Departamento de Inteligencia Artificial
Escuela Técnica Superior de Ingenieros Informáticos

PhD Thesis

Conservation of Computational Scientific Execution Environments for Workflow-based Experiments Using Ontologies

Author: Idafen Santana Pérez

Supervisors: Prof. Dra. María Pérez Hernández
Prof. Dr. Oscar Corcho

January, 2016

Tribunal nombrado por el Sr. Rector Magfco. de la Universidad Politécnica de Madrid,
el día 18 de Enero de 2016.

Presidente: Dra. Asunción Gómez Pérez

Vocal: Dr. Erick Antezana

Vocal: Dr. Alexander García

Vocal: Dr. Jesús Carretero

Secretario: Dr. Rubén Montero

Suplente: Dr. Mariano Fernández López

Suplente: Dr. Alberto Sánchez Campos

Realizado el acto de defensa y lectura de la Tesis el día 22 de Enero de 2016 en la
Escuela Técnica Superior de Ingenieros Informáticos

Calificación: _____

EL PRESIDENTE

VOCAL 1

VOCAL 2

VOCAL 3

EL SECRETARIO

A Gina y Carlos.

A Isa.

Agradecimientos

And now, the end is near, and so I face this final curtain, I have to take a moment to thank all the people that I found along the way, that have helped me in this journey. From research colleagues, to friend and family, many people have contributed to this thesis, both at the professional and personal level.

First of all, I would like to thank my advisors, María Pérez and Oscar Corcho, for guiding and supporting me all these years, allowing me to find my own path in my research career, and helping me whenever I needed it. As well I would like to thank Asunción Gómez Pérez, for the leading role she has played in our group during these years.

Also, I am very grateful to the spanish ministry of education, for being granted with a FPU scholarship, which has funded this PhD and its associated research stays. In this regard, I would also like to thank Ewa Deelman, for hosting my at the Information Sciences Institute, and to the rest of the Pegasus team, Karan, Gideon, Mats, and specially to Rafael Ferreira, who has helped me a lot, with highly valuable feedback and ideas for this thesis. As well, I would like to thank Rosa Filgueira and Amy Krause for their help with dispel4py, and to the Makeflow team.

Specially, I would like to thank the OEG and all its members. I have had the pleasure to work with a terrific group of researchers that became friends throughout the years. Thanks to Dani, a good friend, great researcher and soccer mate. Thanks to Miguel Ángel, for being there in the lab and in the trips we made. Thanks to Esther, for your help on the early days, and for the silly jokes. To María, for being there when it was needed, for the many coffees, the cheesecake ice cream, and the movies. And to the rest of the group, Filip, Freddy, Alcazar, Pablo, Andrés, Olga, Raúl, Rafa, Nandana, Julia, Carlos, Alejandro and Víctor, for your support and help. We are all part of a really great team.

Last, but not least, on the personal side, there are many people I would like to thank. And for that, I would switch to spanish. Quiero dar las gracias a Gina y Carlos, por haber sido fuente de apoyo, de cariño y de consejo durante estos años, los de la tesis y todos los anteriores. Por estar ahí en los momentos duros, salvando los 1700 km que nos han separado, por hacerme sentir en casa en la distancia. Y a mi hermano, Himar, por ser mi contrapunto natural y un modelo a seguir en tantas cosas que me completan. Y a toda mi familia, por estar siempre ahí para mí.

Y a Isa, por aparecer en mi vida, por quedarse en ella, y por los años que nos quedan por vivir juntos. Gracias por aguantarme, por tu infinita paciencia y por no rendirte nunca. Sin tí hoy no sería ni la mitad de buena persona de lo que soy, ni creo que hubiese llegado hasta aquí. Me haría falta otra tesis entera para contarte lo feliz que me haces.

Además, quiero dar las gracias a todas las personas que de una forma u otra me han ayudado con las pequeñas y grandes cosas de la vida de un canario en Madrid. Gracias a Rubén Quesada, amigo y compañero en toda mi carrera académica previa, por estar siempre en el sitio adecuado. A Elena García, por haber acudido en el peor momento, por transformarlo en no tan malo, por haber sido mi apoyo en el primer momento, cuando solo se quedan los mejores. Y a Alberto, por la amistad en la distancia, por los retos culinarios, y por saber que siempre puedo contar contigo.

A Ángel, Abian y Sergio, por compartir el día a día conmigo durante años. A todos los miembros del Brasil del Gofio, una pequeña familia y grandísimo equipo. Al Cícero Canary, a Ricky y a Dafne, a Valle, a Concha, a mi familia política y a Madrid, por haberme pedido tanto, para luego darmelo todo. A todos, gracias.

Abstract

Reproducibility of scientific studies and results is a goal that every scientist must pursue when announcing research outcomes. The rise of computational science, as a way of conducting empirical studies by using mathematical models and simulations, have opened a new range of challenges in this context. The adoption of workflows as a way of detailing the scientific procedure of these experiments, along with the experimental data conservation initiatives that have been undertaken during last decades, have partially eased this problem. However, in order to fully address it, the conservation and reproducibility of the computational equipment related to them must be also considered. The wide range of software and hardware resources required to execute a scientific workflow implies that a comprehensive description detailing what those resources are and how they are arranged is necessary. In this thesis we address the issue of reproducibility of execution environments for scientific workflows, by documenting them in a formalized way, which can be later used to obtain an equivalent one. In order to do so, we propose a set of semantic models for representing and relating the relevant information of those environments, as well as a set of tools that uses these models for generating a description of the infrastructure, and an algorithmic process that consumes these descriptions for deriving a new execution environment specification, which can be enacted into a new equivalent one using virtualization solutions. We apply these three contributions to a set of representative scientific experiments, belonging to different scientific domains, and exposing different software and hardware requirements. The obtained results prove the feasibility of the proposed approach, by successfully reproducing the target experiments under different virtualization environments.

Resumen

La reproducibilidad de estudios y resultados científicos es una meta a tener en cuenta por cualquier científico a la hora de publicar el producto de una investigación. El auge de la ciencia computacional, como una forma de llevar a cabo estudios empíricos haciendo uso de modelos matemáticos y simulaciones, ha derivado en una serie de nuevos retos con respecto a la reproducibilidad de dichos experimentos. La adopción de los flujos de trabajo como método para especificar el procedimiento científico de estos experimentos, así como las iniciativas orientadas a la conservación de los datos experimentales desarrolladas en las últimas décadas, han solucionado parcialmente este problema. Sin embargo, para afrontarlo de forma completa, la conservación y reproducibilidad del equipamiento computacional asociado a los flujos de trabajo científicos deben ser tenidas en cuenta. La amplia gama de recursos hardware y software necesarios para ejecutar un flujo de trabajo científico hace que sea necesario aportar una descripción completa detallando que recursos son necesarios y como estos deben de ser configurados. En esta tesis abordamos la reproducibilidad de los entornos de ejecución para flujos de trabajo científicos, mediante su documentación usando un modelo formal que puede ser usado para obtener un entorno equivalente. Para ello, se ha propuesto un conjunto de modelos para representar y relacionar los conceptos relevantes de dichos entornos, así como un conjunto de herramientas que hacen uso de dichos módulos para generar una descripción de la infraestructura, y un algoritmo capaz de generar una nueva especificación de entorno de ejecución a partir de dicha descripción, la cual puede ser usada para recrearlo usando técnicas de virtualización. Estas contribuciones han sido aplicadas a un conjunto representativo de experimentos científicos pertenecientes a diferentes dominios de la ciencia, exponiendo cada uno de ellos diferentes requisitos hardware y software. Los resultados obtenidos muestran la viabilidad de propuesta desarrollada, reproduciendo de forma satisfactoria los experimentos estudiados en diferentes entornos de virtualización.

Contents

1	Introduction	1
1.1	Computational Science	3
1.1.1	Experimental Sciences Approaches	4
1.1.2	Computational Scientific Experiments	7
1.2	Scientific Conservation and Reproducibility	15
1.3	Thesis Structure	19
2	State of the Art	21
2.1	Data Infrastructures for eScience	21
2.2	Computational Infrastructures for eScience	23
2.2.1	Distributed Computing	24
2.2.2	Workflow Management Systems	31
2.3	Reproducibility in Computational Science	36
2.3.1	Scientific procedure conservation	37
2.3.2	Data conservation	40
2.3.3	Equipment conservation	41
2.4	Summary	47
3	Work Objectives	49
3.1	Open Research Problems	49
3.2	Hypothesis	50
3.3	Goals	51
3.4	Restrictions	51
3.5	Assumptions	52
3.6	Research methodology	53

4	Execution Environment Representation	57
4.1	An Ontology Network for the Representation of Execution Environments	58
4.1.1	Our Ontology Network Development Process	58
4.1.2	The WICUS Ontology Network	64
5	Experiment Reproduction	71
5.1	Parsing Tools & Semantic Annotations	72
5.1.1	Workflow Specification File	74
5.1.2	Workflow Parser & Annotator	75
5.1.3	Workflow annotations	76
5.1.4	WMS annotations	77
5.1.5	Software Components Registry	79
5.1.6	Software Components Annotator	80
5.1.7	Software Components Catalog	81
5.1.8	Workflow & Configuration Annotation File	82
5.1.9	Scientific Virtual Appliances Catalog	84
5.2	Specification Process	85
5.2.1	Infrastructure Specification Algorithm	86
5.2.2	Abstract Deployment Plan	92
5.3	Enactment and Execution	94
5.3.1	Script Generator	94
5.3.2	Enactment Files	95
6	Evaluation	99
6.1	Experimental Setup	100
6.1.1	FutureGrid	100
6.1.2	Amazon Elastic Cloud Computing	101
6.1.3	Vagrant & VirtualBox	101
6.1.4	Computational Appliances	102
6.2	Computational Experiments	104
6.2.1	Pegasus	104
6.2.2	dispel4py	111
6.2.3	Makeflow	116
6.3	Results discussion	118

7	Conclusions and Future Work	121
7.1	Conclusions	121
7.1.1	Goals Achievement	122
7.1.2	Hypothesis Validation	124
7.2	Discussion and Future Work	126
7.3	Selected Publications	129
A	WICUS ORSD	131
B	Terminology	135
	Bibliography	137

List of Figures

1.1	Terminology diagram.	18
4.1	WICUS ontology network overview.	65
4.2	Workflow Execution Requirements Ontology overview.	66
4.3	Software Stack Ontology overview.	67
4.4	Scientific Virtual Appliance Ontology overview.	68
5.1	WICUS system overview.	72
5.2	Detailed diagram of the WICUS system.	73
5.3	Pegasus SoyKB workflow annotations diagram.	77
5.4	Makeflow BLAST software annotations diagram.	83
5.5	Workflow requirements description.	87
5.6	Workflow requirements propagated.	87
5.7	Dependency graph example.	89
5.8	Dependency graph removing process.	89
5.9	Overview of the WICUS software stack relation diagram.	92
6.1	Pegasus WMS software dependency graph.	105
6.2	Montage workflow diagram.	107
6.3	Epigenomics workflow.	108
6.4	Epigenomics workflow software dependency graph.	109
6.5	SoyKB workflow.	110
6.6	SoyKB workflow software dependency graph.	111

6.7	Overview of the generated annotations for the Montage, Epigenomics, and SoyKB workflows using the WICUS ontology network (yellow rectangles represent the workflow component; blue squashed rectangles represent the Montage workflow; green bevelled rectangles represent the Epigenomics workflows; and red hexagons represent the SoyKB workflow).	112
6.8	dispel4py WMS software dependency graph.	113
6.9	dispel4py version of the Internal Extinction of Galaxies workflow	113
6.10	Internal Extinction of Galaxies workflow software dependency graph. . .	114
6.11	Seismic Ambient Noise Cross-Correlation workflow.	115
6.12	Makeflow version of the BLAST workflow.	117
6.13	BLAST workflow software dependency graph.	118

List of Tables

3.1	Correspondences among Goals and Hypothesis	52
4.1	Prefix and URI of the ontology network	62
4.2	Concept equivalences	62
6.1	CentOS 6 Appliances	103
6.2	Ubuntu 12 Appliances	103
A.1	WICUS ORSD Slots 1 to 5	131
A.2	WICUS ORSD Requirements	132
A.3	WICUS ORSD Pre-Glossary of terms and objects.	133

Chapter 1

Introduction

“Normal science means research firmly based upon one or more past scientific achievements, achievements that some particular scientific community acknowledges for a time as supplying the foundation for its further practice”

Thomas Kuhn, *The Structure of Scientific Revolutions*, 1962

Reproducibility in scientific studies has been a matter of discussion and controversy in research activities, which can be traced back at least to the 17th century (Shapin and Schaffer, 1985). Scientists and research stakeholders have argued through the history about the importance of being able to repeat scientific experiments in order to trust their outcomes and conclusions, as well as for using them as the foundation of the future advances of science.

Reproducibility of scientific contributions requires the definition of a conservation plan in order to guarantee that the experimental materials and results are being properly handled. The activities that compose this plan vary along different scientific domains, such as the curation of both input and output data, documentation of the involved assets and their context, or publication of the experimental protocol and conditions.

In this work, we explore the challenges related to the reproducibility principles for experiments in computational science, focusing on their equipment and how the required tools for executing them can be conserved and reproduced.

Scientific publications, and the experiments that support them, deserve special attention in the context of reproducibility. These publications are meant to both announce a result of interest to the scientific community and to demonstrate that the exposed claims are true (Mesirov, 2010), by providing enough evidences and materials in order to be verified and understood.

In fact, several controversial studies and scandals have exposed the necessity of establishing the culture of reproducibility among scientific practitioners and communities throughout the history of Science (Ioannidis, 2005). Studies in clinical trials (Ioannidis et al., 2001), pharmacology surveys (Wells, 1997), surveys on psychological science (OSC, 2015), analysis in molecular science (Konig, 2011), and more recent issues in computational science studies (Baggerly and Coombes, 2010), have driven the attention of scientific organizations, such as academic institutions or funding agencies, to the reproducibility challenges. Not only does reproducibility allow verifying scientific findings, but also represents a relevant procedure in the advance of science, enabling the building of new processes based on previous discoveries within the context of a scientific paradigm (Kuhn, 1962).

As a multidimensional subject, reproducibility has to be considered from different angles and perspectives. It also has to take into account the scientific area in which it is being studied, as the requirements imposed by the research culture of the different communities vary from one to another. Requirements for biological experiments to be reproduced usually require a precise description of the individuals involved in the process, whereas social science ones require a characterization of the studied population, in terms of the relevant features in relation with the study. Other areas such as astrophysics do not only require describing the natural phenomena to be studied, but also the apparatus involved in the process to be declared.

Even though a unique definition of what constitutes a reproducible study has not been yet agreed (Begley and Ioannidis, 2015), the different denotations of the term reproducible science share some aspects. One of them is the need to provide access to the information describing the elements involved on a given experiment. Being able to retrieve and understand the knowledge behind the setup of a scientific process is a key point when aiming to clone, under different degrees of exactness, an experimental environment.

The emergence of computational science, that is, science executed using computational models and simulations, has also implied an increasing interest on reproducibility. This recent branch of science adds new challenges and dimensions to reproducibility, intrinsic to its unique characteristics. Most of the resources involved in a computational experiment are defined by digital assets, thus, their reproducibility differs from the physical ones involved in traditional experimental science. How to develop such reproducibility solution is not straightforward and many different approaches have been proposed in the last years.

Our work applies all these principles to the reproducibility of computational experiments, defining the conservation process of a computational experiment as the act of providing understandable information describing the original context of the experiment. More concretely, we target the problem of reproducing the computational environment of experiments conducted in the form of computational scientific workflows. In the following sections we will introduce our approach, framing it in the context of computational science and comparing it to reproducibility approaches in traditional science.

1.1 Computational Science

During the last decades, the rapid growth in computational power and analytical capabilities has allowed scientists to address complex problems of science and engineering by using mathematical models and simulations. In contrast to classic empirical scientific disciplines, which usually rely on laboratory or field experiments for hypothesis evaluation, computational science aims to capture the features of interest of a study from nature and codify them into a computable model.

Several advantages arise from using computational experiments, as they provide a more configurable and faster way of examining scientific evidences. Experiments can be conducted several times, modifying their input data and configuration parameters as required, being more effective in terms of time and cost.

Once a model has been developed and tested, it can be reused by the members of the community, adapting it to the different needs of each experiment. Rather than substituting traditional approaches, computational science complements classical experimental and theoretical disciplines.

Computational science, which originally started as a complementary tool for analyzing the results of experiments requiring large amounts of calculations, has evolved to reach a maturity level that consolidates it as a transversal branch of science. In the following sections we introduce how it evolved from traditional disciplines and discuss its key singularities in the context of scientific reproducibility.

1.1.1 Experimental Sciences Approaches

Based on the environment and context in which scientific studies are conducted, we can distinguish between different types of experiments. From traditional scientific disciplines, to the more modern ones using computational resources for solving scientific problems, each of these categories requires a proper reproducibility approach that focus on their intrinsic needs.

1.1.1.1 In Vivo and In Vitro Science

As introduced before, we can distinguish between two main types of experimental areas in the history of science, depending on the context in which the experiments are performed. We divide these disciplines into two major types of studies; *in vitro* and *in vivo*.

Experiments usually conducted in a laboratory, where samples from nature are studied outside their normal context, fall into the *in vitro* classification. These experiments aim to study concrete organisms and compounds in an isolated way, which allows a more fine-grained and adaptable analysis of them, as they avoid the potential analytical noise due to the interactions generated by the thousands of other elements involved on whole individuals or systems.

Whereas these experiments are probably the most common procedure in scientific areas such as biology or medicine, correlating their results to real scenarios is not always straightforward. In those cases, as well as for studying the experimental effects on living organisms, it is necessary to involve individuals as a whole. These experiments are classified as *in vivo* science, which are conducted as field experiments or in a controlled environment, but using complete representative individuals. Going one step further into this classification, we can distinguish between *in situ* and *ex situ* disciplines, depending on whether the experiments are conducted within the natural habitat of those individuals, or in an artificial environment (such as a zoo or a research institute).

In all cases, experimental procedures are often documented as logs in laboratory notebooks, where the different steps that compose an experiment are recorded lively as the experiment progresses. These notebooks typically belong to the organization where the experiment is conducted, rather than to the scientist performing it. The information written on them includes not only a detailed step-by-step guide to execute the experiment, but also the hypothesis, motivation, and resources studied and used on it.

This information is of extreme value, as it allows the experiment to be verified and traced by its developer and eventually to be repeated. Even when these notebooks are not usually shared publicly, several authors and initiatives have argued about the need to make them available along with the publication describing the results and conclusions of the experiment.

1.1.1.2 In Silico Science

In contrast to the aforementioned types of science, which make use of real entities of nature for their studies, computational experiments study their logical representations. The term *in silico* science was introduced to encompass these type of experiments, where computational models aim to mimic and analyze real phenomena. In some cases, this type of experiments are used as a previous stage of a more complex experimental procedure, allowing the extraction of indications that can be later verified, such as in the case of medical trials, where potential treatments can be analyzed and filtered before testing them in a real *in vivo* process. In other cases, such as analytical studies of astronomical data, these experiments are the only feasible way of processing vast amounts of data for obtaining scientific insights.

Due to the novelty of this type of experiments, there is a general lack of formalization in the way they are described. Even when nothing prevents computational scientific practitioners from using documentation methods such as the aforementioned scientific notebooks, their adoption is of marginal relevance among them. Probably the closest approach used so far in this discipline is the release, under a public license, of the source code and documentation of the software involved in the experiment, as a way of providing insights and evidences about the process. Even when this approach is considered as a best practice in the computational scientific community, being required

in some publications, several studies have shown that it is not sufficient for guaranteeing their reproducibility (Carpen-Amarie et al., 2014).

1.1.1.3 The Challenge of Scientific Reproducibility

Approaching the reproducibility problem on these different types of scientific practices highly varies from one to another. The study of complex entities involved in *in vivo* and *in vitro* experiments, such as animals, cell systems or even human beings, requires an exhaustive and standardized description of the protocols involved. Experiments that seem robust and repeatable in the laboratory may not stand a reproducibility process, as the consistency of the reported results is usually not sufficient enough to include the necessary methodological details.

Solutions such as checklists¹ and standardized reporting protocols have been used and are still discussed within these communities². These approaches aim to indicate authors the necessary theoretical and technical information required to analyze an experiment. Besides, publishing and funding entities are encouraging editors and reviewers to consider these types of additional materials as part of their evaluation process, in order to widespread the reproducibility culture among scientific communities. Even when these approaches contribute to increase the reproducibility degree of a process, elements such as how to recreate the environmental conditions, are still challenging. In some cases, these aspects may not be obvious neither to the author of the experiment nor to the scientist aiming to reproduce it.

Ideally, bringing access to the same environment, being it a laboratory or the field context where the experiment was developed, would help to address such problem, as most conditions would still hold. Such practice is usually not feasible, as geographical and policy restrictions usually hold, preventing it. These limitations are not so strict when dealing with *in silico* experiments, as digital artifacts are easier to share and replicate. Since the very beginning of computational science, along with the irruption of networked systems, computational scientific assets, such as algorithms and datasets, have been built, shared and maintained by scientific communities.

Most of the computational experiment materials can be shared digitally among colleagues and interested audiences, but as well as in the case of traditional experiments,

¹<http://www.nature.com/authors/policies/checklist.pdf>, last accessed 2016

²<http://www.nature.com/nprot/info/gta.html>, last accessed 2016

setting up the proper experimental conditions is not always straightforward. When we rely on digital artifacts for solving scientific problems we have to assume that the developed experiments are subject to some software being available and properly configured (Cerf, 2011).

In this work we aim to tackle this problem, introducing and testing a new solution on how to share the necessary information for reproducing the experimental environment of a computational experiment, in terms of its execution dependencies. As discussed in the following sections, we focus on computational experiments defined as scientific workflows, one the most representative and widespread ways of designing them.

1.1.2 Computational Scientific Experiments

Scientific workflows have gained momentum as a way of representing scientific processes in the form of interrelated computational tasks. As computational science evolved and spread among scientific communities, a way of stating scientific processes in an understandable, exchangeable and potentially executable manner was required. The concept of scientific workflows can be considered as a specialization of the generic workflow concept, which was used as a way of modeling complex business processes in private and public institutions. To this day, such workflows are still used in companies and public agencies for designing and executing analytical processes over large amounts of data, such as bank accounts and population census.

These two types of workflows share the same main principles, although they have different requirements. On the one hand, business workflow systems aim to model processes involving different people and information systems, focusing on defining the control-flow of such processes in terms of order between different activities. On the other hand, scientific workflows are usually defined in terms of the data consumed and generated, being the order of the tasks implicitly defined by the producer-consumer relation. Whereas in business workflows data dependencies are not usually clearly stated, scientific workflows explicitly declare them, defining how data flows through the different path of the workflow. Thus, scientific workflows can be defined as data-intensive processes aiming to simplify large computational scientific tasks (LudÄd'scher et al., 2009).

