowler.com/articles/injection.html` (Accessed Jun 2, 2019), January 2004.

[88] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, volume 10, pages 707–710, February 1966.

[89] Brian Goetz and Tim Peierls. *Java Concurrency in Practice*. Pearson Education, 2006.

[90] Brian K Shoichet, Irwin D Kuntz, and Dale L Bodian. Molecular docking using shape descriptors. *J. Comput. Chem.*, 13(3):380–397, April 1992.

[91] Namrata Tomar and Rajat K De. Immunoinformatics: a brief review. In *Immunoinformatics*, pages 23–55. Springer, 2014.

[92] Benjamin Schubert, Luis de la Garza, Christopher Mohr, Mathias Walzer, and Oliver Kohlbacher. ImmunoNodes — graphical development of complex immunoinformatics workflows. *BMC Bioinformatics*, 18(1):242, May 2017.

[93] Taverna 2–Galaxy. `http://www.taverna.org.uk/documentation/taverna-galaxy/` (Accessed Jun 2, 2019), 2011.

[94] CTDopts sample script. `https://github.com/WorkflowConversion/CTDopts/blob/master/example.py` (Accessed Jun 2, 2019), November 2013.

Appendix A

Abbreviations

A	
ACD	*AJAX Command Definition*
API	*Application Programming Interface*
ASM	*Application Specific Module*
B	
BALL	*Biochemical Algorithms Library*
C	
CADDSuite	*Computer Aided Drug Design Suite*
CTD	*Common Tool Descriptor*
CWL	*Common Workflow Language*
D	
DAG	*Directed Acyclic Graph*
DCI	*Distributed Computing Interface*
E	
EBI	*European Bioinformatics Institute*
EMBOSS	*European Molecular Biology Open Software Suite*
EMF	*Eclipse Modeling Framework*
F	
FRED2	*Framework for Epitope Detection*
G	
GKN	*Generic KNIME Nodes*
GUI	*Graphical User Interface*
gUSE	*Grid and Cloud User Support Environment*
H	
HPC	*High–Performance Computing*

I	
IDE	*Integrated Development Environment*
I/O	*Input/Output*
IWIR	*Interoperable Workflow Intermediate Representation*
H	
HLA	*Human Leukocyte Antigen*
J	
JSP	*Java Server Page*
JSDL	*Job Submission Description Language*
JSON	*JavaScript Object Notation*
K	
KNIME	*Konstanz Information Miner*
N	
NCBI	*National Center of Biotechnology Information*
P	
PDB	*Protein Data Bank*
PIN	*Personal Identification Number*
POM	*Project Object Model*
PrT–Nets	*Predicate/Transition–Nets*
R	
REST	*Representational State Transfer*
S	
SHIWA	*Sharing Interoperable Workflows for large–scale scientific Simulations on Available DCIs*
SSH	*Secure Shell*
SDF	*Structure Data File*
U	
UNICORE	*Uniform Interface to Computing Resources*
URI	*Uniform Resource Identifier*
W	
WAR File	*Web Application Resource File*
WSDL	*Web Services Description Language*
WS–PGRADE	*Web Services Parallel Grid Runtime and Developer Environment Portal*
X	
XML	*Extensible Markup Language*
Y	
YAML	*YAML Ain't Markup Language*

Appendix B

Sample Code

B.0.1 Platform–Independent Workflow Representation

In this section we will present both CWL and IWIR representations for the following workflow:

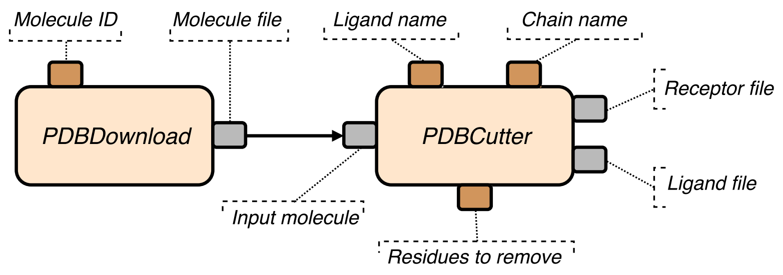


Figure B.1: Molecule preparation using the CADDSuite[64]. Input and output files are shown as gray ports, while parameters are shown as dark orange ports.