1.1.2.1 Workflows in Science

Also referred to as *in-silico* experiments, scientific workflows provide a high-level paradigm for workflow design, acting as a common reference framework for grouping data analysis and transformation tasks belonging to a common conceptual procedure. These tasks usually include general steps such as input data filtering and curation, format transformation, and data split and merge. Defining computational experiments using the workflow paradigm provides several advantages, such as the automation and repetition of the process, fault tolerance capabilities, and the record of provenance, both at the task level and at the process as a whole. This increases the repeatability and verifiability degree of such experiments, as they provide a structured way for encoding them.

Scientific workflows are being published more and more frequently as digital artifacts, along with their related publications. These includes not only the description of the experiment itself, but also the additional materials that facilitate their full understanding (Belhajjame et al., 2012). These materials may include any type of information that could help to demonstrate to the target audience of the publication that what is described and stated on it is true. As stated before, scientific workflows can be defined as a precise and executable description of a scientific procedure (Roure et al., 2011). Thus, it is advisable to enrich them with such additional materials, including the proper description of an execution environment, that along with the workflow and the information about the rest of its components, would enable it to be executed.

Several dimensions of the experimental setup must be considered for achieving the reproducibility of a workflow, including the subject of interest of the experiment, the analytical solution defined for studying it, and the means that support the process. We distinguish three main components on a computational scientific experiment, which must be properly conserved to achieve its reproducibility: the data, the scientific procedure and the equipment.

Data components are defined by the input and output data of the experiment, as well as by the potential intermediate results produced during the experiment, which are of value when tracing the execution and validating the overall results. The input data represents the information of the domain we study (e.g. light from the stars, molecule composition, etc.), and is of interest for a certain community and usually requires considerable efforts for being collected. Conserving it is thus of vital importance, both for

reproducing an already created experiment as well as for designing new ones that may obtain different scientific insights from the same data.

The output data is the result of carrying out the experiment (e.g. charts, statistic deviation, plots, etc.) which will enable to verify the hypothesis that the experiment was designed for. This type of data must be properly curated and conserved in order to be shared and corroborated by scientific peers. It is of high importance for repeating an experiment, specially when replicating it, as it allows verifying that the replicated process is successful. Several approaches exist that consider the conservation of input, output and intermediate data, as we will introduce in the following section, since data conservation was one of the first issues considered by the computational scientific community.

Besides the experimental data used or generated by a workflow, the process that transforms and produces it is also of interest and must be conserved. The scientific procedure of an experiment is defined by the set of steps required to perform it, including as many details as possible for guaranteeing its procedural traceability and repeatability. Poor descriptions of scientific procedures, as well as the lack of standards in the computational scientific community, have harnessed the reproducibility of computational experiments.

The adoption of the workflow paradigm for modeling scientific problems introduced a more formal way of doing so, as it provides a more precise and standard description of the scientific procedure. A scientific workflow usually contains several details of each step of the process, and relates how those steps are connected to each other by explicitly stating the data flow connections among them. Similarly, having the procedure defined and encoded under the specification imposed by a workflow language eases the process of sharing it. We will introduce in the following section how the emergence of platforms for workflow cataloguing and sharing has increased the conservation degree of computational scientific procedures.

1.1.2.2 Computational Scientific Equipment

Whereas several initiatives have been undertaken for exploring how to solve reproducibility issues on both the data and the procedural side of scientific experiments, not so many of them consider the experimental apparatus as part of their solution. We consider that this apparatus, which plays a key role on supporting its execution, must

be also taken into account. We can define this apparatus, also referred to as the experimental equipment, as the set of tools and materials involved in the execution of any experimental process. In many cases, the design of an experiment is driven by the features and limitations exposed by the available equipment.

In computational science, this equipment is defined by the infrastructure, that is, the set of computational resources (computers, storage devices, networking, etc.) and software components, necessary for carrying out an experiment. These dependencies, both hardware and software, can be specified in an ad hoc manner for a concrete experiment or can be part of a well-established scientific suite for scientific analysis.

As in the case of traditional science, the equipment of computational experiments is a basic component that sets the foundation for designing and executing them. Thus, the importance of conserving such components is vital for the reproducibility of any computational experiment. Without the proper set of tools being in place and correctly configured it is not feasible to execute a former process obtaining equivalent results.

In order to guarantee its proper conservation, we need to document and describe the tools involved (types, brand, provider, version, etc.) as well as the information for setting it up (calibration, configuration, handling, etc.). When the software artifacts and hardware specifications are not documented and included alongside with the rest of the experimental materials, it is hardly feasible to understand which are the required features that an experimental environment must expose, thus harnessing its reproduction.

1.1.2.3 Equipment Reproducibility

We can distinguish two main different approaches for conserving the experimental equipment, which can be categorized upon how relevant the equipment is and how hard or easy is to obtain an equivalent individual of the tools that compose it:

- Physical conservation: that is, conserving the real object in terms of its physical properties, mainly due to its relevancy and the impossibility or high cost of getting a new equivalent counterpart. The Large Hadron Collider (LHC)³, specialized telescopes for high-energy astronomy, or satellites, are examples of this kind of equipment, due to their singularity. In those cases it is mandatory to conserve

³<http://home.web.cern.ch/topics/large-hadron-collider>

the real equipment and to try bring access to them to the research community, executing the necessary actions for protecting them and prolonging their life;

- Logical conservation: in many cases the equipment used in experimental science can be obtained by most of the research community members. Commonly it is easier to obtain a counterpart than accessing the original tool. Sometimes accessing the original one is even impossible due to its natural decay (e.g. individuals used in experiments such as plants or animals). In those cases, it is more suitable to describe the object so an equivalent one can be obtained in a future experiment: buying the same model and version of a microscope, cultivating a plant of the same species, etc. This requires a precise and understandable description of those elements.

As introduced before, in computational science, the main tools for carrying on an experiment are computational infrastructures, either in the form of software or hardware components. The high amount and variety of requirements of a computational experiment implies a highly heterogeneous environment. Classical initiatives in computational science have faced the reproducibility issue by sharing the infrastructure, bringing access to it within a community of users from related scientific areas. This approach clearly fits on the physical conservation category; an organization, or a set of them, sets up an infrastructure (supercomputers, clusters, grids) for a specific goal and allows some users to access their resources under some conditions. These are usually big and expensive infrastructures that require high effort for setting them up as well as for maintaining them in the long term. These types of infrastructures have proved to be a significant contribution in computational science. For many years, and even nowadays, they have allowed the execution of large experiments in many scientific disciplines and they are still improving the quality and performance of scientific applications.

However, several challenges arise when dealing with this kind of approaches. Within the context of this work we have identified the following ones regarding the infrastructure reproducibility:

- Static infrastructures: these infrastructures require a large technical maintenance effort, as they must be tuned and configured in order to fulfill the requirements of the different experiments developed by the community that has access to it. These

requirements often vary from one experiment to another, requiring an extra effort in coordination and management, to avoid incoherences on the experimental conditions from different executions. This hinders the re-execution of an experiment once the infrastructure has been modified, which eventually will happen within the period of its lifecycle.

- Vendor locking: even though the main purpose of most of these infrastructures is to be shared with as many users as possible, it is not feasible to assume that any organization can guarantee access to their infrastructure to anyone interested on executing or repeating an experiment. Even the users that have access to it have to compete for the resources of the infrastructure. That leads into a dependency between the users and the original infrastructure, which at some point in time may get not supported.
- Long term conservation: guaranteeing the conservation of an infrastructure is not a trivial task. Issues such as projects ending or funding cuts may challenge its future availability. Moreover any infrastructure suffers an aging process, due to their inner nature. Machines get eventually broken or obsolete, new tools appear that are required by new users as new software and hardware requirements are needed, so machines must be replaced. It makes no sense to keep a machine ready to be used forever, as this might even impede the advancement of new scientific processes based on them.

The purpose of this work is to propose a solution that tackles the above-mentioned challenges. We introduce in this thesis a novel approach that aims to contribute to the previous ones, rather than substituting them. We aim to apply a logical-oriented approach to the issue of computational science reproducibility.

Instead of trying to conserve and share the real physical features of an infrastructure, we propose to describe their main capabilities and, based on that description, reconstruct the former infrastructure, or an equivalent one, using virtualization techniques. That is, we will discuss how using virtual machines, as highly customizable computational resources, we are able to generate an infrastructure that is capable of executing a former experiment, under equivalent conditions, and obtaining consistent results.

Virtualization is a mature technique that has been used for the last three decades, and that has lately gained its momentum with the birth of the cloud computing revolution. We argue that by using these techniques we are facing the long-term conservation challenge, as virtualization solutions are mature enough to assume that they will be available in the future. Also, we claim that by having the capability of emulating computational resources independently of the underlying infrastructure, we will be able to expose the same computational features through time.

The use of these technologies introduces some restrictions and assumptions on the scope of the applicability of this work, as we will discuss in sections 3.4 and 3.5. These limitations are mainly related to the performance aspects of the reproduced experiments. Using virtualization may imply a potential loss of processing and storing speed, and guaranteeing certain performance is not feasible, specially when dealing with external virtualization solutions. Therefore, we must take into account that experiments evaluating or considering performance as part of their analysis are not going to be reproducible using our approach. However, after surveying and studying the main characteristics of computational experiments in different areas of science, we can conclude that most of them aim to generate results that are not related to performance metrics.

The purpose of this work is to define a way for describing the capabilities of the computational resources involved in the execution of the experiment, rather than using just virtual machine images with those resources installed on them, what has been a common practice in the area of reproducibility. We argue that even when this kind of approaches serve to the purpose of replicability in the short term, they behave as black boxes, hiding the information of the elements involved in the execution. With a plain machine image it is hard to discriminate which components are tied to the precise experiment being reproduced and which ones are not. As introduced before, the lack of information is one of the key threats for reproducibility, thus, it is advisable to provide not only the computational materials, such as the case of virtual machine images, but also an understandable and shareable documentation illustrating their content.

We aim to provide such kind of documentation in a structured manner, so it can serve as the basis for the reproducibility process. We are introducing in this work a process that, based on these descriptions, will be able to generate a virtual machine exposing the required documented capabilities. As cloud computing, either in the form of private solution or as public infrastructure providers, brings access to almost everyone to the

necessary resources for creating virtual computational nodes, its use within our approach eases the aforementioned vendor-lock problem, by implementing a common and shared pool of resources, where any researcher could reproduce scientific applications.

As mentioned before, virtualization techniques allow the customization of resources, enabling the definition of specific infrastructure configurations customized for each experiment. This way we will be able to provide a highly dynamic shared infrastructure, that could automatically offer a customized set of capabilities adapted to the different experiments being reproduced on it. This would ease the configuration and maintenance process required by large infrastructures on classical approaches. By using this kind of approach we are relying on well-known providers and solutions for the management effort, rather than on scientists and institutions staff.

We envision two main contributions to be developed in the context of this work, in order to fulfill the requirements exposed by these objectives : i) a representation framework to support the generation of a set of shareable catalogues containing the required information, and ii) a set of computable processes for generating and consuming this information in order to define an equivalent environment.

The first goal of our solution is to provide means for sharing the information describing the requirements of a computational experiment. This information must be specified in a way that can be interpreted and understood, otherwise it is of no use for its target audience, usually a scientist aiming to reproduce an experiment. To this end, we need to define a representation framework that allows encoding the knowledge about the execution requirements in a structured, shareable and self-contained way. Based on this criteria, we have selected semantic technologies as the basis for developing such framework, as they provide a reference model for building conceptualization of domains in a structured manner.

Using this set of models, we must be able to generate the corresponding datasets, which will contain the descriptions of the execution components involved in an experiment, as defined by them. As it is not feasible to assume that we will be able to deal with the whole range of scientific applications and computational resources available, we need to develop a scientific catalogue that will contain a set of representative virtual appliances as well as the software components involved on the experiments being reproduced. These catalogues will be the basis for the process of generating the infrastructure configuration that will support the new execution of the experiments.

These catalogues will constitute a knowledge base that may be populated collaboratively by the members of a scientific community, serving as a common pool based on which adaptive infrastructure specifications can be generated. This information, when properly shared and conserved, can be of great value to a community, allowing not only experimental reproducibility, but also the reusability and repurposing of the described experiments.

In order to populate these models, it is necessary to provide a set of tools for extracting the information from the different potential sources containing the list of resources and their configuration properties. This process must be as much automated as possible in order to reduce the number of potential errors and to alleviate the annotation effort from the developers of an experiment. However, as we will discuss later in this thesis, a fully automated process is hard to achieve due to the nature of scientific software, which many times do not explicitly define the required properties. Thus, manual intervention from users, either scientists or supporting staff, must be allowed.

Finally, in order to prove that the information defined by the models and contained in the catalogues is complete enough for reproducing the execution environment of an experiment, it is mandatory to define a computable process, able to read the mentioned information and specify a new incarnation of it, supporting the former required features. Such process must be able to analyze the different descriptions available and based on a defined criteria, select the resources to be created and their corresponding configuration, taking into account the necessities of the target workflow.

The design and implementation of this process would depend on the needs and priorities of the scientists reproducing experiments. Depending on the selected criteria the algorithm developed would be different, and therefore, designing a unique solution is not feasible. In this work, we aim to provide one of the many possible implementations for showing how an algorithmic process meeting this goal would work.

1.2 Scientific Conservation and Reproducibility

Many different definitions of the terms related to the conservation and reproducibility aspects of science, both in classical and digital fields, can be found on the literature. In this work we aim to justify how those concepts are defined in the context of this thesis, in order to clarify its goals and its scope of applicability.

The border between conservation and preservation is often blurry, and it highly depends on the ultimate goal of the process or on the inner nature of the object studied. In both cases, the main objective of these concepts is to prolong the existence of an object of interest, retaining its relevant features as close as possible to their original form. The main difference between them resides in the way they allow access to those objects. Whereas preservation processes prevent any kind of interaction with the elements being preserved, conservation approaches define plans for guaranteeing a proper and feasible interaction.

Similarly, distinguishing between terms such as reproducibility and replicability in the context of scientific experiments is not always straightforward. The granularity and the level of abstraction under which they are being considered makes a repeated experiment fall into one category or another. It is commonly agreed that replicating an experiment consists in producing an exact copy of a former one, using the same input data, under the very same parameters and mimicking the experimental conditions, in order to obtain the same results. When any of these conditions is modified, the experiment is considered to be a reproduction of the original one, instead of a replica. While the first one is highly relevant for verifying scientific outcomes, the second allows to repurpose an experiment, applying its principles to new areas or improving its behavior by modifying some of its components.

In the context of this work, we provide definitions for these concepts within the domain of digital artifacts, which is the general framework of this work. Several other concepts, which are often related to scientific conservation and reproducibility, are listed in Appendix B.

- Replicability: an exact repetition of a process done by someone else or by the same individual certain time ago, using the same elements involved on the original process or a close copy of them.

The main purpose of replicability is usually the corroboration of previous results and many scientists consider it as a mandatory requirement for accepting the outcomes of an experiment to be true.

- Reproducibility: repeating an experiment pursuing the same or similar goals as the original one, but introducing at least one significant variant to it in order to customize its behavior.

The goal of a reproducibility process is to follow the same spirit as the one the original experiment was designed for, mainly aiming to broaden the applicability of the provided solution to a larger scope or to a new area.

- **Preservation:** the process of maintaining the features of an object of interest, storing it in a proper manner and avoiding any potential risky interactions, aiming at making it last as close to its original form as possible. In digital preservation processes, in contrast to physical ones, it makes no sense to avoid the human interaction with the preserved object, as a copy of it can be obtained in most cases without harnessing the original component. Thus, the purpose of a digital preservation process is to maintain the ability of accessing the essential elements of the studied object.
- **Conservation:** the process of describing an entity of interest, collecting as much information for describing its original context, cataloguing its main capabilities, and identifying its purpose. Conservation processes are designed for allowing external interaction, including its use, under certain conditions for minimizing its decay and making it last as long as possible. In the context of digital assets, such as software entities, conservation plans must take into account that they are subject to an evolution process, and rather than avoiding it, they have to be designed in a way that these changes are documented and understood by the audience accessing them.

These definitions are depicted by Figure 1.1, which shows an illustrative example of how they would apply to a physical object (i.e. a typewriter). On the left hand side of the figure, we see how preservation is a more restrictive approach, preventing any interaction, while conservation focuses on providing descriptions and plans for minimizing the damage risk, allowing it to be used. On the right hand side, we can distinguish between a replication of the object, an exact copy of the original one, and reproduction, a new version of it, retaining its original features and potentially providing new ones.

In this work we focus on the **reproducibility issues related to the scientific infrastructure of computational experiments**, concretely on the reproducibility of execution environments, considering them as digital artifacts that must be properly conserved in order to be reproduced in the future. We define our solution as a conservation approach, as we aim to catalogue and document the relevant features of the

Figure 1.1: Terminology diagram.

studied execution environment components, describing their original context in terms of required elements and how they can be used in a new environment.

The goal of this conservation process is to achieve the reproducibility of the execution environment, by providing a new equivalent infrastructure according to the scientific workflow in consideration. The infrastructure generated may not be able to reproduce all the general capabilities of the former one, but it must be able to support the necessary ones for executing the workflow. While the components involved on the execution are mainly an exact copy of the original ones, the overall infrastructure configuration is not, containing potential variations on the way it is arranged.

Throughout the following chapters we will introduce our approach, highlighting and exemplifying how these terms fit into each of the contributions stated before.

1.3 Thesis Structure

This thesis is divided into three main blocks. We start by discussing the state of the art and the work objectives derived from it, followed by the contributions of this work, and then finally we introduce the experimentation process used for evaluating the ideas of this work and the conclusions we obtained.

In this first chapter we have introduced and motivated the overall concepts and ideas of our approach, aiming to state in a clear way which are the problems that we are facing and which are the means we are proposing for solving it.

Chapter 2 describes the main infrastructures used in computational science, both for the data and computation elements of computational experiments. We also discuss the main approaches defined by the time of this work for attaining the reproducibility problem on computational science. We categorize these approaches based on the area of an experiment that they focus on, namely the scientific procedure, the data assets, and the equipment.

Chapter 3 states in a more formal way the objectives of this work, defining the limitations of our approach and the goals that we aim to achieve. We also define our research hypothesis and the research methodology that we have followed during its development.

Chapter 4 introduces the first contribution of this work, the framework for information representation. We will describe the way we have designed and developed it, explaining the methodology followed during the process and the main concepts that compose the generated models.

Chapter 5 describes the second contribution of this work, composed by the set of tools that we have developed for generating the required information and the methods we have implemented for consuming it and generating new reproduced environments.

Chapter 6 introduces the experimental process we conducted in this work in order to evaluate our research hypothesis. It describes the defined experimental setup and the use cases involved on our evaluation, which represent real examples of the reproducibility process from several computational experiments belonging to different areas of science. We explain the results obtained for each of them and discuss the overall outcome of the experimentation.

Chapter 7 wraps up this work by extracting the main conclusions derived from this thesis, discussing their relevance and setting up the future lines of work.

Chapter 2

State of the Art

“If I have seen further, it is by
standing on the shoulders of giants.”

Issac Newton, 1676

Long before terms such as Computational Science or eScience were coined, computers and computational resources were used in Science. Actually it was science which motivated some of the most remarkable progress in computer science history. From the definition and partial implementation of the Difference Engine by Charles Babbage for computing polynomial functions, to the introduction of the World Wide Web by Tim Berners-Lee for exchanging information among scientists at CERN⁴, computer science achievements have been widely driven by scientific purposes.

This chapter aims to summarize the main contributions on Computational Sciences infrastructures, both from the data and the computation point of view, and to introduce some of the most relevant lines of work to address the reproducibility problem in eScience.

2.1 Data Infrastructures for eScience

Experimental data is one of the most fundamental assets in any area of scientific research. We can distinguish three main types of data when studying a scientific experiment: input and output data, and intermediate results, all of them highly valuable

⁴<http://home.web.cern.ch/>, last accessed 2016

and thus worthy to be properly handled. On the one hand the input data, which represents the phenomenon that a scientist is willing to study, can be considered as the starting point of any experiment. In most cases, the experimental procedure itself is derived from the inner structure of the data. On the other hand, the output data, generated as a result of an experiment, is an extremely valuable resource that must be properly handled. Scientists must ensure that these results are adequately preserved and communicated, and as far as possible, allow the rest of the community to use them.

Intermediate data, which can be considered as the output data of intermediate steps, can be also of high value, as it allows tracing the steps along the execution of the experiment and verifying the correctness of both the experiment and the final data.

The evolution of storage systems as well as the increased computational capacity for processing data has led to a new scenario where datasets play a key role on scientific production. Large experiments such as LHC or SKA produce vast amounts of data, reaching the Petabyte scale every year. The exploration of this data has derived into the so-called *4th paradigm* of Science (Hey et al., 2009), also referred to as eScience, where the analysis of large amounts of data may generate new scientific contributions and theories.

Even though in this work we do not specifically focus on data issues related to scientific problems, it is worth introducing some of the most relevant initiatives on scientific data management, due to its tight relation with scientific computational infrastructures. In the digital age of science, there is no way of conceiving the data without the computational means (i.e. software and hardware tools) to process it (Cerf, 2011), and vice versa.

Several large initiatives were started worldwide in the early 2000s in order to support the data management needs of scientists from different domains, including collection, curation, preservation, and publication, among others. Examples of this kind of initiatives include centers such as the National Center for Atmospheric Research (NCAR)⁵, for providing support to Earth scientists data, cyber-infrastructure like the Open Science Data Cloud (OSDC) (Grossman et al., 2012), which provides different scientific communities with resources for sharing and analyzing large amounts of data, or the UK Digital Curation Center (DCC) (Rusbridge et al., 2005), targeting the curation and preservation of the digital information produced by UK institutions. Other more recent

⁵<http://ncar.ucar.edu/>, last accessed 2016

initiatives, such as OpenAIRE⁶ or EUDAT⁷, promote research data sharing produced within the EU context.

All these initiatives aim to support the data storage, sharing and access processes for scientists from different communities. They provide means for uploading the data produced by scientific experiments, as well as the data retrieved from observations, so that it can be used by communities of a scientific domain. This data is catalogued and documented, either by the users providing it or by the project's staff, in such a way that it can be later browsed, promoting its discovery. Most of the times, the data handled by these services is provided under a public access license, being it open to any scientist interested on them and increasing their target audiences. Long-term preservation of the managed data assets is also a goal of these initiatives.

New initiatives for data management are still to come, since new projects are challenging current solutions. The already mentioned SKA telescope, which is actually composed by a set of radio telescopes that once built will cover an area of approximately one square kilometer, will produce several petabytes of data each year. Even when the telescopes are still under construction by the time of this work, their main component for handling data has already been designed. The SKA's Science Data Processor is the component in charge of processing, moving and taking decisions in almost real-time about huge amounts of data, at an expected ratio of hundreds of gigabits per seconds. By 2020, the starting date for the project, it will generate and process about 1 exabyte of data, that is, around the amount of data handled on internet during 2000 (Drake, 2014).