The task *PDBDownload* receives a parameter, *Molecule ID*, which is the 4–character unique identification code from the PDB website, and outputs the molecule in a file. The *PDBCutter* task receives said molecule file as an input, taking three parameters: name of the reference ligand, name of the chain in which the reference ligand is found, and the residue(s) to remove. *PDBCutter* outputs two files: a file in which the reference ligand is contained, and the macromolecule that will be used as a receptor. The output files of *PDBCutter* can be used in any standard molecular docking pipeline. Both tools are part of the CADDSuite[64].

Listing B.1: CWL definition of *PDBCutter* from the workflow depicted in Figure B.1

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: PDBCutter
requirements:
  EnvVarRequirement:
    envDef:
      BALL_DATA_PATH: /Users/delagarza/Projects/ball/data
inputs:
  pdb_in:
    type: File
    inputBinding:
      prefix: -i
  receptor_filename:
    type: string
    default: receptor.pdb
    inputBinding:
      prefix: -rec
  ligand_filename:
    type: string
    default: ligand.pdb
    inputBinding:
      prefix: -lig
  ligand_chain:
    type: string
    inputBinding:
      prefix: -lig_chain
  ligand_name:
    type: string
    inputBinding:
      prefix: -lig_name
  remove_residue:
    type: string
    inputBinding:
      prefix: -rm_res
outputs:
  receptor_out:
    type: File
    outputBinding:
      glob: $(inputs.receptor_filename)
  ligand_out:
    type: File
    outputBinding:
      glob: $(inputs.ligand_filename)
```

Listing B.2: CWL definition of *PDBDownload* from the workflow depicted in Figure B.1.

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: PDBDownload
requirements:
  EnvVarRequirement:
    envDef:
      BALL_DATA_PATH: /Users/delagarza/Projects/ball/data
inputs:
  pdb_id:
    type: string
    inputBinding:
      prefix: -id
  output_filename:
    type: string
    default: molecule.pdb
    inputBinding:
      prefix: -o
  proxy:
    type: ['null', string]
    inputBinding:
      prefix: -p
outputs:
  molecule_out:
    type: File
    outputBinding:
      glob: $(inputs.output_filename)
```

Listing B.3: CWL definition of the workflow depicted in Figure B.1. The files presented in Listings B.1 and B.2 are automatically included if they are located on the same directory as this workflow definition. In this case, they were stored under *PDBCutter.cwl* and *PDBDownload.cwl*, respectively.

```
cwlVersion: v1.0
class: Workflow
inputs:
  pdb_id: string
  ligand_chain: string
  ligand_name: string
  remove_residue: string

outputs:
  receptor_out:
    type: File
    outputSource: PDBCutter/receptor_out
  ligand_out:
    type: File
    outputSource: PDBCutter/ligand_out

steps:
  PDBDownload:
    run: PDBDownload.cwl
    in:
      pdb_id: pdb_id
    out: [molecule_out]

  PDBCutter:
    run: PDBCutter.cwl
    in:
      pdb_in: PDBDownload/molecule_out
      ligand_chain: ligand_chain
      ligand_name: ligand_name
      remove_residue: remove_residue
    out: [receptor_out, ligand_out]
```

Listing B.4: CWL *run file* to execute the workflow defined in Listing B.3. Run files contain the parameters that will be passed to tasks and workflows.

```
pdb_id: 1DX6
ligand_chain: A
ligand_name: GNT
remove_residue: HOH
```

Listing B.5: Command line to execute the workflow defined in Listing B.3. The *cwl–runner* tool is passed two files as inputs: a workflow definition (see Listing B.3) and a run file (see Listing B.4), here shown as *PDBPreparationWorkflow.cwl* and *PDBPreparationWorkflow_run.yml*, respectively.

```
$ cwl-runner PDBPreparationWorkflow.cwl PDBPreparationWorkflow_run.yml
```

Listing B.6: IWIR representation of the workflow depicted in Figure B.1.

```
<IWIR version="1.1" wfname="PDBPreparationWorkflow"
      xmlns="http://shiwa-workflow.eu/IWIR">
  <task name="PDBDownload" tasktype="binary">
    <inputPorts>
      <inputPort name="pdb_id"            type="string"/>
      <inputPort name="molecule_filename" type="string"/>
    </inputPorts>
  <outputPorts>
    <outputPort name="molecule_out" type="file"/>
    </outputPorts>
  </task>
  <task name="PDBCutter" tasktype="binary">
    <inputPorts>
      <inputPort name="pdb_in"            type="file"/>
      <inputPort name="ligand_name"       type="string"/>
      <inputPort name="ligand_chain"      type="string"/>
      <inputPort name="remove_residue"    type="string"/>
      <inputPort name="ligand_filename"   type="string"/>
      <inputPort name="receptor_filename" type="string"/>
    </inputPorts>
  <outputPorts>
    <outputPort name="receptor_out" type="file"/>
    <outputPort name="ligand_out"   type="file"/>
    </outputPorts>
  </task>
  <links>
    <link from="PDBDownload/molecule_out" to="PDBCutter/pdb_in"/>
  </links>
</IWIR>
```

B.0.2 CTD Usage

Listing B.7: Definition of CADDSuite's *PDBDownload* using a CTD (refer to Appendix B.0.1 for details on this tool). The CADDSuite offers CTD–enabled command line tools.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tool version="1.1.0" name="PDBDownload" category="Get Data" ctdVersion="1.7">
    <description>retrieve pdb-file from pdb.org</description>
    <manual>Download a pdb-file from the pdb data bank (http://www.pdb.org/)
        using the specified ID of the desired protein structure.</manual>
    <executableName>PDBDownload</executableName>
    <PARAMETERS version="1.7">
        <NODE name="1" description="Instance '1' section for 'PDBDownload'">
            <ITEM name="id" type="string" required="true" value=""
                  description="PDB ID for desired structure" />
            <ITEM name="o" type="output-file" supported_formats="pdb" value=""
                  description="output file" required="true" />
            <ITEM name="p" type="string" description="proxy" value=""/>
            <ITEM name="env" type="string" value="cmdline"
                  description="set runtime environment (default cmdline) " />
        </NODE>
    </PARAMETERS>
</tool>
```

Listing B.8: Executing a CTD–enabled tool using a CTD file. The contents of *params.ctd* are as shown in Listing B.7.

```
$ PDBDownload -par params.ctd
```

Listing B.9: Execution of a CTD–enabled tool without usage of CTD files.

```
$ PDBDownload -id 1DX6 -o /Users/delagarza/Projects/1DX6.pdb -p "http://proxy.edu"
```

B.0.3 CTDopts Usage

Listing B.10: Generating a CTD file for a non–CTD–enabled command line tool using CTDopts. Modified from[94].

```
import CTDopts.CTDopts
from CTDopts.CTDopts import CTDModel

model = CTDModel(
    name='Sample Tool',
    version='1.0',
    description='This is an example tool illustrating CTDopts usage',
    category='Testing',
    executableName='sampletool',
    executablePath='/path/to/sampletool')

model.add(
    'positive_int',
    type=int,
    num_range=(0, None),
    default=5,
    description='A positive integer parameter')

model.add(
    'input_files',
    required=True,
    type='input-file',
    is_list=True,
    file_formats=['fastq', 'fastq.gz'],
    description='A list of filenames to feed this tool with')

model.write_ctd('example_tool.ctd')
```

B.0.4 Sample ToolConfig Files

Listing B.11: Sample ToolConfig file with Python–like code in the `<command>` section. Only selected arguments of the *wget* tool are shown.