2.2 Computational Infrastructures for eScience

Data-driven scientific experiments do not only require infrastructures for data management. The challenges and opportunities raising from the increase in the size and variety of the scientific datasets to be handled has also led to important advances in the area of computational sciences applied to different disciplines.

On the one hand, many specific solutions have been developed focusing on the specific computational requirements of data-driven scientific experiments. On the other

⁶<https://www.openaire.eu>, last accessed 2016

⁷<http://eudat.eu/>, last accessed 2016

hand, the rapid increase on the need for computational power demand in several scientific disciplines, such as astronomy or biology, has led to the development of several paradigms and tools for managing and sharing computational infrastructures across teams and projects, from local laboratories up to global-scale facilities.

In the following sections we will cover some of the most relevant developments in Computational Science, both in terms of software and hardware infrastructure, which have been applied to scientific experiments.

2.2.1 Distributed Computing

After the introduction of the first networked systems on the 1960s and the 1970s, the first formal contributions on distributed computing started to appear. This new paradigm started to become popular during the following decades and it was applied to solve heavy computational problems. What started as a solution implemented mainly on localized environments, rapidly reached to a global-scale level, where scientists from different parts of the world shared their resources. We can talk about several different paradigms targeting this goal, such as Cluster, Grid and Cloud computing.

2.2.1.1 Grid Computing

On the mid 1990s, the term Grid Computing (Foster and Kesselman, 1999) was introduced as a concept for homogenizing the different existing approaches on distributed computing, such as Cluster Computing, sharing them in a non-dedicated way. Grid Computing uses the analogy of public utilities, such as electricity, water or gas, to distribute applications over multiple computers belonging to several administrative domains.

Those resources are usually shared and exploited by Virtual Organizations (Foster et al., 2001), which can be defined as a group of researchers, belonging to different institutions, that share a common scientific interest. Virtual Organizations are coordinated under a set of agreed rules to ensure a *“flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources”*.

With the advent of Grid Computing, the Open Grid Service Architecture (OGSA)⁸ was introduced by the Global Grid Forum (GGF) (one of the former institutions that

⁸<http://www.globus.org/ogsa/>, last accessed 2016

merged later into the Open Grid Forum (OGF))⁹, becoming the de-facto standard for designing grid solutions. Based on it, many initiatives appeared offering access to computational resources in a more accessible manner. OGSA defined those resources using XML descriptions for accessing them remotely. As Grid Computing embraced the service-oriented paradigm, the concept of Virtual Organization evolved to consider it as the set of services provided and consumed by its participants.

During the early 2000s, due to the irruption and popularity of the Semantic Web (Berners-Lee et al., 2001) and its related technologies, Semantic Grid (De Roure, 2005) emerged as a term for defining an extension of the original Grid approach. The main goal of the Semantic Grid was to define grid components, resources, services and roles, in a structured manner, both human and machine readable, following the Semantic Web principles. This approach was meant to facilitate the resource discovery problem, increase the interoperability of different grids, and ease the deployment of new services.

The emergence of Grid computing and its popularity was partially led by the appearance of several tools and frameworks for turning available resources into grid resources within institutions. From simple job submission systems to more complex environments supporting complex scheduling and resource allocation, a wide range of solutions was developed.

Systems such as Portable Batch System (PBS) (Henderson, 1995), Oracle Grid Engine (formerly known as Sun Grid Engine) (Microsystems), 2001), and Condor (Litzkow and Livny, 1990) were queue managers designed to distribute computational tasks among local distributed resources. Other platforms, such as UNICORE (Uniform Interface to COmputer REsources) (Romberg, 2002), provided a more robust and user-friendly way of managing jobs, including a GUI and allowing the submission of multi-part applications.

The Globus Toolkit (Foster, 2005), an open source implementation of several grid standards, can be considered one of the most popular framework in Grid computing. Globus is composed by several modules for file transfer, security, execution and reporting features. It is also worth mentioning gLite (Laure et al., 2006), a grid middleware that evolved from the Globus Toolkit and was in charge of managing most of the computational resources hosted by the LHC experiment¹⁰.

⁹<https://www.ogf.org/>, last accessed 2016

¹⁰<http://lhc.web.cern.ch/lhc/>, last accessed 2016

Furthermore, several initiatives have been undertaken worldwide to provide researchers with resources for conducting scientific discoveries, being many of them still available at the time of writing this thesis. The following is a list of the most relevant ones in terms of resources and usage:

- Open Science Grid (OSG) (Pordes et al., 2007): managed by the Open Science Grid Consortium, was founded in 2004. This US-based Grid facility comprises more than 25.000 nodes and delivers over 100 million CPU hours a year, being most of them dedicated to Physics simulation and data analysis for LHC.
- The European Grid Initiative (EGI) (Kranzlmler et al., 2010) emerged as a continuation of the Enabling Grids for eScience in Europe (EGEE)¹¹ project, which ran from 2004 to 2010. Since that year, EGI coordinates and manages the resources belonging to the different National Grid Infrastructures (NGI) from countries in the European Union.
- TeraGrid & XSEDE: Teragrid (Beckman, 2005) was an NSF-founded project coordinated by four US institutions: the Argonne National Laboratory¹², National Center for Supercomputing Applications¹³, the San Diego Supercomputer Center¹⁴ and the Center for Advanced Computing Research¹⁵. Started in 2001, it was developed to become the main distributed national facility and was successfully extended up to 2011. After the end of the TeraGrid project, all its resources and facilities were migrated to the Extreme Science and Engineering Discovery Environment (XSEDE) (Towns et al., 2014) project, which is run from 2011 and is supposed to end in 2016. Among the facilities supported by XSEDE, we can highlight FutureGrid (von Laszewski et al., 2010), which supports experimental computer science research and is used in the context of this work.
- Worldwide LHC Computing Grid (WLCG) (Shiers, 2007): LHC is probably the project that has led the development of most distributed infrastructures in the last two decades. As mentioned before, many Grid initiatives have supported the

¹¹<http://eu-egge-org.web.cern.ch/>, last accessed 2016

¹²<http://www.anl.gov/>, last accessed 2016

¹³<http://www.ncsa.illinois.edu/>, last accessed 2016

¹⁴<http://www.sdsc.edu/>, last accessed 2016

¹⁵www.cacr.caltech.edu, last accessed 2016

execution of physics analysis from the LHC. Founded in 2006, it is the CERN's main facility for processing the experimental data produced by LHC. It is considered to be the largest Grid facility ever accomplished, aggregating more than 170 facilities all around the world. The WLCG resources are managed using the aforementioned gLite grid middleware.

2.2.1.2 Cloud Computing

Due to the increase of computational resources in market companies and academic institutions, partially related to Grid initiatives, and the improvements on virtualization technologies, a new paradigm on distributed computing arose on the mid 2000s. Cloud Computing (Foster et al., 2008) pursues some of the goals of Grid Computing, and to some extent it can be considered as an evolution of it.

While Grid computing was mainly science-oriented since its very beginning (even if some commercial grids were developed), Cloud Computing followed a market-driven approach. In many ways Cloud Computing was able to achieve the facility-like approach that Grid Computing did not fully met. This was due to the virtualization approach used in Cloud Computing, making most of the resources fully customizable and disposable for users.

Most of the aforementioned Grid initiatives and infrastructures have evolved to support Cloud Computing, providing the computational foundation for exposing virtualized services on top of them. As a market-oriented approach, Cloud Computing offers services on demand, normally charging users based on a pay-as-you-go model.

Even though its original business-oriented model, Cloud Computing was rapidly adopted by computational scientists as an ubiquitous, scalable, secure and easy-to-configure approach for accessing computational resources. We can distinguish between two main types of Cloud Computing facilities; public or private, depending on whether they are accessible within the context of an organization (or a group of them) or they can be exploited commercially by any user. In some cases, hybrid solutions can be also identified, where public and paid resources are accessed when the capacity of a private solution is overwhelmed due to computational demand peaks.

Cloud Computing solutions offer their resources using a service-oriented model. These services are generally classified on three tiers, based on the level of abstraction

that they provide for the end user. The three main levels for this classification, explained below, can be seen as an incremental stack, as they usually rely on the previous level for supporting its execution.

- Infrastructure as a Service (IaaS) offers computational resources, such as virtual machines or storage capacity, using virtualization technologies. Resources can be allocated and used for fixed periods of time or dynamically.
- Platform as a Service (PaaS) provides a framework where users can deploy their applications to be executed on the provider's infrastructure. Usually these services can also provide automatic ways for scaling and balancing the deployed application dynamically.
- Software as a Service (SaaS) brings access to users to fully functional software. This software can be remotely exploited, usually by using a web browser, without the need of downloading and installing it on the user's local environment.

Unexpectedly, the most popular cloud provider and one of the pioneer companies in Cloud Computing spread was not a technology company. In 2006, Amazon, whose main business started as an online retailer, released an IaaS solution, known as Amazon Web Services Elastic Cloud Computing (AWS EC2)¹⁶, which was rapidly adopted by the community due to its competitive pricing model. In a few years, Amazon became the main cloud provider, adding new services such as storage and network, and defined the standard in the area.

Technology companies soon realized a business opportunity on Cloud Computing services. Companies such as Microsoft or Google introduced their own cloud platforms. Microsoft Azure (formerly known as Windows Azure)¹⁷ is a solution mainly focused on the PaaS level introduced in 2010 that supports several programming languages, allowing users to deploy their applications. More recently, in 2014, IBM introduced its own PaaS platform, IBM Bluemix¹⁸, which is based on the Cloud Foundry¹⁹ open source solution, also supporting several programming languages and APIs. In 2006, Google

¹⁶<http://aws.amazon.com/ec2>, last accessed 2016

¹⁷<http://azure.microsoft.com/>, last accessed 2016

¹⁸<http://www.ibm.com/cloud-computing/bluemix/>, last accessed 2016

¹⁹<https://www.cloudfoundry.org/>, last accessed 2016

introduced Google Apps for Work²⁰, a software productivity solution at the SaaS level, which provides users with a set of web-based applications for managing email, storage, document and communications. The different applications that compose the suite were developed and released almost every year since it was introduced.

As mentioned before, many of the existing Grid initiatives evolved to support Cloud Computing services. This was possible due to the development of several projects that eased the migration from Grid facilities to cloud solutions. Most of these tools are focused on transforming Grid computational resources into IaaS resources. Some of the most popular cloud solutions are described below.

- OpenNebula (Milojicic et al., 2011) is an open-source project started in 2008 that provides IaaS capabilities allowing the building of private and public clouds. It supports different hypervisors for virtualizing resources and provides several cloud interfaces, including the AWS EC2 specification, exposing the same capabilities as the Amazon provider.
- OpenStack²¹ was created as a result of NASA’s Nebula²² project, a Cloud Computing platform for research projects. OpenStack, established in 2010, evolved to become an open-source project run by its own community. It contains several components in charge of the different IaaS capabilities exposed by OpenStack, from computing resources, to databases or data analysis. OpenStack was highly adopted as a cloud solution for many of the US cloud facilities.
- Nimbus (Keahey and Freeman, 2008) is a toolkit aiming to provide scientists access to computational infrastructure under the IaaS paradigm. Nimbus aimed to leverage the cloud features to the average user. Among the tools integrated in Nimbus, we can highlight cloudinit.d (Bresnahan et al., 2011) as one of the pioneering tools for VM customization. This tool has been widely adopted by the community and is also used in this work, as explained in section 5.3.

Scientific communities were one of the earliest adopters of Cloud Computing, as they previously were with Grid Computing. Several organizations introduced some of the

²⁰<http://www.google.es/Apps/Work>, last accessed 2016

²¹<https://www.openstack.org/>, last accessed 2016

²²<http://www.nasa.gov/open/nebula.html>, last accessed 2016

main principles of this new paradigm to their research routine, mainly by virtualizing their computational experimental resources. For example, scientists at CERN were encouraged to use CernVM (Buncic et al., 2010) as a way of sharing their experimental environments, hence reducing the installation of the experimental software, as well as serving as a seamless interface to the underlying grid resources.

Next we summarize some of the most relevant Cloud Computing initiatives for scientific purposes.

- Red Cloud²³, hosted by the Center for Advanced Computing at Cornell University²⁴, offers a subscription-based access to cloud resources as well as support from cloud specialists from the hosting university. In the same way market companies have got into the academic field, Red Cloud offers commercial cloud services for external users.
- CERN Cloud (Drake, 2014) is the in-house cloud facility at CERN to process the data generated by experiments such as the LHC. This infrastructure is a clear example of the evolution of grids into clouds, as it lies on the previously built Grid facilities at CERN. It allows scientists to obtain new hardware faster than before by using customized virtual resources, and brings on-demand computational capacity and data storage.
- Magellan (Yelick and andBrent Draney andRichard Shane Canon, 2011), hosted by the Argonne National Laboratory, supports scientific workloads so as to explore the impact of different computing models and hardware designs over scientific applications. Several scientific experiments, belonging to different areas, such as high energy and nuclear physics, were tested under Magellan for benchmarking as well as to produce novel scientific results.
- The EU Helix Nebula (Jones, 2015) aims to lower the trust and legal barrier from research institutions to access resources from a public cloud provider and study the costs of moving to the cloud using real experiments (such as CERN’s ATLAS²⁵) to measure it.

²³<https://www.cac.cornell.edu/redcloud/>, last accessed 2016

²⁴<https://www.cac.cornell.edu>, last accessed 2016

²⁵<http://atlas.ch/>, last accessed 2016

2.2.2 Workflow Management Systems

As introduced before, scientific workflows represent scientific methods designed for a concrete problem in a formal way. They allow scientists to represent multi-step computational tasks. For example, retrieving data from an instrument or a database, re-formatting the data, and running an analysis generating a table or plot as output. These workflows are described using high-level abstraction languages, which conceal the complexity of execution infrastructures to the user.

In most cases workflows are described as directed acyclic graphs (DAGs), where the nodes represent individual computational tasks and the edges represent data and control dependencies between tasks. Workflow interpretation and execution are handled by a workflow management system (WMS) that manages the execution of the application on a computational infrastructure.

In recent years several Scientific Workflow Management Systems (WMS) have arisen as a user-friendly way for designing and implementing computational scientific procedures under the workflow paradigm, providing GUIs and tools for easing the task of handling large and complex computational processes in science. WMS can be considered as part of the computational infrastructure, as the necessary middleware for abstracting and orchestrating the scientific procedure. Here we describe some of the most popular WMS, which are normally associated to specific scientific communities.

2.2.2.1 Galaxy

Galaxy (Goecks et al., 2010) is a web-based WMS that aims to bring computational data analysis capabilities to non-expert users in the biological sciences domain. The main goals of the Galaxy framework are accessibility to biological computational capabilities and reproducibility of the analysis result by tracking the information related to every step on the process. The Galaxy workflow model does not follow the DAG paradigm, as it allows to define loops, being a directed cyclic graphs (DCGs) approach.

In Galaxy, a workflow is considered as an abstract recipe, not attached to any specific data source. When the data to be analyzed is specified, workflows are considered as “*Histories*”. A single workflow can be concretized over several data sources, producing many “*Histories*” and results. These results can be produced as Galaxy “*Pages*”, a sort of publication within the context of the framework describing the experiment.

Galaxy can be deployed locally or on the cloud through the Galaxy CloudMan component (Afgan et al., 2010).

2.2.2.2 ASKALON

ASKALON (Fahringer et al., 2005a) is a UML-based, grid-oriented, and platform independent WMS. Workflows are defined using the Abstract Grid Workflow Language (AGWL) (Fahringer et al., 2005b), which is based on the UML 2.0 notation. The main aim of ASKALON is to hide the complexity of grid computing to end users. It allows defining a workflow either by using its associated GUI or by editing the AGWL file.

ASKALON uses Globus Services for Grid infrastructure management. In order to enable workflow execution on ASKALON grids an application must be described using two files: the GWDD (Grid Workflow application Deployment Descripton) and the *buildFile*. Both files must be registered to the GLARE, a *“component of resource management that deals with logical resources including components, legacy scientific application (executables) and Grid/Web services.”* (Fahringer, 2011). The GWDD file contains the description of the application and its activities. The *buildFile* defines the required steps for an automatic or on-demand deployment of the application.

2.2.2.3 Taverna

Taverna is a Web Service-based WMS, as all the components of the workflow must be implemented as web services (either locally or using an available remote service). Taverna is able to integrate Soaplab²⁶, REST (Fielding, 2000) and WSDL (Christensen et al., 2001) web services. It offers a wide range of services for different processing capabilities, such as local Java services, statistical R processor services, XPath scripts, or spreadsheet import services.

The main way of interacting with Taverna is through its GUI, but a Command Line Interface (CLI) is also provided, mainly for managing workflows remotely. By the time of this work several languages have been supported by Taverna. The first version of the WMS worked with SCUFL, a data-driven DAG language. For Taverna 2, the language was changed to t2flow and SCUFL2 was adopted for Taverna 3, which aimed

²⁶<http://sourceforge.net/p/soaplab/wiki/Home/>, last accessed 2016

to add semantic capabilities for describing the workflow structure based on the SCUFL2 Ontology. However, this ontology was not publicly available by the time of this work.

This WMS is integrated with myExperiment (Roure et al., 2009), a project for sharing experimental results and scientific results, so the user can login and access workflow examples and even upload its own workflows to be shared with the community. Taverna, as well as the myExperiment framework, were part of the myGrid²⁷ consortium.

2.2.2.4 Kepler

Kepler (Ludäscher et al., 2006) is a multi-platform WMS built upon Ptolemy II (Eker et al., 2003b). Kepler workflows are written in MoML (an XML format) or KAR files, which are an aggregation of files into a single JAR file.

The main components of any Kepler workflow are *Directors* and *Actors*. The former ones control the execution of the workflow, while the actors execute actions when specified by directors. *“In other words, actors specify what processing occurs while the director specifies when it occurs”*²⁸.

This actor-oriented way of defining workflows is complemented by *Ports*, as the input/output data points, *Relations*, the edges representing the data flow among *Ports*, and *Parameters*, configurable values that serve as input of the workflow components.

2.2.2.5 KNIME

KNIME (Berthold et al., 2007) is a WMS mostly known for data mining. It is a free and open-source project that is also available under a commercial license sold by the KNIME company.

It is an IDE-based system and all the components (or at least the majority of them) that can be used to implement workflows, are included on the KNIME distribution. Usually, these components are developed in the form of plugins by scientists and/or community developers, and are published in trusted repositories which are listed on the KNIME documentation.

KNIME workflows are mainly, when not exclusively, designed through its GUI.

²⁷<http://www.mygrid.org.uk>, last accessed 2016

²⁸<https://code.kepler-project.org/code/kepler-docs/trunk/outreach/documentation/shipping/2.4/getting-started-guide.pdf>, last accessed 2016

2.2.2.6 Pegasus

Pegasus (Deelman et al., 2005) is a WMS able to manage workflows comprised of millions of tasks, recording data about the execution and intermediate results. In Pegasus, workflows are described as abstract workflows, which do not contain resource information, or the physical locations of data and executables.

The abstract workflow description is represented as a DAX (DAG in XML), capturing all the tasks that perform computation, the execution order of these tasks, and for each task the required inputs, expected outputs, and the arguments with which the task should be invoked. During a workflow execution, Pegasus translates an abstract workflow into an executable workflow, determining the executables, data, and computational resources required for the execution.

Pegasus maps executables to their installation paths or to a repository of stageable binaries defined in the Transformation Catalog (TC). A workflow execution includes data management, monitoring, and failure handling. Individual workflow tasks are managed by a task scheduler (HTCondor (Litzkow and Livny, 1990)), which supervises their execution on local and remote resources.

2.2.2.7 WINGS

WINGS (Gil et al., 2011) may not be considered as a proper WMS by itself, as it does not provide workflow enactment and execution features. However it is widely known for workflow design. WINGS can be seen as a top-level and domain-oriented design tool whose workflows can be later enacted in different workflow engines, such as Pegasus or Apache OODT²⁹.

WINGS was designed as a semantic-driven composition tool, where workflow components and datasets are suggested and restricted by a set of defined rules and restrictions. For composing workflows, WINGS provides a web-based interface that can be accessed remotely or locally by deploying WINGS on a local server.

A catalog of components belonging to different scientific domains is available for designing WINGS workflows. These workflows and their related datasets can be validated, and metadata about the results can be also generated for new data products.

²⁹<https://oodt.apache.org/>, last accessed 2016

2.2.2.8 dispel4py

dispel4py (Filguiera et al., 2014) is a Python (Rossum, 1995) library for describing workflows. It describes abstract workflows for data-intensive applications, which are later translated and enacted in distributed platforms (e.g. Apache Storm, MPI clusters, etc.).

Workflow activities are defined as processing elements, which can be composed by a subgraph. These elements can be imported from a registry or created and registered by users.

dispel4py provides means for enacting its workflows over different engines. Currently, four main types of engine are supported: sequential, multiprocessing, MPI (Forum, 1994), and Apache Storm³⁰.

To design dispel4py workflows a set of CLI commands are provided as well as an IPython (Pérez and Granger, 2007) extension, which allows executing the dispel4py commands using IPython.

2.2.2.9 WS-PGRADE/gUSE

gUSE (grid and cloud User Support Environment) (Kacsuk et al., 2012a) is a science gateway for EGI resources. WS-PGRADE (Kacsuk et al., 2012b) is the evolution of PGRADE (Kacsuk, 2011) (which is almost deprecated) and is based on gUSE. Actually, WS-PGRADE can be seen as the presentation tier of gUSE.

WS-PGRADE uses its own XML-based workflow language to represent DAG-based workflows, with support for embedded subworkflows. Workflows are composed of jobs and ports for connecting them. It allows the definition of executables for each job, which can be referenced locally or remotely.

There are two levels of workflows: abstract and concrete workflows. Abstract workflows are defined by the graph without the semantics of its nodes. A concrete workflow is a concretization of an abstract workflow with the semantics of the nodes (executables, input/output files, etc.) and the DCIs where it is going to be executed.

Workflows are designed to run over EGI resources and therefore are usually within the context of a VO. WS-PGRADE also adds support for cloud execution of the workflows.

³⁰<https://storm.apache.org/>, last accessed 2016

2.2.2.10 Makeflow

Makeflow (Albrecht et al., 2012) is a WMS that extends the idea of classical makefiles to build workflows composed by large numbers of tasks. Workflows can be written in a technology-neutral language, agnostic of the underlying batch system or distributed computing solution. As it follows a makefile approach for designing, the Makeflow workflows lie under the DAG paradigm.

It supports many different systems for job submission, including HTCondor (Litzkow and Livny, 1990), PBS (Henderson, 1995), OGE/SGE (Microsystems), 2001), Torque (Staples, 2006), and the Work Queue system, its own built-in batch solution.

The software components involved on the workflow, as well as the input and output data files consumed and produced, are explicitly declared on the Makeflow file.

2.2.2.11 SHIWA/ER-flow

The SHIWA framework is part of the *SHaring Interoperable Workflows for large-scale scientific simulations on Available DCIs*³¹ project, which continued as the ER-flow³² project.