```
<?xml version='1.0' encoding='UTF-8'?>
<tool id="Wget" name="wget" version="1.17.1">
  <description>The non-interactive network downloader.</description>
  <command>
    wget --quiet
    #if $bypass_certificate:
      --no-check-certificate
    #end if
    -O $output_file
    #for url in $urls:
      $url
    #end for
  </command>
  <inputs>
    <param name="urls" type="text" multiple="true" optional="false"/>
    <param name="bypass_certificate" type="boolean" optional="true"/>
  </inputs>
  <outputs>
    <data name="output_file" format="html"/>
  </outputs>
</tool>
```

Listing B.12: Two possible command lines generated by the Galaxy engine for the ToolConfig shown in Listing B.11.

```
  $ wget --quiet --no-check-certificate -O http://server.com/page.html
  $ wget --quiet -O http://server.com/page.html http://site.com/index.html
```

B.0.5 Galaxy Support Files in CTDConverter

Listing B.13: Macros file used for the generation of the ToolConfig shown in Listing 3.4. This file is included in CTDConverter.

```xml
<?xml version='1.0' encoding='UTF-8'?>
<macros>
  <xml name="requirements">
    <requirements>
      <requirement type="binary">@EXECUTABLE@</requirement>
    </requirements>
  </xml>
  <xml name="stdio">
    <stdio>
      <exit_code range="1:"/>
      <exit_code range=":-1"/>
      <regex match="Error:"/>
      <regex match="Exception:"/>
    </stdio>
  </xml>
  <xml name="advanced_options">
    <conditional name="adv_opts">
      <param name="adv_opts_selector" type="select" label="Advanced Options">
        <option value="basic" selected="True">Hide Advanced Options</option>
        <option value="advanced">Show Advanced Options</option>
      </param>
      <when value="basic"/>
      <when value="advanced">
        <yield/>
      </when>
    </conditional>
  </xml>
</macros>
```

B.0.6 Representation of Workflows in WS–PGRADE

Listing B.14: Sample *workflow.xml* file containing a possible implementation of the workflow shown in Section 2.1.3 using CADDSuite tools and a computer cluster where the Moab Workload Manager has been installed. The abstract and conrete layers are defined by the `<graf>` and `<real>` elements, respectively. The script referred to in `binary` attributes is responsible to parse the provided command line provided via `params` attributes. WS–PGRADE/gUSE expects such scripts to be preppended with the `C:/fakepath/` path. Jobs will be submitted to a Moab batch queueing system on the *fast* queue, as specified by the `gridtype` and `resource` attributes.

```xml
<workflow name="Ligand_Preparation_WF" maingraf="LigPrep" mainreal="LigPrep"
          download="all" export="proj" mainabst="">
  <graf name="LigPrep">
    <job name="MoleculeCheck">
      <input name="molecule" prejob="" preoutput="" seq="0"/>
      <output name="checked" seq="1"/>
    </job>
    <job name="3DGenerator">
      <input name="2dmolecule" prejob="MoleculeCheck" preoutput="1" seq="0"/>
      <output name="3dmolecule" seq="1"/>
    </job>
  </graf>
  <real name="LigPrep" abst="" graph="LigPrep">
    <job name="MoleculeCheck">
      <input name="molecule" prejob="" preoutput="" seq="0">
        <port_prop key="file" value="C:/fakepath/mol.sdf"/>
        <port_prop key="intname" value="molecule"/>
      </input>
      <output name="checked" seq="1"/>
      <execute key="gridtype" value="moab"/>
      <execute key="resource" value="fast"/>
      <execute key="binary" value="C:/fakepath/runjob.sh"/>
      <execute key="jobistype" value="binary"/>
      <execute key="grid" value="masternode.informatik.uni-tuebingen.de"/>
      <execute key="params" value="/share/bin/BALL/LigCheck -i molecule -o checked/>
    </job>
    <job name="3DGenerator">
      <input name="2dmolecule" prejob="MoleculeCheck" preoutput="1" seq="0"/>
      <output name="3dmolecule" seq="1"/>
      <execute key="gridtype" value="moab"/>
      <execute key="resource" value="fast"/>
      <execute key="binary" value="C:/fakepath/runjob.sh"/>
      <execute key="jobistype" value="binary"/>
      <execute key="grid" value="masternode.informatik.uni-tuebingen.de"/>
      <execute key="params" value="/share/bin/BALL/Ligand3DGenerator -i 2dmolecule -o 3dmolecule"/>
    </job>
  </real>
</workflow>
```

B.0.7 Support Ant Script for the WS–PGRADE Extensions

Listing B.15: Command required to build and deploy portlets using our support Ant Script (shown in Listing B.17). Files are copied using *scp*. Credentials are provided via paswordless authentication, location of the required SSH public key is provided using a configuration file, *deployment.properties* (refer to Listing B.16 for an example of this file).