SHIWA's goal is to achieve the interoperability of different workflows, belonging to different workflow systems and languages. It aims to increase the collaboration between communities using different solutions and the degree of reuse of scientific workflows.

Workflows can be published and shared over the SHIWA Simulation Platform, which offers services for supporting workflow interoperability. The execution of the workflows is handled by the SHIWA Portal, which allows enacting and executing them over the different European distributed computing facilities.

As explained in Chapter 6, we have selected experiments from Pegasus, dispel4py and Makeflow for the evaluation of this work.

2.3 Reproducibility in Computational Science

A computational experiment involves several elements, each of which must be conserved in order to ensure reproducibility. In this section we survey the main conservation

³¹<http://www.shiwa-workflow.eu/>, last accessed 2016

³²<http://www.erflow.eu/>, last accessed 2016

approaches for experiment components.

We expose how most of the work done so far is focused on the scientific procedure and the data components, where we can find several works facing the issues related to their reproducibility and proposing ways of conserving them.

We also expose several works that point out the necessity of providing an execution environment as part of the experiment description, and describe some approaches that consider the conservation and reproduction of some of the components of the computational equipment.

2.3.1 Scientific procedure conservation

The workflow paradigm has been widely adopted in different scientific areas, such as the bioinformatics community, for studying genome sequencing (Blankenberg et al., 2010; Giardine et al., 2005a), disease-related experiments (Fisher et al., 2009; Gaizauskas et al., 2004) and many others. Several studies have exposed the difficulties of trying to reproduce experimental results on life sciences, such as biology (Ioannidis et al., 2009) and cancer analysis (Errington et al., 2015).

Existing WMSs, such as Taverna (Oinn et al., 2006), Wings (Gil et al., 2011) and Pegasus (Deelman et al., 2005), Kepler (Ludäscher et al., 2006), GrADS (Mandal et al., 2004), Galaxy (Giardine et al., 2005b), etc., provide development and execution features that allow end users, usually scientists with no large skills on computing, to perform complex distributed tasks, facilitating the access to large distributed execution environments that involve a large amount of computational resources, Web Services, data sources, etc.

These systems cover all the range of scientific areas and even though they may be considered to be domain-agnostic, they are usually related to a specific area. This is due to the fact that sometimes they have been specifically designed and implemented to fit the necessities of a certain kind of scientific task, other times because a certain scientific community has adopted some of them for their specific purposes, based on their popularity or features.

The core component of every computational experiment is the scientific procedure that must be carried out to perform it; that is, the process of transforming the input data, representing a relevant fact or observations about real life, to obtain a set of results that allows concluding that a guess, in the form of an hypothesis, is true or not. This

is what makes the Scientific Method a powerful and verifiable tool and thus it is the resource we have to take more care of when conserving an experiment.

Several works have faced the problem of conserving and reproducing a scientific experiment procedure. As pointed out in (Belhajjame et al., 2011) "*a workflow specifications encapsulate knowledge that documents scientific experiments, and are, therefore, worth preserving*".

As the source code can be considered as the description and documentation of a process that solves a certain problem, a workflow specification can be seen as the description of a scientific procedure. Several languages have been developed for representing workflows, usually associated with a certain tool or WMS. Probably the most popular and widespread language for describing workflows is BPEL³³ due to its degree of standardization and the abundance of tools and execution engines. Even though some authors have argued about its feasibility to be used as an standard for describing scientific procedures (Barker and Van Hemert, 2008), BPEL is a business-oriented language, and the scientific community have tend to develop and adopt a more data-oriented and user-agnostic languages to represent their experiments.

As seen in section 2.2.2, we can distinguish two kind of languages based on the structure of the workflow that they allow to define: DAG and non-DAG . The first ones (e.g. SCUFL³⁴ in Taverna or DAX³⁵) are more restrictive, as they only allow to represent workflows by means of Direct Acyclic Graphs. That is, a set of nodes connected through edges that are able to define sequence, parallelism and choice structures, without iterations. Non-DAG languages (e.g. MoML³⁶ used in Kepler inherited from Ptolemy II (Eker et al., 2003a)) include iteration primitives as part of their specification, increasing their expressiveness but making them harder to schedule and enact, as they require the execution system to implement a persistent representation of the execution state (Gil et al., 2007).

The wide range of existent languages challenges the future reproduction of a workflow, as it may make the experiment platform-dependent. Most of the languages have been designed and implemented in an ad-hoc manner as part of the development of a WMS, complicating the migration of a workflow written in a certain language to another

³³<http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>, last accessed 2016

³⁴<http://www.taverna.org.uk/developers/taverna-1-7-x/architecture/scufl/>, last accessed 2016

³⁵<http://pegasus.isi.edu/wms/docs/schemas/dax-3.3/dax-3.3.pdf>, last accessed 2016

³⁶<http://ptolemy.eecs.berkeley.edu/publications/papers/00/moml/>, last accessed 2016

platform. Several solutions have been suggested for this problem; such as abstracting the definition of a workflow as much as possible, in such a way that it can be later translated to any specific language and avoiding the definition of new languages when implementing a new platform (Barker and Van Hemert, 2008).

Describing a scientific procedure by means of a workflow specification is a way of conserving it, but it is not enough. Once described, it is necessary to store and share it, in order to make it accessible in the future. Virtual scientific communities have been developed to achieve this goal, allowing scientists to expose their work to the community along with resources and support, bringing a more complete vision of the procedure and increasing the chances of reproducing it. myExperiment explores how collaboration and sharing increase the potential of workflows for handling scientific experiments, and provides design principles and dimensions for maximizing scientific software reuse (Roure and Goble, 2007).

myExperiment is a contribution of the myGrid³⁷ consortium, which has also explored the preservation of scientific experiments in the Wf4Ever project³⁸. This project aimed to preserve scientific workflows by defining a set of best practices in the scientific production process and models for describing scientific experiments as Research Objects (Belhajjame et al., 2012). These objects follow the “beyond the PDF”³⁹ principles and propose to publish scientific results by aggregating files and information into a bundle, that will contain not only the publication itself, but also experimental data, code, slides, etc.

Research Objects include a set of semantic models for describing the aggregated resources. These models allow describing relevant concepts including the workflow, data links, involved agents, files, process executions, etc. Another initiative following the same principles is BagIt⁴⁰, a hierarchical packaging format specification meant for transferring digital content for its preservation. It defines a *bag* as a set of files (the *payload*) and metadata describing them in the form of *tags*.

³⁷<http://www.mygrid.org.uk>, last accessed 2016

³⁸<http://www.wf4ever-project.org/>, last accessed 2016

³⁹<https://www.force11.org/meetings/beyond-pdf-2>, last accessed 2016

⁴⁰<https://tools.ietf.org/html/draft-kunze-bagit-11>

2.3.2 Data conservation

A scientific procedure aims to derive conclusions based on the results obtained from transforming input data. Both input data and results (and even the intermediate results obtained during the process) are valuable resources that must be considered as part of conservation and reproduction approaches. Input data is usually an expensive resource as it may require a lot of effort to collect, which may be even impossible to collect again as in the case of one-time phenomena. This data may be reused in different experiments along the time (Garijo et al., 2013; Kinnings et al., 2010) or shared among the community. Therefore it is mandatory to define and implement ways for conserving and accessing this information. Initiatives such as the Research Data Alliance (RDA)⁴¹ targets this challenge, promoting the sharing of scientific data among scientists from different disciplines. Since 2013, RDA aims to develop the means for improving the technical and social aspects of data sharing at a worldwide scale.

Experimental results are also a valuable data resource of an experiment, as they represent the major outcome of an experiment. These results validate the contribution of a scientific work and should be carefully documented and explained to convince the audience that they are consistent. Scientists must explain their input data and results in their publications, and should bring access to them in a way that they could be accessed in the future by themselves and the community.

To this end some authors have proposed to use lists for checking that all the necessary details of the experimental data are included as part of its description, allowing the community to understand and verify the exposed information. Inspired by the work done in the biological and biomedical areas on the MIBBI project (Taylor et al., 2008), and following the Linked Data (Bizer et al., 2009) principles, authors proposed in (Gamble et al., 2012) both a vocabulary and a framework that allows scientists to report their experimental data and checking its validity, improving the data reproducibility as the community can access more and better documented data sources.

Tracking the quality of the data involved in the experiment during the time is proposed as a way of avoiding data volatility and therefore workflow decay (Roure et al., 2011), measuring data integrity (completeness) and authenticity (lineage). Authors propose provenance solutions as a source for measuring these dimensions, based on

⁴¹<http://rd-alliance.org/>, last accessed 2016

the Open Provenance Model (Moreau et al., 2008) for measuring the integrity and authenticity of the workflow respecting to the data. Provenance has been also proposed to be used, by means of its various existent models^{42,43}, for facing workflow decay problems, allowing the total or partial reproduction of an experiment (Garijo and Gil, 2011). The combination of provenance and recomputation has been also explored for achieving data preservation in Scientific Workflow Middleware (Liu et al., 2006).

As mentioned before, results are a valuable part of the experimental data. In fact, the relevance of the method exposed in a scientific publication relies on the quality of the obtained results and on how they allow deriving the expected conclusions from the hypothesis. Usually results are just included and explained in the publication, sometimes by means of a link to an external resource containing the experiment resultant raw data. In (Gavish and Donoho, 2011) authors expose how trusting the results published in this way is often hard, as they may be poorly explained, and understanding the computations involved on them is usually difficult. To solve this problem they propose the Verifiable Computational Research (VCR) discipline, in which experimental results are published as Verifiable Computational Results, including metadata explaining how they were obtained. These results would be identified through a unique Verifiable Result Identifier and stored in a Verifiable Result Repository.

However, conserving data is not enough. Without the ability of properly consuming the conserved information we are left with files full of rotting bits (Cerf, 2011). Therefore it is mandatory to maintain the operability of the tools for accessing, consuming and interpreting the components of a scientific experiment (either input data or the description of the method).

2.3.3 Equipment conservation

Even the most complete and comprehensive description of a computational experiment, including the consumed and generated data and the experimental procedure, may not be enough for guaranteeing its future reproduction if it does not contain a description of the equipment that supported its execution.

In traditional scientific disciplines (either in-vivo or in-vitro) the equipment involved in an experiment is usually a well-known set of resources, which researchers implicitly

⁴²<http://www.w3.org/TR/prov-primer/>, last accessed 2016

⁴³<http://www.opmw.org/>, last accessed 2016

assume to be available in their own laboratories (e.g. microscopes, petri dishes, etc.). Otherwise it is fully described as part of the documentation of the experiment within the laboratory notebook and/or part of the material and tools section of a publication. Apparatus such as laboratory reagents in biological experiments must be described with a high level of detail, from its type and composition to its specific brand and manufacturer, so as to guarantee correct results. High technology resources such as telescopes are described explaining their hardware characteristics and also the specific configuration used during the experimental process.

Computational Science should not be an exception to this way of publishing scientific contributions. As the equipment used in a computational experiment is composed by computational resources such as computers, clusters, Web Services, software components, etc., authors should be able to document sufficiently these infrastructures within the context of their experiments.

In 2015, a comprehensive study was published (Collberg et al., 2014), where 601 papers from ACM conferences were surveyed, studying how authors shared the data and code supporting their results. 402 of those papers were supported by code, which was obtained by looking for links within the paper itself, searching on code repositories, or contacting the authors when necessary. After the code was obtained, several students were asked to build it. This whole process was limited by experimental design to a period of 30 minutes. Results showed that in 32.3% of the 402 papers students were able to obtain and build the code within the given period. In 48.3% of the cases, code was built with some considerable effort, and in 54% of the papers code was either built or the authors stated the code would build with reasonable effort. Authors proposed, as a result of this study, a *sharing specification* for publications that allow to state the level of sharing of each paper.

However there is not so much work done on this area, even though several works have pointed out the impact of execution environment issues on computational scientific experiments reproducibility. In (Garijo et al., 2013) authors explain the problems they faced when they tried to reproduce an experiment (Kinnings et al., 2010) about mapping all putative FDA and European drugs to protein receptors within the scope of a given proteome. For each one of the problems authors propose a set of advices explaining how the related issues must be solved. In four out of the total six advises execution

environment problems are mentioned, where data issues are mentioned in only two, and the same for methodological issues.

Another interesting study about issues in computational scientific reproducibility is exposed in (Zhao et al., 2012), where authors conducted a study about workflow decay over a set of biological workflows from myExperiment (Roure et al., 2007) designed for the Taverna (Oinn et al., 2006) platform. Authors define four different categories for workflow decay causes: (i) *Volatile third-party resources*, *Missing example data*, *Missing execution environment*, and *Insufficient descriptions about workflows*. The study shows that nearly 80% of the workflows failed to be reproduced, with around 12% of these failures due to *Missing execution environment* issues and 50% due to *Volatile third-party resources*. Taking into account that around 22% of the tasks in Taverna are related to Web Services (Wassink et al., 2009), some of those third-party resources issues could be considered also as execution environment problems.

Some approaches that deal with the preservation of some of the components of an infrastructure have been proposed, but so far a complete and integral approach for its preservation as a whole has not yet been developed.

In (Matthews et al., 2009) authors expose how software must be preserved. As a complex and dynamic entity, software cannot be preserved just by maintaining its binary executable code. Authors claim that a software component is more likely to be preserved by guaranteeing the performance of its features rather than conserving the same *physical* binary code. To this end they introduce the concept of adequacy, as a way of measuring how a software component performs in relation to a certain set of features. The aim is to build a conceptual model that allows capturing the relevant properties of each software, enhancing the possibilities of successfully conserving them.

In 2011, the Executable Paper Grand Challenge (Elsevier, 2011) pointed out the importance of allowing the scientific community to reexamine the execution of an experiment. As a result of this challenge, some authors proposed the use of virtual machines as a way of preserving the execution environment of an experiment (Brammer et al., 2011; Gorp and Mazanek, 2011). Also as part of the SIGMOD conference on 2011 a study was carried out to evaluate how a set of repeatability guidelines proposed to the authors submitting a paper (i.e. using virtual machines, pre- and post- conditions, and provenance-based workflow infrastructures) may help reviewers to reproduce the experiments described on the submitted paper (Bonnet et al., 2011).

Replicability and reproducibility of computational experiments using cloud computing resources and software descriptions have been proposed as an approach for those studies where performance is not a key experimental result (Crick et al., 2014).

One of the earliest approaches in which virtual machine appliances were suggested as a way of solving reproducibility issues in science was CernVM (Buncic et al., 2010). At CERN, researchers proposed to use virtual machines images to share complete and portable environments for executing computational analysis for the LHC experiments. The use of virtualization techniques allows separating the target computing resource from the underlying infrastructure, enabling also a better resource utilization and supporting legacy systems.

A list of advantages and challenges of using virtual machines for achieving reproducibility is exposed in (Howe, 2012), arguing that availability of a highly distributed and automated solution for computing such as Cloud Computing allows cost reduction, efficient and reliable lifecycle management, large scale processing and cost sharing. However authors expose that using Cloud solutions implies issues that are not yet fully solved, such as the high cost of storing data in the Cloud or the problems of dealing with high interactive experiments through a network connection to remote virtual machines.

Authors also claim that provenance tracking inside a virtual machine or the reuse and repurpose of the infrastructure are real issues when using Cloud solutions. In our opinion these claims are not major issues within the scope of our work. Provenance of a process executed on a virtual machine can be traced by using a Workflow Management System in the same way it can be traced on a local cluster. Regarding to repurposing an infrastructure, this is out of the scope of our work, as we are trying to achieve its conservation and not looking for any kind of improvement.

One of the most challenging issues for achieving workflow reproducibility is how to preserve Web Services, as they are key components of the execution environment of the experiment. Usually these Web Services are not running locally but rather running on a third-party provider, making hard to ensure its future availability. Also having access to their implementation is not always possible and therefore replicating them in another infrastructure almost impossible. Some authors (Belhajjame et al., 2011) have proposed to deal with this problem by annotating the capabilities of the Web Services employed in a scientific workflow in order to find substitute counterparts offering the same features which can be used in the future, making possible to re-execute the experiment.

In this context, some authors⁴⁴ have clearly exposed the necessity of capturing and preserving the execution environment of an experiment, providing tools for analyzing and packaging the resources involved on it. ReproZip (Chirigati et al., 2013) is a relevant tool in this direction, which is aligned with some of the principles of our work, as it aims to capture the knowledge about an infrastructure and tries to reproduce this in a new environment. This tool reads the infrastructure components involved on the execution (files, environment variables, etc.) by using SystemTap⁴⁵ and stores this information in a MongoDB⁴⁶ database. Then the described elements are collected and packaged. This package can be later unpackaged into another machine in order to repeat the experiment. These kind of approaches, which focus on packaging the physical components of a given infrastructure, limit their scope of applicability, as the packages require most of the target machine to be the same.

In the same way, the SOLE framework also targets the reproducibility of computational environments by means of its proper documentation (Pham, 2014). It provides methodological and technological support for improving and testing the reproducibility of paper-based scientific results, covering its data, environment and even the motivation that led to the study. As part of its framework, SOLE includes PTU (Provenance-To-Use), a tool for reproducing the environment of an experiment. PTU uses the *Code, Data, and Environment* (CDE) tool⁴⁷ as the enactment tool for creating and running packages, which includes the data, environment variables, and code. PTU audits the execution of an experiment by using ptrace⁴⁸, in a similar way to ReproZip.

Some efforts have been undertaken to define syntaxes for describing the execution environment of scientific applications. The recently released TOSCA⁴⁹ standard has been proposed to specify the components and life cycle of scientific workflows, including their dependencies. TOSCA is a Cloud management standard for the creation of declarative templates to define the workflow as well as the information required for enacting its environment. As a result, workflows are portable to any TOSCA-compliant

⁴⁴<http://icerm.brown.edu/tw12-5-rcem/>, last accessed 2016

⁴⁵<http://sourceware.org/systemtap/>, last accessed 2016

⁴⁶<http://www.mongodb.org/>, last accessed 2016

⁴⁷<http://www.pgbovine.net/cde.html>, last accessed 2016

⁴⁸<http://man7.org/linux/man-pages/man2/ptrace.2.html>, last accessed 2016

⁴⁹<http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html>

provider (Qasha et al., 2015). Although this approach significantly increases the reproducibility degree of the experiment, it requires the workflow to be rewritten.

Similarly, several scientific entities have proposed the use of Docker containers⁵⁰, lightweight wrappers for packaging pieces of software, as a way of sharing the computational environment of an experiment relying on virtualization. Docker Compose⁵¹ allows defining the components of a computational resource in a text file, that can be enacted as a container, deploying these components on the fly.

Another relevant contribution to the state of the art of workflow preservation involving the description of the workflow environment in a formal way was developed within the context of TIMBUS (Mayer et al., 2014). The project aims to preserve and ensure the availability of business processes and their computational infrastructure, aligned with the enterprise risk and the business continuity management. They also proposed a semantic approach for describing the execution environment of a process. However, even though TIMBUS has studied the applicability of their approach to the eScience domain, their approach is mainly focused on business processes. TIMBUS proposes a set of ontologies⁵² to handle the preservation and re-execution of business processes. This modular network is composed by one core ontology and seven ontology extensions, and follows the high-cohesion and low-coupling principles. Its core ontology is based on ArchiMate (Lankhorst et al., 2010), a domain-independent model for describing enterprise architectures. Linked to this ontology authors provide models for describing the software and hardware components, legal assets related to the business process, format information, and provenance.

Other semantic approaches have been also proposed focusing on specific domains on science, such as biomedical research, as a way of achieving reproducibility of published experiments (Malone et al., 2014). In this work authors proposed to annotate the software artifacts in the same way that gene products or phenotypes are annotated. In order to do so, authors developed the *Software Ontology* (SWO), a model for describing the software involved on the storage and management of data related to biomedical activities.

⁵⁰<https://www.docker.com/what-docker>, last accessed 2016

⁵¹<https://docs.docker.com/compose/>, last accessed 2016

⁵²<http://timbusproject.net/portal/publications/ontologies>, last accessed 2016

2.4 Summary

In this chapter we have introduced some of the most relevant contributions in the context of scientific reproducibility, establishing the starting context of the work developed in thesis. As seen in the previous sections, several initiatives have targeted the different dimensions of conservation and reproducibility of scientific experiments. Whereas most of them focus on the scientific procedure and data aspects of the problem, a lack of approaches for the conservation of scientific equipment has been identified.

Among the solutions that do consider the conservation of the execution environment for computational experiments, we identify some limitations that should be covered or improved in order to achieve a solution. In many cases, these initiatives propose packaging the resources involved in the execution of an experiment into a bundle that can be shared and deployed as it is. While this kind of approach allows transferring the environment and replicating it, some issues arise from them. On the one hand, these bundles, which usually consist of a virtual machine image containing all the experimental materials, are large in terms of size, causing a high demand on storage for them, what may be a challenging problem (Mao et al., 2014; Zhao et al., 2014). On the other hand, providing just these aggregated materials, without providing any further information about their contents, harnesses their reproducibility. Virtual machine images behave as black boxes that do not provide any insight about the relevant components of the environment.

A more adaptive and expressive solution is required for improving the reproducibility degree of computational environments. We claim that in order to do so, it is necessary to provide access to the required materials, but also information describing them, defining how they behave and which is their context. This information would allow not only understanding the provided materials, but also identifying potential counterparts for them if necessary.

In the following chapters of this thesis we will describe our proposed solution for this issue, by implementing a reference framework for the description of the materials related to the execution environment of computational experiments. This way we are exposing the knowledge behind these infrastructures, which enables the consumer of this information, either a user or a system, to understand and manage it. Based on this, a more adaptive reproducibility process can be built: having enough insights on

how the environment is composed allows adapting it to the evolving conditions that occur when reproducing an experiment.

This adaptive capacity also eases the high storage problem introduced before, as rather than sharing the whole set of experimental materials, it is possible to share a description containing pointers to them and instructions on how to deploy and use them. This way, new environments can be obtained on the fly, being a reproduction of the original one according to the provided description. A common pool of shared resources can be defined, within the context of a scientific community or domain, which by their own cultural behavior, tend to use the same components and resources, just arranged and configured differently to fit the necessities of each experiment.

Chapter 3

Work Objectives

“First, we guess it. Then we compute the consequences of the guess, to see what if this law we guess is right. If it disagrees with experiment, it’s wrong. In that simple statement is the key to science.”

Richard Feynman on the Scientific Method, 1964

The main focus of this work is scientific reproducibility in the context of Computational Science, complementing existing approaches for reproducibility in scientific processes and data with a new approach that takes care of reproducing the execution environment of a scientific computational experiment.

We have identified several open research problems, which led us to define the goals of this work. For achieving these goals we have defined a set of hypothesis that formalize them. We then define a set of facts that we assume hold in the context of this work and also restrict its scope of applicability.

3.1 Open Research Problems

The challenges identified in the context of this work can be decomposed into three main research problems, which depict the reproducibility problem through the different phases of an experiment lifecycle.