```
$ ant deploy
```

Listing B.16: Sample *deployment.properties* file used to deploy portlets on a remote server.

```
# remote server to which the war file will be copied
remote.server=knime2guse.informatik.uni-tuebingen.de

# location of your keyfile
keyfile.location=~/.ssh/id_rsa

# username (passwordless SSH, using provided key)
remote.server.username=guseuser

# remote server’s liferay deploy folder
remote.server.deploy.path=/home/guseuser/guse/deploy
```

Listing B.17: Apache Ant script to facilitate deployment of portlets on a running Liferay instance.

```
<project name="WS-PGRADE-Extensions" default="deploy" basedir=".">
  <description>Builds, deploys portlets on a remote Liferay instance.</description>
  <property name="deployment.properties.file" value="deployment.properties" />
  <property name="application.manager.path" value="application-manager" />

  <target name="resource-check">
    <available file="${deployment.properties.file}" property="deployment.properties.present" />
  </target>

  <target name="fail-if-missing-properties-file" depends="resource-check"
          unless="deployment.properties.present">
    <fail message="Missing file ${deployment.properties.file}" />
  </target>

  <target name="fail-if-missing-property" unless="${required.property}">
    <fail message="Missing property ${required.property} in file ${deployment.properties.file}" />
  </target>

  <target name="fail-if-missing-properties" depends="fail-if-missing-properties-file">
    <property file="${deployment.properties.file}" />
    <antcall target="fail-if-missing-property">
      <param name="required.property" value="remote.server" />
    </antcall>
    <antcall target="fail-if-missing-property">
      <param name="required.property" value="remote.server.username" />
    </antcall>
    <antcall target="fail-if-missing-property">
      <param name="required.property" value="remote.server.deploy.path" />
    </antcall>
    <antcall target="fail-if-missing-property">
      <param name="required.property" value="remote.server.scripts.path" />
    </antcall>
    <antcall target="fail-if-missing-property">
      <param name="required.property" value="keyfile.location" />
    </antcall>
  </target>

  <target name="build">
      <exec dir="${basedir}" executable="mvn"><arg value="clean"/><arg value="package"/></exec>
  </target>

  <target name="upload-portlet-tmpfile" depends="fail-if-missing-properties">
    <property file="${portlet.dir}/build.properties" />
    <echo>Uploading ${portlet.name}.war.</echo>
    <scp file="${portlet.dir}/target/${portlet.name}.war" trust="true" verbose="false"
         remoteTofile="${remote.server.username}@${remote.server}
                        :${remote.server.deploy.path}/${portlet.name}.war_tmp"
          keyfile="${keyfile.location}" />
    <sshexec host="${remote.server}" username="${remote.server.username}" trust="true"
           keyfile="${keyfile.location}" verbose="false"
             command="mv ${remote.server.deploy.path}/${portlet.name}.war_tmp
                         ${remote.server.deploy.path}/${portlet.name}.war" />
  </target>