- Open Research Problem 1: Computational Infrastructures are usually a predefined element of a Computational Scientific Workflow. The majority of computational scientists develop their experiments with an already existing infrastructure in mind, thus not considering its definition as part of the experiment.
- Open Research Problem 2: Execution Environments are poorly described, or even not described at all, when describing the results of an experiment. Often, the infrastructure used in the evaluation process is summarized explaining briefly its hardware overall capabilities and the basic software stack. This lack of information compromises the conservation and reproducibility of the experiment.
- Open Research Problem 3: Current approaches for Computational Scientific Experiments conservation and reproducibility take into account only the computational process of the experiment (scientific procedure) and the data used and produced, but not the execution environment.

3.2 Hypothesis

From the research problems stated above, we have defined the following as the main hypothesis of this work.

“It is possible to describe the main properties of the Execution Environment of a Computational Scientific Experiment and, based on this description, derive a reproduction process for generating an equivalent environment using virtualization techniques.”

This hypothesis can be decomposed into three different sub-hypotheses, listed below.

- Hypothesis 1: Semantic technologies are expressive enough to describe the Execution Environment of a Computational Scientific Experiment.
- Hypothesis 2: An algorithmic process can be developed that, based on the description of the main capabilities of an Execution Environment, is able to define an equivalent infrastructure for executing the original Computational Scientific Experiment obtaining equivalent results.

- Hypothesis 3: Virtualization techniques are capable of supporting the reproduction of an Execution Environment by creating and customizing computational resources, such as Virtual Machines, that fulfill the requirements of the former experiment.

3.3 Goals

To address the challenges exposed above the following goals have been defined. Each of them aim to face one or more of those problems. These goals will allow us to verify the stated hypotheses, and will guide the design and development of the contributions of this work.

- Goal 1: Create a model able to conceptualize the set of relevant capabilities that describe a Computational Infrastructure.
- Goal 2: Design a framework to provide means for populating these models, collecting information from the materials of a Computation Scientific Experiment and generating structured information.
- Goal 3: Propose an algorithm that, based on the description of a former Computational Infrastructure, is able to define an equivalent infrastructure specification.
- Goal 4: Integrate a system able to deploy virtual machines on several Virtualized Infrastructure providers, meeting a certain hardware specification and install and configure the proper software stack, based on the deployment plan specified by the afore-mentioned algorithms.

Each of the sub-hypotheses exposed before can be related to one or more of the aforementioned goals of this work, as listed in Table 3.1. By achieving each of these goals, we can consider that the corresponding hypothesis hold.

3.4 Restrictions

We have defined a set of restrictions that allow focusing our work into a concrete scope, enabling us to state which are the cases in which our solution is applicable.

Hypothesis	Goal
H1	G1,G2
H2	G3
H3	G4

Table 3.1: Correspondences among Goals and Hypothesis

- Restriction 1: As we deal with external providers and solutions, each one of them defining its own *Quality of Service*, we cannot guarantee a customized *QoS*.
- Restriction 2: Guaranteeing the execution performance of a reproduced computational experiment is out of the scope of this work. That is, we can guarantee that every reproduction of the experiment will obtain equivalent results, but the processing time may vary from one execution to another.
- Restriction 3: Considering the whole spectrum of applications that may be used in a Computational Scientific Experiment is not feasible. Therefore in the context of this work, we consider the subset containing the most common ones for the Computational Scientific Experiments belonging to the areas of science we are studying.
- Restriction 4: External infrastructure dependencies, such as Web Services, that are not under control of the designer of the experiment, are out of the scope of this work, as we cannot guarantee their availability.
- Restriction 5: As we are not considering data-related aspects, we will study only the Computational Scientific Experiment execution environments that can be studied and reproduced independently from the experimental data.

3.5 Assumptions

As well as restricting the scope of our work, we have to clearly state which are the facts that we take for granted from the world. These assumptions support some of our claims in this work.

- Assumption 1: There are several available solutions and providers for virtualizing computing infrastructures, each of them exposing its services through their own APIs and services, and using their own nomenclature.
- Assumption 2: The sets of applications and tools used in computational scientific workflows belonging to a certain scientific area are often the same and are usually reused among those workflows.
- Assumption 3: In general, reproducibility is more important than performance in a scientific experiment. That is, it is more important to obtain equivalent results than improving the time to obtain them.
- Assumption 4: Computational Scientific Workflows are a widely accepted approach for representing Computational Scientific Experiments.
- Assumption 5: Virtualization techniques are a mature and stable technology. Therefore virtualized infrastructure providers will be available in the long term, either in the form of public Infrastructure as a Service providers or as software solutions that will support the management of virtualized resources on a private infrastructure.
- Assumption 6: Two Execution Environments are considered to be equivalent if they are able to execute the same Computational Scientific Experiment, obtaining equivalent results.
- Assumption 7: Two Computational Scientific Experiment results obtained from the same experiment are considered to be equivalent if they allow deriving the same conclusions from the hypothesis of the experiment.
- Assumption 8: The Execution Environment of a Computation Scientific Experiment is defined by the set of software and hardware components that are involved on the experiment's execution.

3.6 Research methodology

This work has followed an standard research methodology to identify and resolve the research problems exposed on it. In the very beginning we explored how scientific

experiments were designed, executed and repeated in *in-silico* science. Based on our observations we concluded that there was a lack of efforts on the conservation and reproducibility of the execution environments of scientific workflows. This idea was later confirmed by the evidences on the surveyed literature of the area. We then envisioned how those issues could be faced and started designing a plausible solution for them.

The procedure we have followed falls into the classical definition of a research methodology, which can be defined as “*a systematic way of solving a research problem*” (Goddard and Melville, 2004). As introduced in chapter 1, this work focuses on the reproducibility process associated to a computational experiment, specifically on scientific workflows, as one of the most common ways for designing them. This topic was selected due to the previous work developed by its author and the increasing momentum of this topic in the scientific community. Reproducibility has proved to be a crucial aspect in computational science, whith several efforts being developed during the period of this work.

Once this topic was selected, we conducted an extensive survey of the scientific literature on the topic, as explained in chapter 2. This study showed that most of the available approaches on reproducibility were only considering the aspects related to the scientific data and its analysis process. Despite the importance of using computational solutions for solving scientific processes, the computational infrastructures were not being taken into account as part of the reproducibility cycle. We considered that the lack of approaches in that sense was a weak point for reproducibility, and thus decided to face this problem.

Based on this study we defined a set of goals that in our view must be achieved in order to solve the problem. We then went through an iterative process for defining a hypothesis that supported those goals. We refined it as well as restricted its applicability by defining the related assumptions and restrictions. For the sake of clarity and testability, we decomposed this main hypothesis into three different ones. By doing this we could state more clearly how each of the goals of our work relate to each part of the main hypothesis.

To verify that our guesses were valid or not, we envisioned a research process for developing the necessary contributions to test them. We defined three main contributions for our work. First, we needed a way of representing and storing the information related to the execution environment of a scientific workflow. As described in chapter 4, we

decided to implement a semantic model, as semantic technologies represent a standard and consolidated way for information representation. For developing it we followed the NeOn methodology, a well known methodology for ontology design and development.

The second contribution of our work is the set of tools for populating the aforementioned models. On the one hand, expecting a purely manual annotation from scientists of all the information describing an execution environment is not feasible. On the other hand, it is highly complicated to develop a fully automated process that is able to harvest and define the main capabilities of a system. In this work we have implemented a set of tools for assisting scientists on the process. Different scraping tools have been developed for parsing the different files that define a workflow on the different systems we have worked with, namely Pegasus, Makeflow and dispel4py. With this information, a first draft of the dataset is generated. This draft can be completed and corrected by the user.

Finally, to demonstrate that it is possible to produce an equivalent environment based on that information, we have designed and implemented an algorithm that produces different infrastructure specifications. As explained in section 5.2.1, we chose a decoupled design, in which our algorithm consumes the input data about a workflow and generates a platform-independent specification format. This specification is then concretized into the different enactment solutions we have included in this work and that are explained in section 5.3.

The algorithm introduced in this work is a concrete implementation among many others that can be developed. We argue that starting from a structured and well-defined description of the execution environment, different algorithm can be implemented, exploring different approaches for resource scheduling or data transfer optimization among many others.

To show how the exposed contributions were able to solve the stated problems, we selected several examples of scientific workflows from different scientific domains and workflow systems. This variety of workflows allows showing that the designed solution is general enough to cover a wide range of experiments. We have also explored different virtualization solutions for deploying computational resources, as well as several enactment solutions. In chapter 6 we introduce which are those components and describe, for each workflow, why it was selected and detail its reproducibility process.

This experimentation allowed us to evaluate the feasibility of our approach. For each workflow we study the outcome of the process and compare it with the expected results. In chapter 7 we state the conclusions of our work, explaining how it applies to current approaches and how it could be applied to new systems. We also discuss which are the next steps to be carried out, based on the evidences of our work.

Chapter 4

Execution Environment Representation

“And here we need a warning that to go further, you are going to bump into the O word for “ontology”, the S word for “schema”, and “controlled vocabularies”. That is to say, in going down this path, you’re going to start talking about semantics”

Jim Gray, The 4th Paradigm 2009

According to the ideas exposed in previous chapters, in this work we envision a novel solution that approaches scientific reproducibility of execution environments. As we have identified before, there is a need of exposing the underlying knowledge about the resources required for a computational environment, rather than just sharing them physically. This knowledge must be codified under a structured and documented manner, in order to make it understandable in such a way that its different parts can be related to their context.

To this end, we describe in this chapter a reference framework for representing the computational resources involved in the execution of a scientific workflow. This framework must contain the concepts needed to represent the different elements that compose a computational infrastructure and the properties that relate them to each other, as well as to the specific information codifying their behavior.

Among the different options that can be selected for representing this information, such as tag-based systems or markup languages, we argue that semantic technologies are the most suitable solution for the purpose of this work. Semantics and their associated technologies have evolved during last decades, involving several communities and joint efforts to define ways for information representation. Mainly led by the World Wide Web Consortium (W3C)⁵³, several protocols and languages, such as OWL2⁵⁴, RDF⁵⁵, and SPARQL⁵⁶, have been proposed and maintained by the community, along with a wide range of development and management tools, that make them a mature and suitable solution in the context of this work.

4.1 An Ontology Network for the Representation of Execution Environments

In this section we first describe the methodological process followed to develop the semantic models that support this work, which is then presented in Section 4.1.2. These design and development processes are based on the NeOn methodology (Suárez-Figueroa et al., 2012, 2015), a scenario-based methodology for building ontology networks. We followed the main steps of this methodology to drive the process. Some of these steps are abstract, meaning that they do not define a concrete process for fulfilling the activities. On those cases we have relied on other well-known guidelines (Noy and McGuinness, 2001).

4.1.1 Our Ontology Network Development Process

The NeOn methodology provides nine different scenarios for building ontologies, each of them meant to describe the different ways in which an ontology can be developed, depending on the starting point and resources that the ontology engineer wants to use. As our work started as an exploratory process, aiming to test the feasibility of using semantics for conserving computational environments, we have selected the first scenario of the methodology: *“From specification to implementation”*.

⁵³<http://www.w3.org/>, last accessed 2016

⁵⁴<http://www.w3.org/TR/owl2-overview/>, last accessed 2016

⁵⁵<http://www.w3.org/RDF/>, last accessed 2016

⁵⁶<http://www.w3.org/TR/rdf-sparql-query/>, last accessed 2016

This scenario fits in the context of our work as it is designed for developing ontologies from scratch. It assumes that no other knowledge resources are going to be reused for the new model. In this work we intend to develop a network that is self-contained and that can be further extended and aligned with other resources.

The following sections explain the different stages of the development process by detailing each of the activities of the NeOn methodology we performed. Through this process we motivate the decisions we took when building the **Workflow Infrastructure Conservation Using Semantics (WICUS)** ontology network described in Section 4.1.2.

4.1.1.1 Ontology Requirement Specification

According to the selected scenario of the methodology, the very first step of the process is to identify the requirements for the target ontology. In this activity we determined the needs of our work and generated the Ontology Requirements Specification Document (ORS), which allowed us to state which knowledge resources must be included in our models.

The first step for developing it was to identify the purpose of the ontology. In our case, the main purpose is to define a consensual model of the execution environment of computational experiments designed as scientific workflows. Secondly, we stated the scope of our models, being it the different workflow management systems and their related environments for developing workflows as well as the specific scientific software tools that are used by them.

For the third step we defined the implementation language required for the ontology. Whereas we did not have any imposition regarding which language to use, we selected OWL ⁵⁷, as the de facto standard and most used language for encoding ontologies.

The fourth step consists on identifying the final end-users of the ontology, which in our case are usually domain scientists, either the ones developing an experiment or the ones accessing an already available experiment with the aim of reproducing it. The intended use of the ontology, as required in step five, can be also defined in the same terms; it can be used for producing information about the environment of an experiment as part of its design and publication process as well as for consuming this information to understand how this environment works and reproduce it.

⁵⁷<http://www.w3.org/TR/owl2-primer/>, last accessed 2016

The sixth step defines the core of the ORSD, describing the non-functional and functional requirements for the future ontology. The only non-functional requirement of our specification is defined by the naming convention we used for classes and properties. All the names of the elements in our ontology follow a CamelCase convention (Manaf et al., 2010), where class names are capitalized and property names are lower case. This naming convention improves the readability of the model and is the de-facto standard on the semantic community.

For the functional requirements we followed the recommendations of the methodology and defined them by means of a set of competency questions. This technique has been also proposed in other methodologies for ontology development, such as METHONTOLOGY (López et al., 1999) or On-To-Knowledge (Sure et al., 2003), and represents a set of questions that the target ontology must be able to answer.

For this step we followed a bottom-up strategy, starting from simple and general questions that evolved into more concrete and complex ones. The list containing these questions is displayed within the ORSD in appendix A.

Finally we extracted a pre-glossary of terms from both the competency questions and their answers. This allowed us to have a first set of words that could represent the attributes and relations, usually extracted from the competency questions, and the concepts, mainly extracted from the answers, for our models.

4.1.1.2 Ontology Conceptualization

For this activity we relied on the *Ontology Development 101* guideline (Noy and McGuinness, 2001). This guide proposes a set of steps for building the class hierarchy, their corresponding properties and instances.

We followed a top-down development process, starting by identifying the more general concepts of our domain. Some of these concepts were directly extracted from the pre-glossary of terms generated as part of the ORSD. For example, we started by defining the concept *Workflow*, which we then specialized into two subconcepts; *Abstract Workflow* and *Concrete Workflow*.

Then, based on the list of terms extracted from the competency questions on the ORSD, we defined the properties of our ontology. We defined their domain and ranges according to the competency questions, as well as their cardinality. We also followed a

top-down approach, starting from general properties such as *version* to more concrete ones, such as *lower version* or *higher version*.

4.1.1.3 Ontology Modularization

Modular design, as well as in other fields in computer science, is a best-practice when developing ontologies. For improving the maintainability and extensibility of our models we identified the main sub-domains of our ontology. As introduced later on this chapter, we identified four different and interrelated domains in which our model can be split:

- Software: describes any kind of software component meant to solve a computational experiment.
- Hardware: describes the set of hardware specifications of a computational environment.
- Computational resources: describes the processing units for executing a computational experiment.
- Workflow requirements: describes the different requirements associated to the different sections of a scientific workflow.

We also identified that these domains are not isolated but related among them complementing each other. For example, we need to use the concepts from the hardware domain to explain the characteristics of some of the elements of the computational resources domain. We may also need to describe the requirements of a workflow using concepts from the software domain. Therefore we created several inter-domain relations, included in a top-level ontology, generating an ontology network.

4.1.1.4 Ontology Implementation

As stated before the ontology has been developed according to the OWL 2 syntax and serialized using the RDF/XML specification. For implementing it we used the Protégé⁵⁸ ontology editor.

Following the naming convention stated in the ORSD we created all the classes using CamelCase capitalized names. Classes such as *Abstract Workflow* were created

⁵⁸<http://protege.stanford.edu>, last accessed 2016

Domain	Prefix	URI
Computational resources	wsva	http://purl.org/net/wicus-sva#
Hardware	whw	http://purl.org/net/wicus-hwspecs#
Software	wstack	http://purl.org/net/wicus-stack#
Workflow requirements	wreqs	http://purl.org/net/wicus-reqs#
Top-level ontology	wicus	http://purl.org/net/wicus#

Table 4.1: Prefix and URI of the ontology network

WICUS Concept	External Concept	Equivalence Type
wicus:requiresHardware	dterms:requires	rdfs:subPropertyOf
wreqs:requiresExecutionEnvironment	dterms:requires	rdfs:subPropertyOf
wreqs:Workflow	p-plan:Plan	rdfs:subClassOf
wreqs:WorkflowStep	p-plan:Step	rdfs:subClassOf
wreqs:isStepOfWorkflow	p-plan:isStepOfPlan	rdfs:subPropertyOf
wstack:ConfigurationParameter	p-plan:Variable	rdfs:subClassOf
wstack:DeploymentPlan	p-plan:Plan	rdfs:subClassOf
wstack:DeploymentStep	p-plan:Step	rdfs:subClassOf
wstack:isConfigurationParameterOf	p-plan:isInputVarOf	rdfs:subPropertyOf
wstack:isDeploymentStepOf	p-plan:isStepOfPlanop	rdfs:subPropertyOf

Table 4.2: Concept equivalences

as *AbstractWorkflow*. Properties such as *version* were created as *hasVersion*, adding a verb when necessary and keeping the first lower case.

We implemented our ontology network by generating five different OWL 2 files, one for each of the domain ontologies and one for the properties linking them. Defined different URI namespaces for identifying them. Table 4.1 depicts the URI convention for each of the ontologies as well as their corresponding prefix.

4.1.1.5 Ontology Alignment

During the conceptualization process we identified some similarities with other concepts from well-known vocabularies. Even when this step is not strictly necessary for the purpose of our work, we included several mappings establishing equivalences between concepts. The list containing these equivalences is shown in Table 4.2.

4.1.1.6 Ontology Annotation

As a best practice, we have added several annotations to our ontologies. These annotations provide extra information about an ontology, describing who created it, when it was created and modified, its current version, etc.

This information is essential in cases such as our work, where we have introduced new versions of our models, easing the control tracking of them.

4.1.1.7 Ontology Documentation

We have introduced before the importance of having an explicit model for guiding the description of computational environment and achieving its later reproducibility. Therefore we considered crucial to provide a public documentation describing the concepts and relations of our models.

To do so, we used the Live OWL Documentation Environment (LODE)⁵⁹ tool. This online tool generates an HTML file for OWL files. We generated the five corresponding files for our ontologies and adapted them to meet our requirements.

4.1.1.8 Ontology Publication

To make models and documentations publicly available we decided to publish them online. In order to get a consistent experience for the users and facilitate its understanding we followed best practices for publishing ontologies. We selected persistent URLs as the URIs of our models, and made them dereferenceable.

This way even if we change the physical location of our files, the URIs will still point to the ontology resources. We also added content negotiation, so if a user aims to type the URI of a resource on a browser, he will be redirected to its online HTML documentation by default. If an agent accesses the URI using a loader, it will get the machine-readable version of the ontology, that is, the OWL raw file.

4.1.1.9 Ontology Validation

To validate that our ontology correctly represents the domain we have tried to model in this work, we have to compare it with a real world scenario. The best, and arguably

⁵⁹<http://www.essepuntato.it/lode>, last accessed 2016

the only way of assessing the quality of an ontology, is by using it on the application for which it was designed (Noy and McGuinness, 2001).

As explained in section 6.2.1.1, we test the validity of our models using a real case scenario, a scientific workflow on the astronomy domain. We documented its execution environment and using the algorithm introduced in 5.2.1 we verify that our ontologies were almost able to fully capture the information of a computational environment.

4.1.1.10 Ontology Diagnosis & Repair

We detected that some aspects of computational environments were missing on the models and that others should be further specialized in order to describe them completely. Based on the results obtained from the above stated validation process we identified the concepts to add and define further.

We added and modified the corresponding concepts on the different ontologies.

4.1.1.11 Ontology Upgrade & Versioning

In order to ensure compatibility and maintain a structured production cycle through when adding changes to the ontologies we upgraded them using a version convention. Every time we uploaded a new release of the ontology network we saved the previous release and placed a new version of the documentation as well. Also, the previous and current releases are stated in using the annotation properties of each ontology.

In an exploratory work as ours, this validation, diagnose and repair process is expected to occur several times, as new experiments are conducted and new potential improvements are identified, thus it is important to keep a consistent version control.

4.1.2 The WICUS Ontology Network

As a result of the development process listed before, we have generated the WICUS ontology network (Santana-Pérez and Pérez-Hernández, 2015) (available online at <http://purl.org/net/wicus>). This ontology network describes the main concepts of a computational infrastructure in the context of scientific workflow execution.

The network is composed by four domain ontologies, namely, the *Software Stack Ontology* (4.1.2.2), the *Hardware Specs Ontology* (4.1.2.3), the *Scientific Virtual Appliance Ontology* (4.1.2.4), and the *Workflow Execution Requirements Ontology* (4.1.2.1).

Figure 4.1: WICUS ontology network overview.

As mentioned before, we link these ontologies by means of the *WICUS Ontology*, a top-level ontology that defines object properties relating concepts from the domain ontologies, as depicted in Figure 4.1. In this section we introduce the main concepts and properties of the ontologies that compose the network.

4.1.2.1 Workflow Execution Requirements Ontology

We have implemented this ontology for describing and relating the different components of a scientific workflow with their related execution requirements.

This ontology defines the concept of `wreq:Workflow`, which represents a Scientific Workflow, and links it to the `wreq:ExecutionEnvironment` concept. This class defines the hardware and software dependencies that must be available for executing the workflow. An execution environment dependency can be either a software or a hardware specification, thus this ontology defines the sub-classes `wreq:HardwareRequirement` and `wreq:SoftwareRequirement`. A `wreq:Workflow` is linked to its corresponding requirements by the `wreqs:requiresExecutionEnvironment` property.

Depending on whether the steps of the workflow are fully specified in terms of infrastructure or not, we distinguish between `wreq:ConcreteWorkflow` and `wreq:AbstractWorkflow`. An abstract workflow is defined as a workflow that can be enacted using different infrastructure components, whereas in a concrete one all its execution requirements are fully stated.

For the purpose of relating the components of an execution infrastructure to the different parts of a workflow, we consider that a workflow can be composed of several subworkflows, defining each one of them their own requirements description. A workflow and its subworkflows are linked through the object property `wreqs:hasSubworkflow`.

Figure 4.2: Workflow Execution Requirements Ontology overview.

We do not describe the execution order of these workflows or their inner steps, as in our approach this information is not relevant from the infrastructure point of view.

An overview of this ontology is depicted in Figure 4.2.

4.1.2.2 Software Stack Ontology

This ontology describes the software elements related to a scientific workflow. These elements can be used to describe both, the components already deployed software in a computational resource and the software requirements of a workflow, depending on whether they are being used for describing workflow requirements or a virtual appliance.

The central concept of this ontology is the `wstack:SoftwareStack` class, which groups a set of `wstack:SoftwareComponent` by the `wstack:hasSoftwareComponent` property. A `wstack:SoftwareComponent` encodes the information of a software package, a library, a script, or in general, any piece of executable software. Two set of components can be defined as dependent by linking two `wstack:SoftwareStack` using the `wstack:dependsOn` property.

To encode the information about how to deploy and configure a software element, this ontology also includes the `wstack:DeploymentPlan`, `wstack:DeploymentStep`, `wstack:`

Figure 4.3: Software Stack Ontology overview.

ConfigurationInfo, and wstack:ConfigurationParameter classes. A plan defined by a wstack:DeploymentPlan is composed by several steps, by means of the property wstack:isDeploymentStepOf and each of these steps has a wstack:ConfigurationInfo representing the set of parameters for executing them.

An overview of this ontology is depicted in Figure 4.3.

4.1.2.3 Hardware Specs Ontology

The Hardware Specs Ontology is meant to describe the hardware characteristics of a computational infrastructure and the hardware requirements of a software component. As well as in the Software Stack Ontology, the concepts of this ontology can be used to describe either the hardware characteristics of a computational resource or the hardware specification required by a workflow.

This ontology describes the set of hardware characteristics/requirements of an element by means of the class whw:HardwareSpec, which aggregates a set of whw:HardwareComponent, such as the CPU or the RAM memory. Each of these components have one or more measurable characteristics, described by the whw:Feature class. This class

Figure 4.4: Scientific Virtual Appliance Ontology overview.

includes two properties for defining what we are measuring (`whw:unit`) and its value (`whw:value`).

4.1.2.4 Scientific Virtual Appliance Ontology

This ontology defines the main concepts related to virtualized computational resources, provided by virtualization solutions and infrastructure providers (e.g. IaaS cloud providers). It focuses on Scientific Virtual Appliances, that is, the assembly of virtual hardware and software components designed within the scope of a scientific process.

The main concept of this ontology is the `wsva:ScientificVirtualAppliance` class, a computational resource described as a Virtual Machine with a certain hardware specification and a concrete set of software components deployed on it. These resources define a set of features specifically designed for supporting the execution of a scientific workflow.

`wsva:ScientificVirtualAppliance` is based on an `wsva:ImageAppliance` that aggregates a set of `wsva:VMImage`. This image can be used for enacting a Virtual Machine in a certain `wsva:InfrastructureProvider`. Two different `wsva:VMImage` having the same software and hardware specification would expose the same feature from the point of view of workflow execution, and therefore they are grouped under a single appliance. Which image to use could be later decided based on other criteria, such as pricing or size, but both will be equally capable of supporting the execution of a given process.

4.1.2.5 The WICUS ontology

As a result of the modular design of our models we implemented an ontology network composed by the ontologies described above. Thus, we have developed a top-level ontology defining five object properties that link the main concepts from the domain ontologies.

To relate the requirements of a workflow, represented by the `wreqs:wreqs:SoftwareRequirements` and `wreqs:HardwareRequirements` classes, we defined the `wicus:composedBySoftwareStack` and `wicus:composedByHardwareSpec` object properties, that relate these requirements to the `wstack:SoftwareStack` and `whw:HardwareSpec` classes respectively.

We also need to relate the computational resources domain with the software and hardware domains for describing the capabilities of a resource. In order to do that we have created the `wicus:hasHardwareSpecs` and `wicus:hasSoftwareStack` properties, that also link to the `wstack:SoftwareStack` and `whw:HardwareSpec` classes respectively.

Finally, to state the hardware specification required by a piece of software we created the `wicus:requiresHardware` object property. This property relates the `wstack:SoftwareComponent` class with the `whw:HardwareSpec` class.

The ontology network described in this chapter aims to serve as a reference framework for the description of the execution environments of scientific experiments. Throughout the following chapters we will explain the role that these ontologies play in our system, describing how they are used along the reproducibility process of several computational experiments.

As a result of these experiments, we obtained valuable feedback about the ontologies, identifying the required changes to improve them. These changes have been incorporated to the WICUS network over the different internal and public releases. The modular design we selected for the ontologies minimized the impact of modifying them. These changes have been also reflected in the tools that make use of these semantic models, which we introduce in Chapter 5.

We plan to maintain the WICUS ontology network and its public documentation, adapting it when necessary to the different necessities that may appear in the future.

Chapter 5

Experiment Reproduction

“The purpose which that engine has been specially intended and adapted to fulfill, is the computation of nautical and astronomical tables”

Ada Lovelace, notes upon “Sketch of The Analytical Engine Invented by Charles Babbage”, 1842.

Based on the ontology network described in Chapter 4, we introduce in this chapter a set of tools that facilitate the annotation of the resources involved in the execution of a scientific workflow. These tools aim to automate as much as possible the process of generating the semantic descriptions of workflow execution dependencies. User interaction may still be required in this process, as in some cases not all the dependencies are fully specified in the workflow definition and its related files.

We also introduce several modules for consuming the generated annotations to specify and enact the execution environment. These modules take into account the available infrastructure providers and design a deployment plan. This plan will define the necessary steps for creating and customizing the required computational resources.

Fig. 5.1 shows an overall view of the execution flow of the WICUS process to achieve the reproducibility of a workflow, as well as its main input and outputs. A more detailed diagram of the process, including sub-modules and intermediate results, is depicted in Fig. 5.2.

Below we describe each of these modules and their inputs, as well as the results

Figure 5.1: WICUS system overview.

that we have obtained by executing them. These results include both the final specification files for enacting the reproduced infrastructure and several indirect results used throughout the process, including the description of the tools required by the workflow and how they relate to each part of the it.

5.1 Parsing Tools & Semantic Annotations

Workflow Management Systems and Languages provide information within the specification of each workflow, describing their inner structure and some of their related components. This includes the activities to be performed and usually the associated data inputs and the information about the computational resources for executing them.

This information may include the binary files and its physical location, the list of modules which a certain script import, or even specific configuration parameters for the software components. Depending on which WMS we are working with, we will have to parse different files under different languages and formats.

In this section we describe the different files that we are able to parse and introduce the tools that we have developed in this work. Some of these files are manually gener-

Figure 5.2: Detailed diagram of the WICUS system.

ated, as we have not been able to identify an explicit and structured information source to be used as the input of an automated process. This mainly happens when describing resources that are out of the scope the workflow definition itself, such as the workflow system or the computational nodes.

5.1.1 Workflow Specification File

Following the conventions defined by the corresponding language, workflows are usually defined and encoded into files, which establish the set of activities to be performed. These specifications contain the steps of the workflow and their order and hierarchy. Other assets such as the data inputs and some software dependencies are often included on them. As mentioned before, most of the state-of-the-art scientific workflow systems support the definition of Direct Acyclic Graphs (DAG), whereas few of them allow defining workflows containing cycles (non-DAG workflows).

In this work we analyze the workflow definition in order to relate each part of the workflow (i.e. sub-workflows) with their corresponding requirements. Our system is designed to analyze both, DAG and non-DAG workflows, even when in this work we have only processed DAG ones, which are the most common scenario in computational experiments.

Besides the structure of the workflow, the format of the file defining it also varies with every language. This results in a wide range of specifications, from markup languages, such as XML, to declarative workflow definitions using programming languages, such as Perl or Python. For each of them we must be able to detect the names of the activities that compose the workflow as well as how they are related to each other.

Another important feature to take into account when analyzing workflow specifications is whether they are abstract or concrete. This distinction depends on the level of detail that the system is expecting, and varies even during the lifecycle of a workflow. Some WMSs define abstract workflows as those that do not fully define one or more of their steps. When the end user or the execution system chooses the concrete implementation for all of them, it can be considered as concrete. These concrete specifications might be considered still as abstract by other WMS, as those systems require not only the step implementations to be selected, but also the target computational resources and data assets to be fully specified.

As we introduce in Section 6.2.1, the Pegasus WMS declares workflows using an XML-based specification file, which include the steps of the workflow but not its target resources. Due to this, these specifications are considered abstract by Pegasus, and until the system fixes their execution time, it is not considered to be concrete. In the context of our system however, these workflow specifications are considered as concrete, as they already define the steps and resources involved on it, which is enough information for carrying out our process.

Thus, in this work, our definition of concrete workflow fits the first definition of the two provided above. We aim to reproduce workflows that have specified the implementation of their steps, but not necessarily the target resources. Those resources will be defined, created and customized later in our process.

5.1.2 Workflow Parser & Annotator

To identify the main steps and the relations of the workflows that we are aiming to reproduce, in terms of order and hierarchy, we have implemented a parsing tool that reads the corresponding file looking for their name and/or id. As stated before, the identification of these steps depends on the language and format in which the workflow specification was generated. In this work we have developed a parsing module for each of the WMS we have worked with, which are listed in chapter 6.

Once the identifiers of the workflow, its sub-workflows, steps and relations, have been extracted, the tool starts generating the corresponding set of annotations using the terms of the WICUS vocabulary. These annotations state which are the different parts of the workflow, which steps belong to each of these parts and creates a requirement for each of the sub-workflows. The tool generates at least one sub-workflow for each workflow that it is annotating, as we assume that a workflow always contains at least one step that can be considered to constitute a sub-workflow.

The requirements generated by this module are related to the first level software dependencies, as identified in the workflow specification file. If no information about these dependencies can be identified by this module, they would have to be provided by the user later in the annotations process. Besides, the dependency graph will be completed by the following modules of the WICUS system, including indirect dependencies.

As well as for the software requirements, there are some hardware requirements that can be specified at this stage of the process. This module allows to manually state

capabilities such as the memory and storage requirements of the workflow, which are mostly influenced by the amount of data it is going to process.

5.1.3 Workflow annotations

This RDF file contains the description of the workflow and its direct infrastructure requirements, as generated by the *Workflow Parser & Annotator* tool. The information it contains follows the WICUS ontologies specification and therefore can be consumed by the system seamlessly, regardless of the WMS we are working with. As stated before, the requirements included in this file are yet to be completed. They only include the direct dependencies as declared in the workflow specification.

In order to provide a better understanding of this file, Listing 5.1 contains an summarized version of the annotations generated for the Pegasus SoyKB workflow described in 6.2.1.3, which are graphically represented in Figure 5.3. SoyKB is defined as a workflow in the first two lines, and is related to its sub-workflows in lines 3-6 (several sub-workflows have been omitted for the sake of clarity). Each of these sub-workflows is then defined as a concrete workflow and related to its direct execution requirements, as shown in lines 8-11.

These requirements are stated as software requirements (lines 23-24) and each of them is composed by the set of software stacks that has been identified as direct dependencies (lines 25-26).

The different steps that compose this concrete workflow are also defined in this file, as depicted in lines 13-21, and related to their corresponding workflow.

This module uses the *Workflow Execution Requirements* ontology for defining the structure and requirements of the workflow, and the *Software Stack* ontology for stating the direct software dependencies of their requirements.

Listing 5.1: Pegasus SoyKB workflow annotations example.

```
<http://purl.org/net/wicus-reqs/resource/Workflow/soykb_WF>
  a <http://purl.org/net/wicus-reqs#Workflow>;
  <http://purl.org/net/wicus-reqs#hasSubworkflow>
    <http://purl.org/net/wicus-reqs/resource/ConcreteWorkflow/SORT_SAM_CONC_WF>,
    ...
    <http://purl.org/net/wicus-reqs/resource/ConcreteWorkflow/FILTERING_SNP_CONC_WF>.

<http://purl.org/net/wicus-reqs/resource/ConcreteWorkflow/SORT_SAM_CONC_WF>
  a <http://purl.org/net/wicus-reqs#ConcreteWorkflow>;
  <http://purl.org/net/wicus-reqs#requiresExecutionEnvironment>
```



```

<http://purl.org/net/wicus-stack/resource/SoftwareRequirements/SORT_SAM_SOFT_REQ>.

<http://purl.org/net/wicus-reqs/resource/WorkflowStep/SORT_SAM_ID0000133_WF_STEP>
  a <http://purl.org/net/wicus-reqs#WorkflowStep>;
  <http://purl.org/net/wicus-reqs#isStepOfWorkflow>
    <http://purl.org/net/wicus-reqs/resource/ConcreteWorkflow/SORT_SAM_CONC_WF>.

<http://purl.org/net/wicus-reqs/resource/WorkflowStep/SORT_SAM_ID0000575_WF_STEP>
  a <http://purl.org/net/wicus-reqs#WorkflowStep>;
  <http://purl.org/net/wicus-reqs#isStepOfWorkflow>
    <http://purl.org/net/wicus-reqs/resource/ConcreteWorkflow/SORT_SAM_CONC_WF>.

<http://purl.org/net/wicus-stack/resource/SoftwareRequirements/SORT_SAM_SOFT_REQ>
  a <http://purl.org/net/wicus-reqs#SoftwareRequirements>;
  <http://purl.org/net/wicus#composedBySoftwareStack>
    <http://purl.org/net/wicus-stack/resource/SoftwareStack/PICARD-WRAPPER_SOFT_STACK>.

```

Figure 5.3: Pegasus SoyKB workflow annotations diagram.

5.1.4 WMS annotations

One of the most relevant dependencies of any workflow is the management system that it was designed for. Thus, we have to generate the necessary annotations for each of the WMS we want to work with, stating their components and required dependencies. The structure and composition of these systems highly varies from one system to another. Some of them are composed by a single package, which contains all the necessary dependencies for managing and executing workflows. Others consist of a main management module that relies on other modules for tasks such as scheduling or job submission, thus depending on them.

To properly generate the descriptions of the systems used in this work, we have studied their public documentation and tested their installation process, to ensure that all dependencies and configuration parameters were identified. Due to the lack of structured information about the WMS, which can be processed automatically, this file is manually generated. Even when this process requires a considerable amount of effort and has to be checked by hand, once the annotations have been generated, this file can be reused for documenting any workflow of a given WMS. Annotating a new WMS requires to study its main components and identify their dependencies, thus a certain knowledge about it is required. This process .

An example depicting the annotations of a WMS is contained in listing 5.2, where the main dependencies of the dispel4py system are shown. The first eight lines define the dispel4py WMS software stack, which has one software component and has three known software dependencies. This software component, stated in lines 10-11, defines also a deployment plan (lines 13-16), which contains the set of steps and scripts that has to be performed to deploy it.

As in the case of the previous one, this file is generated using the terms of the WICUS vocabulary, and therefore it can be read by the rest of the components of the system regardless of which WMS it is describing. This file mainly describes software dependencies, using the *Software Stack* ontology for it.

Listing 5.2: dispel4py WMS annotations example.

```
<http://purl.org/net/wicus-stack/resource/SoftwareStack/DISPEL4PY_WMS_SOFT_STACK>
  a <http://purl.org/net/wicus-stack#SoftwareStack>;
  <http://purl.org/net/wicus-stack#hasSoftwareComponent>
    <http://purl.org/net/wicus-stack/resource/SoftwareComponent/DISPEL4PY_WMS_SOFT_COMP>;
  <http://purl.org/net/wicus-stack#dependsOn>
    <http://purl.org/net/wicus-stack/resource/SoftwareStack/UBUNTU_12_04_OS_SOFT_STACK>,
    <http://purl.org/net/wicus-stack/resource/SoftwareStack/PYTHON_2_7_3_SOFT_STACK>,
    <http://purl.org/net/wicus-stack/resource/SoftwareStack/PIP_6_0_8_UBUNTU_SOFT_STACK>.

<http://purl.org/net/wicus-stack/resource/SoftwareComponent/DISPEL4PY_WMS_SOFT_COMP>
  a <http://purl.org/net/wicus-stack#SoftwareComponent>.

<http://purl.org/net/wicus-stack/resource/DeploymentPlan/DISPEL4PY_WMS_DEP_PLAN>
  a <http://purl.org/net/wicus-stack#DeploymentPlan>;
  <http://purl.org/net/wicus-stack#isDeploymentPlanOf>
    <http://purl.org/net/wicus-stack/resource/SoftwareComponent/DISPEL4PY_WMS_SOFT_COMP>.
```

5.1.5 Software Components Registry

As in previous cases, the information describing the necessary resources for executing a workflow is exposed in different ways, depending on the WMS that we are considering. In many cases, this information is encoded within the definition of the workflow, as highlighted before, defining the first level dependencies of the workflow activities. Other systems define specific files or registries for describing the components involved on the workflow execution, which are usually common to several workflows rather than designated for an specific one. These catalogs ease the composition, reuse and repurpose of applications, as they give a broader view of the available tools for data processing.

The information available in these specifications is highly heterogenous, as no standard has been agreed yet by the community for describing scientific software assets. In most cases, only the name of the binaries or packages involved on a certain activity is provided, usually along with their relative or absolute path in the system. More expressive and complete catalogs are sometimes available, including relevant information such as version, hardware specification (e.g. CPU architecture) and target operating system.

We can find an example of this type of information on the Transformation Catalog (TC) files from the Pegasus WMS. These files are the means that Pegasus provides for specifying the tools that are available and that can be referenced by a workflow. Listing 5.3 shows an entry from the TC defined for the Montage workflow introduced in Section 6.2.1.1, from which we can extract the target path of the binary file, its name, the corresponding OS family, and its required architecture.

This information can be parsed and used for annotating the tools included on the files, which will be later used to define how they are related to the workflow and how they will be arranged in the reproduced infrastructure.

Listing 5.3: Montage’s Transformation Catalog entry example.

```
tr mDiffFit:3.3 {
  site condor_pool {
    pfn ‘‘/opt/montage/Montage_v3.3_patched_4/bin/mDiffFit’’
    arch ‘‘x86_64’’
    os ‘‘linux’’
    type ‘‘INSTALLED’’
    profile pegasus ‘‘clusters.size’’ ‘‘10’’
  }
}
```

5.1.6 Software Components Annotator

Different syntaxes and formats are used, depending on the WMS, to define the software components. Therefore, this module contains different parsing implementations, as in the case of the workflow specification, to extract the relevant information of these components. This information is then used to generate annotations according to the WICUS vocabularies.

As described before, not all WMS provide an explicit file or database describing these resources. Also, many times indirect dependencies are not defined neither in the workflow specification nor in the software catalog of the WMS. Thus, this module enables the user to manually define them, allowing extra entries to be added, as well as dependency relations between components to be specified.

In the previous example, depicted in Listing 5.3, an entry of the Montage TC file describing a software binary named *mDiffFit* is listed. The software annotator module is able to read this information, encoded under the Pegasus specification, and extract the properties of interest for our system. Based on it, we obtain a set of annotations using the *Software Stack* ontology introduced in Section 4.1.2.2. Listing 5.4 shows a summarized overview of the result of executing this module over the Montage TC file, in which the lines describing the version and target path properties of the component have been annotated. A more detailed and complete description can be found in the following section.

Listing 5.4: Montage’s Software Components Catalog example.

```
<http://purl.org/net/wicus-stack/resource/ConfigurationParameter/
  MDIFFFIT_DEST_DEP_PLAN_CONF_PAR>
a <http://purl.org/net/wicus-stack#ConfigurationParameter> ;
<http://purl.org/net/wicus-stack#isConfigurationParameterOf>
  <http://purl.org/net/wicus-stack/resource/ConfigurationInfo/
    MDIFFFIT_DEST_DEP_PLAN_CONF_INFO> ;
<http://purl.org/net/wicus-stack#parameterName>
  “-DEST_FILE_PATH”^^xsd:string ;
<http://purl.org/net/wicus-stack#parameterValue>
  “/opt/montage/Montage_v3.3_patched_4/bin/”^^xsd:string .

<http://purl.org/net/wicus-stack/resource/Version/MDIFFFIT_VERSION>
a <http://purl.org/net/wicus-stack#Version> ;
<http://purl.org/net/wicus-stack#versionNumber>
  “3.3”^^xsd:string .
```

5.1.7 Software Components Catalog

Based on the information provided by the software registry, either stated as part of the workflow or defined by the user, we obtain a file containing the set of software components, and their related information, that a workflow and its WMS require to execute computational experiments. This file follows the specification defined by the *Software Stack* ontology, allowing the file to be read by the following modules of the system.

Listing 5.5 contains a snippet of the software catalog generated for the Makeflow version of the BLAST workflow we will study in Section 6.2.3.1, which is also depicted in Figure 5.4. It depicts the information describing a Perl-based script that is required for executing on the steps of the workflow. As shown through the first eight lines, this stack is composed by one software component and depends on other software resources: the *blastx* binary file, the Perl interpreter, and the selected operating system. Lines 10-15 define the main information of the related software component, including its version and executable file. The literal value for those characteristics is defined in lines 17-20 and 22-25 respectively.

Finally, lines 27-41 define the deployment process of the component. This includes the different steps that must be executed, as depicted in lines 32-41, their related configuration information (lines 34-35), the corresponding scripts for carrying out the step (lines 36-37), as well as the next steps.

Listing 5.5: Makeflow BLAST software annotations example.

```
<http://purl.org/net/wicus-stack/resource/SoftwareStack/LEGACY_BLAST_SOFT_STACK>
a <http://purl.org/net/wicus-stack#SoftwareStack>;
<http://purl.org/net/wicus-stack#dependsOn>
  <http://purl.org/net/wicus-stack/resource/SoftwareStack/BLASTX_SOFT_STACK>,
  <http://purl.org/net/wicus-stack/resource/SoftwareStack/
    RPM_PERL_V5_14_4_X86_64_SOFT_STACK>,
  <http://purl.org/net/wicus-stack/resource/SoftwareStack/CENTOS_6_5_OS_SOFT_STACK>;
<http://purl.org/net/wicus-stack#hasSoftwareComponent>
  <http://purl.org/net/wicus-stack/resource/SoftwareComponent/LEGACY_BLAST_SOFT_COMP>.

<http://purl.org/net/wicus-stack/resource/SoftwareComponent/LEGACY_BLAST_SOFT_COMP>
a <http://purl.org/net/wicus-stack#SoftwareComponent>;
<http://purl.org/net/wicus-stack#hasBinary>
  <http://purl.org/net/wicus-stack/resource/SoftwareBinary/LEGACY_BLAST_SOFT_BIN>;
<http://purl.org/net/wicus-stack#hasVersion>
  <http://purl.org/net/wicus-stack/resource/Version/LEGACY_BLAST_VERSION>.
```

```

<http://purl.org/net/wicus-stack/resource/Version/LEGACY_BLAST_VERSION>
  a <http://purl.org/net/wicus-stack#Version>;
  <http://purl.org/net/wicus-stack#versionNumber>
    "1.0"^^xsd:string.

<http://purl.org/net/wicus-stack/resource/SoftwareBinary/LEGACY_BLAST_SOFT_BIN>
  a <http://purl.org/net/wicus-stack#SoftwareBinary>;
  <http://purl.org/net/wicus-stack#binary>
    "legacy_blast.pl"^^xsd:string.

<http://purl.org/net/wicus-stack/resource/DeploymentPlan/LEGACY_BLAST_DEP_PLAN>
  a <http://purl.org/net/wicus-stack#DeploymentPlan>;
  <http://purl.org/net/wicus-stack#isDeploymentPlanOf>
    <http://purl.org/net/wicus-stack/resource/SoftwareComponent/LEGACY_BLAST_SOFT_COMP>.

<http://purl.org/net/wicus-stack/resource/DeploymentStep/LEGACY_BLAST_DEP_STEP>
  a <http://purl.org/net/wicus-stack#DeploymentStep>;
  <http://purl.org/net/wicus-stack#hasDeploymentInfo>
    <http://purl.org/net/wicus-stack/resource/ConfigurationInfo/
      LEGACY_BLAST_DEP_PLAN_CONF_INFO>;
  <http://purl.org/net/wicus-stack#hasDeploymentScript>
    <http://purl.org/net/wicus-stack/resource/DeploymentScript/COPY_BIN_DEP_SCRIPT>;
  <http://purl.org/net/wicus-stack#isDeploymentStepOf>
    <http://purl.org/net/wicus-stack/resource/DeploymentPlan/LEGACY_BLAST_DEP_PLAN>;
  <http://purl.org/net/wicus-stack#nextStep>
    <http://purl.org/net/wicus-stack/resource/DeploymentStep/LEGACY_BLAST_DEP_STEP_2>.