  <target name="deploy" depends="fail-if-missing-properties">
    <echo>Building...</echo>
    <antcall target="build" />
    <echo>Stopping WS-PGRADE portal.</echo>
    <sshexec host="${remote.server}" username="${remote.server.username}" trust="true"
           verbose="false" keyfile="${keyfile.location}"
             command="${remote.server.scripts.path}/stop.sh" />
    <echo>Uploading... This might take a while depending on the speed of your connection.</echo>
    <antcall target="upload-portlet-tmpfile">
      <param name="portlet.dir" value="${application.manager.path}" />
    </antcall>
    <echo>Starting WS-PGRADE portal. This will take a few minutes.</echo>
    <sshexec host="${remote.server}" username="${remote.server.username}" trust="true"
           verbose="false" keyfile="${keyfile.location}"
             command="${remote.server.scripts.path}/start.sh" />
  </target>
</project>
```

Appendix C

Contributions

All ideas, approaches and results here presented were developed and discussed with my supervisor, Oliver Kohlbacher (OK). The following colleagues contributed as detailed below:

Ákos Balaskó	(AB)	Julianus Pfeuffer	(JP)
Alexander Fillbrunn	(AF)	Johannes Veit	(JV)
András Szolek	(AS)	Marc Röttig	(MR)
Björn Grüning	(BG)	Mathias Walzer	(MW)
Benjamin Schubert	(BS)	Peter Ohl	(PO)
Bernd Wiswedel	(BW)	Philipp Thiel	(PT)
Christopher Mohr	(CM)	Stephan Aiche	(SA)
Charlotta Schärfe	(CS)	Thorsten Meinl	(TM)
Fabian Aicheler	(FA)	Thomas Gabriel	(TG)
István Márton	(IM)	Zoltán Farkas	(ZF)
Jens Krüger	(JK)		

Chapter 3: Conversion of Workflow Nodes

The project was designed by myself, AS and OK. BG and PT contributed to this project by providing use cases in bioinformatics for the converter.

Chapter 4: Conversion of Complete Workflows

The project was designed by myself and OK. AB, AF, BW, IM, JP, MR, SA, TG, TM, and ZF provided assistance to elucidate implementation details of both workflow engines.

Section 4.3.4: Use Cases

The showcased pipelines were designed by BS, CM, CS, FA, JK, PT, JV, MW, and OK.

Appendix D

Publications

2019

de la Garza, L.*, Fillinger, S.*, Peltzer, A.*, Kohlbacher, O., Nahnsen, S., May 2019. **Challenges of Big Data Integration in the Life Sciences**. *Anal. Bioanal. Chem.*, Manuscript submitted for publication.

2018

Friedrich, A., **de la Garza, L.**, Kohlbacher, O., Nahnsen, S., December 2018. **Interactive Visualization for Large–Scale Multi–Factorial Research Designs**. In *International Conference on Data Integration in the Life Sciences*, Springer, pp. 75–84.

2017

Schubert, B., **de la Garza, L.**, Mohr, C., Walzer, M., Kohlbacher, O., May 2017. **ImmunoNodes: Graphical Development of Complex Immunoinformatics Workflows**. *BMC Bioinformatics*, 18(1):242.

de la Garza, L., Aicheler, F., Kohlbacher, O., March 2017. **From the Desktop to the Grid and Cloud: Conversion of KNIME Workflows to WS-PGRADE**. In *8th International Workshop of Science Gateways*, PeerJ Preprints, 5:e2849v1.

2016

de la Garza, L., Veit, J., Szolek, A., Röttig, M., Aiche, S., Gesing, S., Reinert, K., Kohlbacher, O., March 2016. **From the Desktop to the Grid: Scalable Bioinformatics via Workflow Conversion**. *BMC Bioinformatics*, 17(1):127.