```

5.1.8 Workflow & Configuration Annotation File

This file contains an extended and refined version of the file described in Section 5.1.3, where the *Software Components Annotations* tool has added the information about all the identified dependencies of the workflow activities, which may not have been stated yet by previous steps, as well as the configuration information of the already defined dependencies. This new information is extracted from the *Software Components Registry* and points to the elements described in the *Software Components Catalog* when necessary.

As shown in listing 5.6, this file states the configuration information of a given component (lines 1-7) and the configuration parameters that compose it. Lines 9-16 describes that the deployment plan of the *mProjectPP* component has a parameter defining the path in which its binary must be deployed. Software dependencies of these components, as well as their deployment and configuration information, is defined also in the software catalog described before.

Figure 5.4: Makeflow BLAST software annotations diagram.

Listing 5.6: Pegasus Montage workflow and configuration annotations file example.

```
<http://purl.org/net/wicus-stack/resource/SoftwareComponent/MPROJECTPP_SOFT_COMP>
a <http://purl.org/net/wicus-stack#SoftwareComponent>;
<http://purl.org/net/wicus-stack#hasConfigurationInfo>
<http://purl.org/net/wicus-stack/resource/ConfigurationInfo/MPROJECTPP_CONF_INFO>.

<http://purl.org/net/wicus-stack/resource/ConfigurationInfo/MPROJECTPP_CONF_INFO>
a <http://purl.org/net/wicus-stack#ConfigurationInfo>.

<http://purl.org/net/wicus-stack/resource/ConfigurationParameter/
MPROJECTPP_DEST_DEP_PLAN_CONF_PAR>
a <http://purl.org/net/wicus-stack#ConfigurationParameter>;
<http://purl.org/net/wicus-stack#isConfigurationParameterOf>
<http://purl.org/net/wicus-stack/resource/ConfigurationInfo/MPROJECTPP_CONF_INFO>;
<http://purl.org/net/wicus-stack#parameterName>
“-DEST_FILE_PATH”^^xsd:string;
<http://purl.org/net/wicus-stack#parameterValue>
“/opt/montage/Montage_v3.3_patched_4/bin/”^^xsd:string.
```

5.1.9 Scientific Virtual Appliances Catalog

Besides all the information generated based on the workflow files and related materials, we need to define which are the computational resources available for supporting the reproducibility of a scientific workflow. Thus, we generate an RDF catalog, using the *Scientific Virtual Appliance* ontology, describing the relevant information of the set of appliances that our system can use.

This file states both, the software and hardware information about the virtual machines from different cloud solutions. The annotations contained in it are generated manually and once produced, can be reused by the reproducibility process of different workflows using different WMS.

An example of the annotations describing a virtual appliance is shown in listing 5.7, in which an Ubuntu-based computational resource is described. This appliance is characterized by the software stacks it provides, defined in lines 3-5. Different image appliances are capable to enact and provide the software characteristics exposed by the appliance. In this example, two different images, belonging to two different cloud solutions, are stated, one using Vagrant and the other one using AWS EC2 (lines 6-8).

The Amazon image is described in lines 10-18, including the definition of its hardware specification, its provider, and the virtual machine image for executing it. Three different hardware characteristics are defined, describing the image's CPU, memory, and storage capacity (lines 23-25). In this case, its memory is described by the amount and unit of RAM memory it exposes (lines 27-30). Finally, the identifier of the related VM image is defined in line 34.

This is a multi-domain file which includes concepts from the *Scientific Virtual Appliance*, *Software Stack*, and *Hardware Specs* ontologies, as well as some properties from the *WICUs* main ontology to related them.

Listing 5.7: SVA catalog example.

```
<http://purl.org/net/wicus-sva/resource/ScientificVirtualAppliance/UBUNTU_12_04_SVA>
a <http://purl.org/net/wicus-sva#ScientificVirtualAppliance>;
<http://purl.org/net/wicus#hasSoftwareStack>
  <http://purl.org/net/wicus-stack/resource/SoftwareStack/PYTHON_2_7_3_SOFT_STACK>,
  <http://purl.org/net/wicus-stack/resource/SoftwareStack/UBUNTU_12_04_OS_SOFT_STACK>;
<http://purl.org/net/wicus-sva#isSupportedBy>
  <http://purl.org/net/wicus-sva/resource/ImageAppliance/VAGRANT_UBUNTU_12_04_ImgApp>,
  <http://purl.org/net/wicus-sva/resource/ImageAppliance/UBUNTU_12_04_AWS_ImgApp>.
```



```

<http://purl.org/net/wicus-sva/resource/ImageAppliance/UBUNTU_12_04_AWS_ImgApp>
a <http://purl.org/net/wicus-sva#ImageAppliance>;
<http://purl.org/net/wicus#hasHardwareSpecs>
  <http://purl.org/net/wicus-hwspecs/resource/HardwareSpec/UBUNTU_12_04_AWS_HW_SPECS>;
<http://purl.org/net/wicus-sva#providedBy>
  <http://purl.org/net/wicus-sva/resource/InfrastructureProvider/
    AmazonWebServices_CLOUD_PROV>;
<http://purl.org/net/wicus-sva#hasVMImage>
  <http://purl.org/net/wicus-sva/resource/VMImage/UBUNTU_12_04_AWS_VMImage>.

<http://purl.org/net/wicus-hwspecs/resource/HardwareSpec/UBUNTU_12_04_AWS_HW_SPECS>
a <http://purl.org/net/wicus-hwspecs#HardwareSpec>;
<http://purl.org/net/wicus-hwspecs#hasHardwareComponent>
  <http://purl.org/net/wicus-hwspecs/resource/CPU/UBUNTU_12_04_AWS_CPU>,
  <http://purl.org/net/wicus-hwspecs/resource/Memory/UBUNTU_12_04_AWS_RAM>,
  <http://purl.org/net/wicus-hwspecs/resource/Storage/UBUNTU_12_04_AWS_STORAGE>.

<http://purl.org/net/wicus-hwspecs/resource/Feature/UBUNTU_12_04_AWS_RAM_size>
a <http://purl.org/net/wicus-hwspecs#Feature>;
  <http://purl.org/net/wicus-hwspecs#value '7'>;
  <http://purl.org/net/wicus-hwspecs#unit 'GB'>.

<http://purl.org/net/wicus-sva/resource/VMImage/UBUNTU_12_04_AWS_VMImage>
a <http://purl.org/net/wicus-sva#VMImage>;
  <http://purl.org/net/wicus-sva#vmId> 'ami-9a404af'.

```

5.2 Specification Process

The annotations produced by the first set of modules of the WICUS system constitute the foundation of our reproducibility cycle and serve as the starting point for the following modules of the process. Having a structured and well-defined source of information allows building tools that carry out the recreation and repetition of a computational experiment environment.

The first step consists in selecting the appropriate resources for supporting a new workflow execution. This process can be implemented in many ways, using different criteria for resource selection. As argued before in this work, we do not aim to produce an exact copy of the former infrastructure of a scientific workflow, but rather one that is able to reproduce it and obtain equivalent results. Therefore, different infrastructure configurations can be defined, all of them valid as long as they meet the requirements declared on the annotations.

In this section we describe the specification process that we have designed and implemented for the WICUS system, and that we will test on our experimental evaluation.

The elements described in this section, the specification algorithm and the data it produces, compose the second stage on the WICUS process depicted in Figure 5.2.

5.2.1 Infrastructure Specification Algorithm

The Infrastructure Specification Algorithm (ISA) is a specification process that retrieves the corresponding information of a given workflow and its main dependencies from the annotation datasets and, based on them, calculates the full set of dependencies and compatibility between requirements and the available computational resources. This implementation also aims to reduce resource consumption by considering the software that is already available on the selected computational resources, in order to avoid unnecessary deployment effort. It also tries to merge requirements into compatible appliances, reducing the number of machines created.

We define the main goal of this algorithm as to produce a structured and technology-independent enactment plan, which must: i) create the appropriate computational resources meeting the requirements of the targeted workflow, ii) deploy the necessary pending dependencies, and iii) configure them in order to meet the specific needs of the chosen workflow.

The process starts by loading the annotations describing the resources associated to the workflow that the user is aiming to reproduce, as well as the general shared datasets, such as the virtual appliance and the software components catalog. It then retrieves the software requirements of the target workflow, which, as explained before, are composed by the direct dependencies of the different activities that compose it.

We can distinguish between two main types of dependencies, workflow-level dependencies and activity dependencies. The first ones are those that are imposed by the WMS and that usually related to the top-level workflow, as they are common to all its steps. The second ones are defined by each concrete activity, and vary from one step to another. This means that in order to execute a given sub-workflow, we have to satisfy its inner requirements as well as all the requirements imposed by its ancestor workflows.

As depicted in Figure 5.5, several levels of workflows can be defined through the sub-workflow property, each of them defining their own requirements, creating a hierarchy tree. According to our conceptualization, only leaf workflows on the hierarchy tree are executable, as they are the ones that contain executable steps. Hence, the ISA calculates which are the requirements for each of those sub-workflows, as depicted in Figure 5.6.

In this example, for meeting the requirements of the WF_5 workflow, it is necessary to guarantee that its own defined requirements are satisfied, as well as the ones defined by WF_1 and WF_2 .

Figure 5.5: Workflow requirements description.

Figure 5.6: Workflow requirements propagated.

This way, the algorithm obtains the list of propagated requirements associated to each leaf workflow. As explained before, each requirement is composed by a set of software stacks. Based on the list of propagated requirements the ISA retrieves the set of stacks for each of them. These stacks may have software dependencies defined in the software catalog, thus it calculates the dependency graph for each of the requirements.

Starting from the stacks defined by a requirement, the algorithm iterates the software catalog generated in 5.1.7, adding new stacks each recursive iteration. The algorithm must be aware that this graph may contain circular dependencies. Even though these dependencies are a bad practice in software design and development, they usually occur

in functional programming and among non-expert developers, which may be the case of a scientist developing a computational experiment without sufficient skills on programming. In this work we have not faced any circular dependency. However our algorithm is able to deal with them avoiding infinite loops by keeping track of the visited nodes and skipping them over the recursive iterations.

Once the ISA has calculated the dependency graphs for each workflow, it can begin to select the computational resources that would better fit each workflow necessities. In order to do so, it selects all the available virtual appliances from the catalog described in Section 5.1.9 and calculates the compatibility between the stacks contained in the graphs and the ones listed by each virtual appliance. To do that, it calculates the intersection between the set of stacks of the graph and the set of stacks of the appliance. This intersection is required to be greater than a certain configurable threshold. Appliances below this threshold are removed from the candidates list.

The algorithm then sorts the resultant appliances for obtaining the one with the greater intersection for each requirement, arguing that the more components they have in common, the less deployment effort would be needed, making it more suitable. At this point, the ISA has selected a SVA for each workflow, and from its requirements it has the set of software stacks that have to be deployed on them. Using the intersection calculated before, the ISA removes the unnecessary stacks from the dependency graph, as they are already deployed in the appliance. It removes each stack included in the intersection and its derived dependencies recursively. That is, the dependencies that have no other stack depending on them and therefore get isolated once the stacks from the intersections have been removed. Figures 5.7 and 5.8 depict how this filtering process is carried out. When removing the SW_2 stack, it also removes SW_4 , as there is no need of installing it. SW_5 is not deleted from the graph as other stacks depends on it.

As introduced before, this implementation of the specification algorithm aims to reduce resource consumption. Therefore, once the appliances for each set of requirements have been selected, the ISA tries to group different requirements into the same computational resource. If two sets of requirements share the same SVA, then their software components can be allocated in the same virtual machine. This implementation of the algorithm does not take into account potential software stacks incompatibilities. If such incompatibilities were detected between two stacks belonging to different requirements, then those requirements could not be grouped together.

Figure 5.7: Dependency graph example.

Figure 5.8: Dependency graph removing process.

The final step on the selection process is to filter the virtual machine images based on the hardware requirements stated by the workflow. As stated before, a SVA can be composed by several different VM images, all of them sharing the same software characteristics but that may expose different hardware specifications. Thus, the ISA filters those images that do not meet the required needs for executing the workflow.

Once the algorithm has calculated the computational resources that have to be created and the software components that must be deployed on them, it can generate the deployment plan that will enact the infrastructure. It will generate the set of

instructions to enact each of the VM based on the selected images and the steps for deploying the software stacks. The deployment order of these stacks is calculated by doing a reverse topological sort of the dependency graphs which contain them. A topological sort of the dependency graph would produce a list of stacks where the ones declared by the requirements will be listed first. In this work we argue that those direct dependencies have to be at the very end of the deployment process. That is, all the dependencies of a component should be installed before the component itself, as the result of this installation process could be necessary for installing a dependent software package.

We have developed this implementation of the ISA in Java SE ⁶⁰, using Jena 2.10.2⁶¹ for managing the semantic information of our system, which is stored locally. Jena provides an SPARQL⁶² endpoint which we query for retrieving information.

A pseudo-code overview of the algorithm is depicted in listing 5.8. The process starts by loading the annotated datasets, based on the WICUS ontology network, in order to find a suitable infrastructure specification that meets the requirements of the workflow. The algorithm retrieves and propagates the WMS requirements of the top-level workflow to its related sub-workflows. Requirements and software components are matched, and a dependency graph is built based on the relationship between the requirements and the component dependencies (line 13). ISA then calculates the intersection between the set of stacks installed on the SVA and the set of stacks from the dependency graph, selecting the SVA that maximizes the value of that intersection for each sub-workflow (line 17). Software components already available in the SVA are then removed from the dependency graph, as they do not have to be installed (line 19). To reduce the number of SVAs, the algorithm attempts to merge sub-workflow requirements into a single SVA (line 21). Requirements can be merged if they share the same SVA and all their software components are compatible. Then, the algorithm filters those appliances that do not meet the hardware requirements specified for the workflow (lines 23–25). Finally, ISA generates a deployment plan with the set of required instructions to instantiate, deploy, and configure the computational resources and software components on the corresponding infrastructure provider (line 27). This *Abstract Deployment Plan*,

⁶⁰<http://www.oracle.com/technetwork/articles/javaee/index-jsp-136246.html>

⁶¹<http://jena.apache.org/>

⁶²<http://www.w3.org/TR/rdf-sparql-query/>

explained in detail below, enables representing a generic description of the enactment and deployment process, which defines the steps and scripts to be executed, along with their configuration parameters. This intermediate result can be later concretized to generate executable scripts using the different solutions supported by the system.

Listing 5.8: Pseudo-code overview of the Infrastructure Specification Algorithm (ISA).

```
WorkflowRequirementsDataset.load();

SVADataset.load();

SoftwareCatalogDataset.load();

Map<Workflow, List<Requirements>> wfSwReqs =
    retrieveSwRequirements(WorkflowRequirementsDataset, WorkflowID);

Map<Workflow, List<Requirements>> propagatedWfSwReqs =
    propagateSwReqs(wfSwReqs);

List<List<List<SWStack>>> softwareStacks =
    getSoftwareComponents(propagatedWfSwReqs);

Map<Requirement, D-Graph<SWStack>> softwareComponentsDepGraph =
    getSoftwareDependencies(softwareStacks);

List<SVA> availableSvas = getAvailableSvas(providersList);

Map<Requirements, SVA> maxCompatibilities =
    getCompatibilityIntersection(softwareComponentsDepGraph, availableSvas);

Map<Requirement, D-Graph<SWComponents>> substractedSwCompDepGraph =
    substractSoftwareComponents(softwareComponentsDepGraph, maxCompatibilities);

Map<SVA, List<Requirments>>mergedSvas =
    mergeSubworkflows(propagatedWfSwReqs, maxCompatibilities);

Map<Workflow, List<Requirements>> wfHwReqs =
    retrieveHwRequirements(WorkflowRequirementsDataset, WorkflowID);

Map<SVA, List<Requirments>>filteredSvas =
    getCompatibleHwImageAppliances(mergedSvas, wfHwReqs);

generateAbstractDeploymentPlan(filteredSvas, substractedSwCompDepGraph);
```

The ISA implementation developed and evaluated in this thesis can be accessed, both

as an executable file and as source code, in a public repository⁶³. In the experimentation of this thesis we have used the v.1.1 version of the algorithm.

5.2.2 Abstract Deployment Plan

In order to obtain a general representation of the deployment plans that will define the reproduced infrastructure, the WICUS system defines an abstraction layer. This allows the WICUS system to generate deployment plans regardless of the underlying execution tool. This *Abstract Deployment Plan*, which is based on the *Software Stack* ontology, defines the stacks to be deployed in the execution platform. Figure 5.9 shows the relationships between the different elements that compose the stack domain ontology. A stack can be composed of one or more software components. Each of them has an associated deployment plan according to the target execution platform, which is composed of one or more deployment steps.

The abstract deployment plan contains those steps, along with the information about the scripts or commands required to execute them. This information can be later translated into syntax specified by the different technological solutions covered by the WICUS system. This approach allows the integration of new solutions in the system by adding new translation modules as explained in the next section.

Figure 5.9: Overview of the WICUS software stack relation diagram.

Listing 5.9 shows an example of the abstract plan for the Pegasus SoyKB workflow. The first section of the plan (lines 1–26) describes the deployment of the WMS (i.e. Pegasus) and its related dependencies, which include SSH⁶⁴ or Condor. Note that this section would be common across all deployment plans for the workflows supported by this WMS. The remaining lines describe how the SoyKB software is deployed. The

⁶³<https://github.com/idafensp/WicusISADemos>

⁶⁴<http://www.openssh.com>

SOFTWARE.TAR.GZ stack, which is a dependency for all SoyKB wrappers, is the first component to be deployed (lines 27–29). Finally, the last section of the plan (lines 30–45) describes how the four SoyKB wrappers are deployed. For each wrapper, two deployment steps are required: i) copy of the program execution binary, and ii) the granting of proper execution permissions.

Listing 5.9: Abstract deployment plan of the Pegasus SoyKB workflow.

```

OPEN_SSH_CLIENTS_SOFT_STACK stack
  OPEN_SSH_CLIENTS_SOFT_COMP component
    OPEN_SSH_CLIENTS_DEP_STEP step
OPEN_SSH_SERVER_SOFT_STACK stack
  OPEN_SSH_SERVER_SOFT_COMP component
    OPEN_SSH_SERVER_DEP_STEP step
WGET_SOFT_STACK stack
  WGET_SOFT_COMP component
    WGET_DEP_STEP step
CONDOR_CENTOS_6_5_SOFT_STACK stack
  CONDOR_CENTOS_6_5_SOFT_COMP component
    STOP_CONDOR_DEP_STEP step
    ADD_CONDOR_REPO_DEP_STEP step
    CONDOR_YUM_INSTALL_DEP_STEP step
    CLEAN_AND_SET_CONDOR_DEP_STEP step
    RESTART_DAEMONS_DEP_STEP step
JAVA-1.7.0-OPENJDK.X86_64_SOFT_STACK stack
  JAVA-1.7.0-OPENJDK.X86_64_SOFT_COMP component
    JAVA-1.7.0-OPENJDK.X86_64_DEP_STEP step
JAVA-1.7.0-OPENJDK-DEVEL.X86_64_SOFT_STACK stack
  JAVA-1.7.0-OPENJDK-DEVEL.X86_64_SOFT_COMP component
    JAVA-1.7.0-OPENJDK-DEVEL.X86_64_DEP_STEP step
PEGASUS_WMS_CENTOS_6_5_SOFT_STACK stack
  PEGASUS_WMS_CENTOS_6_5_SOFT_COMP component
    ADD_PEGASUS_REPO_DEP_STEP step
    PEGASUS_YUM_INSTALL_DEP_STEP step
SOFTWARE_TAR_GZ_SOFT_STACK stack
  SOFTWARE_TAR_GZ_SOFT_COMP component
    SOFTWARE_TAR_GZ_DEP_STEP step
PICARD-WRAPPER_SOFT_STACK stack
  PICARD-WRAPPER_SOFT_COMP component
    PICARD-WRAPPER_DEP_STEP step
    PICARD-WRAPPER_DEP_STEP_2 step
SOFTWARE-WRAPPER_SOFT_STACK stack
  SOFTWARE-WRAPPER_SOFT_COMP component
    SOFTWARE-WRAPPER_DEP_STEP step
    SOFTWARE-WRAPPER_DEP_STEP_2 step
GATK-WRAPPER_SOFT_STACK stack
  GATK-WRAPPER_SOFT_COMP component
    GATK-WRAPPER_DEP_STEP step
    GATK-WRAPPER_DEP_STEP_2 step
BWA-WRAPPER_SOFT_STACK stack

```

```

BWA-WRAPPER_SOFT_COMP component
  BWA-WRAPPER_DEP_STEP step
  BWA-WRAPPER_DEP_STEP_2 step

```

5.3 Enactment and Execution

The last stage of the reproducibility process is to translate the deployment plan introduced above from abstract descriptions into the concrete syntaxes of the different solutions and tools integrated in our system. In this section we introduce the main concepts related to this translation process and describe the technologies that have been integrated in this work.

5.3.1 Script Generator

In order to generate a concrete deployment specification, according to a specific syntax, the *Script Generator* must be able to map the concepts described in the abstract deployment plan into the corresponding set of language primitives of each solution. The information for generating the deployment commands is codified in the deployment steps and their related configuration information from the abstract deployment plan. Thus, for each step, the *Script Generator* retrieves their related deployment script and the parameters associated to the configuration information and uses it to generate the set of primitives that will allow to execute them.

In this work we have studied two enactment solutions, PRECIP (Azarnoosh et al., 2013) and Vagrant (Palat, 2012), which cover remote and local deployment of computational resources respectively. Depending on which one of them the user is targeting, the process and primitives to be generated vary. As depicted before in listing 5.9, the last software component to be deployed for the SoyKB workflow is the BWA wrapper. This component specifies a deployment plan composed by two steps. Listings 5.10 and 5.11 describe how these steps are translated into PRECIP and Vagrant.