2015

Herres–Pawlis, S., Hoffmann, A., Balaskó, Á., Kacsuk, P., Birkenheuer, G., Brinkmann, A., **de la Garza, L.**, Krüger, J., Gesing, S., Grunzke, R., Terstyansky, G., February 2015. **Quantum Chemical Meta–Workflows in MoSGrid**. *Concurr. Comp.–Pract. E.*, 27(2):344–357.

Hildebrandt, A.K., Stöckel, D., Fischer, N.M., **de la Garza, L.**, Krüger, J., Nickels, S., Röttig, M., Schärfe, C., Schumann, M., Thiel, P., Lenhof, H.P., Kohlbacher, O., Hildebrandt, A., January 2015. **BALLaxy: Web Services for Structural Bioinformatics**. *Bioinformatics*, 31(1):121–122.

2014

Gesing, S., Krüger, J., Grunzke, R., **de la Garza, L.**, Herres–Pawlis, S., Hoffmann, A., October 2014. **Molecular Simulation Grid (MoSGrid): a Science Gateway Tailored to the Molecular Simulation Community**. In *Science Gateways for Distributed Computing Infrastructures*, Springer, pp. 151–165.

Olabarriaga, S.D., Benabdelkader, A., Caan, M.W., Jaghoori, M.M., Krüger, J., **de la Garza, L.**, Mohr, C., Schubert, B., Danezi, A., Kiss, T., October 2014. **WS–PGRADE/gUSE–Based Science Gateways in Teaching**. In *Science Gateways for Distributed Computing Infrastructures*, Springer, pp. 223–234.

Grunzke, R., Breuers, S., Gesing, S., Herres–Pawlis, S., Kruse, M., Blunk, D., **de la Garza, L.**, Packschies, L., Schäfer, P., Schärfe, C., Schlemmer, T., July 2014. **Standards–Based Metadata Management for Molecular Simulations**. *Concurr. Comp.–Pract. E.*, 26(10):1744–1759.

Krüger, J., Grunzke, R., Herres–Pawlis, S., Hoffmann, A., **de la Garza, L.**, Kohlbacher, O., Nagel, W.E., Gesing, S., June 2014. **Performance Studies on Distributed Virtual Screening**. *BioMed Res. Int.*, 2014:624024.

Herres–Pawlis, S., Hoffmann, A., Grunzke, R., Nagel, W.E., **de la Garza, L.**, Krüger, J., Terstyansky, G., Weingarten, N., Gesing, S., June 2014. **Meta–Metaworkflows for Combining Quantum Chemistry and Molecular Dynamics in the MoSGrid Science Gateway**. In *6th International Workshop on Science Gateways*, IEEE, pp. 73–78.

Herres–Pawlis, S., Hoffmann, A., **de la Garza, L.**, Krüger, J., Grunzke, R., June 2014. **Expansion of Quantum Chemical Metadata for Workflows in the MoSGrid Science Gateway**. In *6th International Workshop on Science Gateways*, IEEE, pp. 67–72.

Krüger, J., Grunzke, R., Gesing, S., Breuers, S., Brinkmann, A., **de la Garza, L.**, Kohlbacher, O., Kruse, M., Nagel, W.E., Packschies, L., Müller–Pfefferkorn, R., May 2014. **The MoSGrid Science Gateway: a Complete Solution for Molecular Simulations**. *J. Chem. Theory Comput.*, 10(6):2232–2245.

2013

Herres–Pawlis, S., Hoffmann, A., **de la Garza, L.**, Krüger, J., Gesing, S., Grunzke, R., September 2013. **User–Friendly Metaworkflows in Quantum Chemistry**. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, pp. 1–3.

de la Garza, L., Krüger, J., Schärfe, C., Röttig, M., Aiche, S., Reinert, K., Kohlbacher, O., June 2013. **From the Desktop to the Grid: Conversion of KNIME Workflows to gUSE**. In *5th International Workshop on Science Gateways*, CEUR Workshop Proceedings, 993:9.

$*$ co–first author