Listing 5.10: SoyKB Deployment Script using PRECIP ('#' denotes comments).

```

# [STACK] Deployment of BWA-WRAPPER_SOFT_STACK stack
# [COMPONENT] Deployment of BWA-WRAPPER_SOFT_COMP component
# [STEP] Execution of BWA-WRAPPER_DEP_STEP step

# copy copyBinary.sh to the instance
exp.put(["inst0"], "copyBinary.sh", "/home/cloud-user/copyBinary.sh", user="root")

```

```

# granting execution for copyBinary.sh
exp.run(["inst0"], "chmod_755_/home/cloud-user/copyBinary.sh", user="root")

# executing the copyBinary.sh script
exp.run(["inst0"], "/home/cloud-user/copyBinary.sh_DEST_FILE_PATH
/home/cloud-user/workflow/soykb/wrappers/_SOURCE_FILE_PATH
_http://example.org/components/soykb/wrappers/bwa-wrapper_", user="root")

# [STEP] Execution of BWA-WRAPPER_DEP_STEP_2 step

# copy grantExecPer.sh to the instance
exp.put(["inst0"], "grantExecPer.sh",
"/home/cloud-user/grantExecPer.sh", user="root")

# granting execution for grantExecPer.sh
exp.run(["inst0"], "chmod_755_/home/cloud-user/grantExecPer.sh", user="root")

# executing the grantExecPer.sh script
exp.run(["inst0"], "/home/cloud-user/grantExecPer.sh_PP
/home/cloud-user/workflow/soykb/wrappers/bwa-wrapper_", user="root")

```

Listing 5.11: SoyKB Deployment Script using Vagrant (‘#’ denotes comments).

```

# [STACK] Deployment of BWA-WRAPPER_SOFT_STACK stack
# [COMPONENT] Deployment of BWA-WRAPPER_SOFT_COMP component
# [STEP] Execution of BWA-WRAPPER_DEP_STEP step

# executing the copyBinary.sh script
config.vm.provision :shell, path: "copyBinary.sh", args:
["-DEST_FILE_PATH", "/home/vagrant/workflow/soykb/wrappers/", "-SOURCE_FILE_PATH",
"http://example.org/components/soykb/wrappers/bwa-wrapper"]

# [STEP] Execution of BWA-WRAPPER_DEP_STEP_2 step

# executing the grantExecPer.sh script
config.vm.provision :shell, path: "grantExecPer.sh", args:
["-PP", "/home/vagrant/workflow/soykb/wrappers/bwa-wrapper"]

```

For evaluation purposes, the *Script Generator* component may add other specific commands to the generated scripts, such as input data transfer or workflow execution. These commands are manually specified, as they are out of the scope of our approach and they serve only for evaluation purposes.

5.3.2 Enactment Files

As introduced before, in this work we have integrated two deployment solutions into the WICUS system: PRECIP, for enacting and managing remote resources and customize

them, and Vagrant, which allows to create computational nodes in a local environment. While the first one bring access to a wide range of providers, allowing to use more powerful computing resources, the second reduces transfer time as files can be shared locally.

5.3.2.1 PRECIP

The Pegasus Repeatable Experiments for the Cloud in Python (PRECIP) (Azarnoosh et al., 2013) is a flexible experiment management control API for running experiments on all types of Clouds, including academic Clouds such as FutureGrid⁶⁵ and the NSF-Cloud^{66,67} (through OpenStack), and commercial Clouds such as Amazon EC2⁶⁸ and Google Compute Engine⁶⁹. In PRECIP, when an instance is provisioned, the scientist can add arbitrary tags to that instance in order to identify and group the instances in the experiment. Then, future interactions can be performed by using the given tags.

API methods such as running remote commands, or copying files, all use tags to specify which instances to target. PRECIP does not force the scientist to use a special VM image, and no PRECIP components need to be pre-installed in the image. Scientists can use any basic Linux image and PRECIP will bootstrap instances using SCP and SSH commands. PRECIP provides functionality to run user-defined scripts on the instances to install/configure software and run experiments, and also manages SSH keys and security groups automatically.

5.3.2.2 Vagrant

Vagrant (Palat, 2012) is an open-source and multi-platform solution for deploying development environments locally using virtualization. It relies on virtualization solutions such as Oracle VirtualBox (Watson, 2008) (also open-source) or VMWare⁷⁰, and supports Amazon EC2-like server configurations. Since version 1.6 it also supports Docker (Merkel, 2014) containers. Vagrant provides a set of commands and configuration files to enact and customize virtual machines (also referred to as boxes). It allows

⁶⁵<http://portal.futuregrid.org>, last accessed 2016

⁶⁶<http://www.chameleoncloud.org>, last accessed 2016

⁶⁷<http://cloudlab.us>, last accessed 2016

⁶⁸<http://aws.amazon.com/ec2>, last accessed 2016

⁶⁹<https://cloud.google.com/compute>, last accessed 2016

⁷⁰<http://www.vmware.com>, last accessed 2016

defining the set of commands and/or scripts to be executed during the different stages of the booting process. Several base images are publicly available for users to download and customize⁷¹.

We use Vagrant as a solution for achieving reproducibility in a local execution environment, usually a scientist's laptop/desktop computer. As a result, users are able to repeat and modify their original experiment, repurposing or improving it, which is a highly desirable goal of any reproducibility process.

The tools described in this chapter define the different stages of the reproducibility process of our approach. Starting from the available resources, combined with the knowledge from the corresponding experts (mainly the workflow developers and the WMS teams), the WICUS system carries out the reproducibility cycle to obtain a new infrastructure. As introduced before, we have detected during the development of the different modules, the need of user interaction to complete the pieces of information generated automatically. This is due to the way in which computational experiments have been released and documented so far, not providing sufficient and explicit insights about the related execution environments. Even when some information is provided by the files associated to the workflows, it is not always feasible to expect them to contain all the necessary information, thus making human expertise required.

In the following chapter we will introduce several real-case scenarios, where we have applied our approach, explaining the problems we found and how we solved them.

⁷¹<http://www.vagrantbox.es>, last accessed 2016

Chapter 6

Evaluation

“It doesn’t make any difference how beautiful your guess is, it doesn’t matter how smart you are, or what his name is. If it disagrees with experiment, it’s wrong. That is all there is to it.”

Richard Feynman, 1964

To evaluate the hypotheses introduced in Chapter 3 and test the feasibility of our approach, we have designed and executed an evaluation process which applies the contributions of this thesis over a representative experimental sample. The three main aspects to evaluate on this work, according to the corresponding hypotheses described in Section 3.2, are how semantic technologies, the algorithmic process, and the virtualization solutions introduced in this work solve the research challenges stated in Section 3.1.

In order to do so, we need to apply the contributions of this thesis to real case scenarios, aiming to reproduce computational experiments by means of the conservation approach and the reproducibility process that we have envisioned. Our system must be able to execute already existing workflows, obtaining equal or equivalent results, starting from the same input data. We have to test that it is possible to describe the execution environment of these workflows, proving that these descriptions are sufficient by using them to generate a new infrastructure specification, and validate that such specification is capable of supporting the execution of a given workflow by enacting computational resources and tuning them accordingly. Using these resources we must be able to repeat the execution and verify the results.

To this end, we have selected a sample set of computational experiments designed as scientific workflows, containing six different applications. These workflows belong to three different scientific domains and were originally implemented using three different workflow management systems. We have documented their execution environments using the WICUS ontologies and the corresponding annotation and specification tools. We have conducted this reproducibility process using different cloud computing providers, and covering local deployment as well as remote ones, both public and private.

6.1 Experimental Setup

Three different environments have been selected as target infrastructures for deploying the reproduced environment specifications. These infrastructures support the enactment of the computational resources specified by the ISA algorithm, as described in Section 5.2.1, and the execution of the reproduced experiments.

To cover a representative range of them, in our experimentation process we have used paradigmatic examples of local infrastructures, such as local servers, as well as remote execution infrastructures, using an academic-based private cloud provider and a commercial public one. We rely on virtualization solutions, either in the form of local software solutions or as external providers, to incarnate the required resources for executing a scientific workflow. We have selected three providers (FutureGrid, Amazon EC2, and Vagrant) for our experimentation, based on their degree of adoption and representativeness.

6.1.1 FutureGrid

FutureGrid, currently maintained as FutureSystems⁷², is an academic cloud testbed facility that includes a large number of computational resources⁷³, including 481 nodes with 1126 CPUs and 4696 cores, which scale up to 47 teraflops. These resources are distributed over several locations belonging to the Extreme Science and Engineering Discovery Environment (XSEDE) project⁷⁴.

⁷²<https://portal.futuresystems.org>, last accessed 2016

⁷³<http://futuregrid.github.io/manual/hardware.html>, last accessed 2016

⁷⁴<https://www.xsede.org/>, last accessed 2016

The experiments described in this section were executed using the India⁷⁵ computing resource, an IBM iDataplex systems with 1024 cores and 3072 GB of RAM memory⁷⁶ running the Havana⁷⁷ OpenStack release.

FutureGrid allows testing our solution under a private academic cloud scenario, which represents a common case for scientists executing experiments within the context of their organizations.

6.1.2 Amazon Elastic Cloud Computing

As introduced in Chapter 2, Amazon established the de facto standard for cloud computing providers in the mid 2000s. Amazon Elastic Compute Cloud (Amazon EC2) is one of the services that compose the Amazon cloud platform. It provides scalable and configurable resources, allowing developers to create, launch, and terminate computing instances as needed, under an hourly pricing policy.

Amazon resources are highly distributed over different geographical locations, being the user able to select the region where the instances are going to be deployed. This is a key feature for scientists that have restrictions on where to execute their experiments, as sometimes imposed by funding agencies.

The popularity of this platform has made others adopt the Amazon EC2 specification, extending the range of available solutions by defining compliant APIs. By including Amazon EC2 into our experimentation process, we are able to evaluate our approach in a public facility, as well as to allow the integration of local EC2-compliant infrastructures. Estimating the amount of resources provided by Amazon and their computational capacity is not straightforward. While in theory the number of resources that Amazon offers is infinite, scaling according to the user needs, some restrictions on the amount of them that can be allocated at the same time may hold.

6.1.3 Vagrant & VirtualBox

In Section 5.3.2.2 we described how our system generates enactment files using the Vagrant syntax for deploying local execution environments. These files can be executed

⁷⁵<https://kb.iu.edu/d/bbms>, last accessed 2016

⁷⁶<http://futuregrid.github.io/manual/hardware.html#t-clusters>, last accessed 2016

⁷⁷<https://www.openstack.org/software/havana/>, last accessed 2016

using the tools provided by Vagrant to create and configure virtual machines on the selected compliant providers, such as VirtualBox or VMWare.

In this work we have executed the selected experiments in an Ubuntu 12.04.5 based machine, with a 64-bits CPU and 4 cores, running at 2 GHz speed with 8 Gb RAM memory. We have selected VirtualBox as the virtualization solution in this evaluation process, more specifically the 4.3.24 version.

6.1.4 Computational Appliances

Based on these three infrastructure providers, we have generated a set of annotations describing the different computational resources belonging to them that are available in this experimentation process. We have defined two different virtual appliances, both using Linux-based distributions. As summarized in tables 6.1 and 6.2, our appliances are based on the CentOS and Ubuntu distributions respectively. For each of them we also described several image appliances, oriented for the different hardware requirements exposed by the different workflows we have evaluated. We annotated a set of eight virtual machine images with different RAM, disk and processing capacities, which are grouped into the CENTOS SVA and UBUNTU SVA appliances, depending on the operating system they expose.

The description of the appliances is included in the Scientific Virtual Appliances catalog that we have used in this experimentation process, so the ISA algorithm is able to select them consequently. According to the SVA model, an SVA is supported by one or more Image Appliances, which represent the set of VMs that are able to enact the characteristics specified for the SVA, but with different hardware characteristics. Thus, the ISA will be able to filter those that do not meet the requirements specified by the workflow being reproduced.

In this experimentation process we have described and consumed public images, which are available in the different repositories of the providers studied. They have been used as they are, with the only exception of the Amazon ones, which were modified to enable root login with cloud-init⁷⁸, as required by PRECIP.

The WICUS system allows specifying the list of providers that are available by means of a configuration parameter of the ISA. This way it is possible to define the

⁷⁸<https://cloudinit.readthedocs.org>, last accessed 2016

	CENTOS SVA					
Provider	Amazon EC2			FutureGrid		Vagrant
Image App.	M1L	M1L40GB	M3XL40GB	M1L	M1L40GB	CVG
RAM (GB)	7	7	15	8	8	3
DISK (GB)	5	40	40	5	40	50
CPU (GHz)	2.4	2.4	2.4	2.9	2.9	2.83
CPU Arch.	64bits	64bits	64bits	64bits	64bits	64bits
OS	CentOS 6.5					

Table 6.1: CentOS 6 Appliances

	UBUNTU SVA	
SVA	Amazon EC2	Vagrant
Image App.	UM1L	UBG
RAM (GB)	7	2
DISK (GB)	40	40
CPU (GHz)	2.4	2.4
CPU Arch.	64bits	64bits
OS	Ubuntu 12.04	

Table 6.2: Ubuntu 12 Appliances

spectrum of resources to be selected, thus allowing to test the different providers, as we will do in this experimentation process.

6.2 Computational Experiments

In order to validate our reproducibility approach and how it can be applied in the context of scientific workflows we have selected a representative sample of workflows and WMSs. The criteria for selecting them was based upon the availability of the experimental materials, so they could be reproduced. These materials include the data analyzed by the workflows, which even though is out of the scope of our approach, is necessary to fully reproduce a workflow. Besides, the workflow specification itself is necessary to test our approach, in order to extract the information about the workflow, as stated in Section 5.1.

These complementary materials have to be defined for evaluating our approach, but more importantly, the computational dependencies of the workflow must be available. Either provided by the workflow developer or hosted in a public domain repository, our approach requires them to be accessible in order to be deployed and configured.

Furthermore, support from the corresponding communities and workflow developers was a key aspect when selecting the workflows for this evaluation process. Access to the specific knowledge about the underlying environment for executing an experiment is mandatory when trying to reproduce it. In our approach we aim to make this knowledge explicit in a way that it can be shared and understood. Thus, we have also consider the possibility of contacting workflow designers as a relevant criterion, which holds for all the workflows exposed below.

6.2.1 Pegasus

As introduced in Section 2.2.2.6, Pegasus is a well known WMS able to manage workflows comprised of millions of tasks, recording data about the execution and intermediate results. In Pegasus, workflows are described as abstract workflows, which do not contain resource information, or the physical locations of data and executables. The abstract workflow description is represented as a DAX (DAG in XML), capturing all the tasks that perform computation, the execution order of these tasks, and for each task the required inputs, expected outputs, and the arguments with which the task

Figure 6.1: Pegasus WMS software dependency graph.

should be invoked. During the execution of a workflow, the Pegasus WMS translates an abstract workflow into an executable workflow, determining the executables, data, and computational resources required for the execution.

Pegasus maps executables to their installation paths or to a repository of stageable binaries defined in a Transformation Catalog (TC), which represents the software registry of the WICUS system explained in Section 5.1.5.

A workflow execution includes data management, monitoring, and failure handling. Individual workflow tasks are managed by a task scheduler (HTCondor (Thain et al., 2005)), which supervises their execution on local and remote resources. Our configuration of Pegasus also requires Java for being executed and SSH for data transferring, as depicted in Figure 6.1. As part of the installation process of Pegasus and HTCondor, the *wget*⁷⁹ command is used, so it has been also included as a dependency in the description of the WMS.

Three different workflows designed for the Pegasus WMS have been selected for our experimentation process, covering the astronomy and genomic domains. All of them have been analyzed and reproduced alongside with the Pegasus team at the Information Science Institute⁸⁰.

6.2.1.1 Montage

The Montage workflow (Berriman et al., 2004) was created by the NASA Infrared Processing and Analysis Center (IPAC) as an open source toolkit that can be used

⁷⁹<http://www.gnu.org/software/wget/>, last accessed 2016

⁸⁰<http://pegasus.isi.edu/>

to generate custom mosaics of astronomical images in the Flexible Image Transport System (FITS) format. In a Montage workflow, the geometry of the output mosaic is calculated from the input images. The inputs are then re-projected to have the same spatial scale and rotation, the background emissions in the images are corrected to have a uniform level, and the re-projected, corrected images are co-added to form the output mosaic.

Figure 6.2 illustrates a small Montage workflow, depicting the different transformations performed in it, being each node related to a different binary software component. Therefore, in this experiment, 9 Montage software components were identified, which are linked to the generated requirements. Even though they were not specified in the definition of the workflow, another two components were also detected and included as dependencies in the software catalog: *mDiffFit* depends on the *mDiff* and *mFitPlane* components, as depicted in figure 6.7.

The size of the workflow depends on the number of images required to construct the desired output mosaic. In this work we have studied a small workflow, analyzing a 0.1 degree image of the sky. To test the correctness of the process we executed the Montage workflow in a predefined VM image, where the execution environment was already in place. Then we executed the ISA over the generated annotations to obtain a new execution environment, in which the workflow was invoked with the same input data. In order to verify that the results were equivalent in the different environments studied in this work, we conducted different experiments configuring the ISA to select different providers.

Results show that the execution environments deployed in all providers are equivalent to the former one. In addition, we used a perceptual hash tool⁸¹ to compare the resulting image (0.1 degree image of the sky) generated by both executions against the one generated by the baseline execution. We obtained a similarity factor of 1.0 (over 1.0) with a threshold of 0.85, which means the images can be considered identical. Therefore we are obtaining the same results as the original workflow.

The materials supporting this experiment are publicly available, allowing its reproduction⁸².

⁸¹pHash - <http://www.phash.org>, last accessed 2016

⁸²<http://w3id.org/idafensp/ro/wicuspegasusmontage>

Figure 6.2: Montage workflow diagram.

6.2.1.2 An Epigenomics workflow

The USC Epigenome Center⁸³ is involved in mapping the epigenetic state of human cells on a genome-wide scale. An Epigenomics workflow (Figure 6.3) processes multiple sets of genome sequences in parallel. These sequences are split into subsets, the subsets are filtered to remove contaminants, reformatted, and then mapped to a reference genome. The mapped sequences are finally merged and indexed for later analysis. In this work, the Epigenomics workflow that we selected was used to align genome sequence reads to a reference genome for human chromosome 21. The size of the workflow depends on the chunking factor used on the input data, what determines the number of sequence reads in each chunk.

This Epigenomics workflow is mainly implemented using Perl⁸⁴ and thus, most of its software dependencies are defined by Perl scripts which depend on the Perl interpreter to execute them. These dependencies are depicted in Figure 6.4, which contains a summarized view of the dependency graph of the workflow. The central node of the graph

⁸³<http://epigenome.usc.edu/>, last accessed 2016

⁸⁴<https://www.perl.org/>, last accessed 2016

Figure 6.3: Epigenomics workflow.

corresponds to the Perl interpreter, which is a common dependency of all the scripts depicted around it. The outer nodes define second level dependencies, corresponding to binaries included on the Epigenomics software distribution and system libraries. The first ones were not included as part of the workflow definition, even if they were declared in the corresponding Pegasus Transformation Catalog. The second ones were identified by analyzing the execution of the workflow, as they were not stated in any of the workflow complementary materials.

We executed the Epigenomics workflow over the different reproduced environments, corresponding to the three different providers studied in this evaluation. All the executions were successfully performed, being the workflow able to finish and produce results. To check that those results were correct we compared them with the results obtained by executing Epigenomics in a baseline machine with every component pre-installed on it. In contrast with the previous case, the output data generated by this workflow is non-deterministic, due to the existence of probabilistic steps. Thus, the use of a hash method is unfeasible. Hence, we validated the correct execution of the workflow by checking that correct output files were actually produced, and that the standard errors produced by the applications did not contain any error message. In all cases the results obtained in each infrastructure were equivalent in terms of their size (e.g., number of lines) and content.

Figure 6.4: Epigenomics workflow software dependency graph.

The materials related to this experiment, including the semantic description and the enactment scripts have been as well published online⁸⁵.

6.2.1.3 SoyKB

The SoyKB (Soybean Knowledge Base) workflow (Joshi et al., 2014a,b) is a genomics pipeline that re-sequences soybean germplasm lines selected for desirable traits such as oil, protein, soybean cyst nematode resistance, stress resistance, and root system architecture. The workflow (Figure 6.5) implements a Single Nucleotide Polymorphism (SNP) and insertion/deletion (indel) identification and analysis pipeline using the GATK haplotype caller⁸⁶ and a soybean reference genome.

The workflow analyzes samples in parallel to align them to the reference genome, to de-duplicate the data, to identify indels and SNPs, and to merge and filter the results.

⁸⁵<http://w3id.org/idaferisp/ro/wicuspegasusepigenomics>

⁸⁶<https://www.broadinstitute.org/gatk>, last accessed 2016

Figure 6.5: SoyKB workflow.

The results are then used for genome-wide association studies (GWAS) and genotype to phenotype analysis. The workflow instance used in this evaluation process is based on a sample dataset that requires less memory than a full-scale production workflow, however it carries out the same process and requires the same software components.

This set of components is comprised by four software wrappers that invoke the different components contained in a software bundle, which groups several software artifacts, mainly encoded as Java JAR files. There are two possible options to describe this scenario. On the one hand, we may generate a description for each one of the several components included in the bundle, describing their location and properties. On the other hand, the bundle may be described as one single entity, defining where it must be deployed and unpackaged. Even when both options are correct, we selected the second one, as the bundle is distributed as such. Figure 6.6 depicts the dependency relations as defined in the SoyKB dependency graph. Direct dependencies are depicted at the bottom of the figure, stating dependencies to the bundle component, which, in turn, depends on Java.

As in the previous case, we executed an instance of the SoyKB workflow on the three

Figure 6.6: SoyKB workflow software dependency graph.

providers studied in this work, obtaining a successful execution on all of them. As well as when studying the Epigenomics workflow results, we manually verified that results were correct. We checked that no errors were produced during the execution and that the obtained results were equivalent, manually comparing their size and structure.

The SoyKB materials have been as well published online as a RO⁸⁷.

6.2.2 dispel4py

The dispel4py (Filgueira et al., 2014) WMS is a Python-based system in the form of a software library implementing the DISPEL (Atkinson et al., 2012) language. While DISPEL workflows and compositions are defined in an abstract way, dispel4py also defines the required executables as Python objects. It allows the definition of *Processing Elements*, transformation components that instantiate input and output connections for data consumption and generation.

These elements, which represent abstract activities that can be customized to perform certain data transformations, are connected through the input and output ports to define the workflow graph.

dispel4py provides an enactment engine able to map and deploy abstract workflows using different parallel processing system, such as MPI (Forum, 1994) platforms, Apache

⁸⁷<http://w3id.org/idaensp/ro/wicuspegasussoykb>

Figure 6.7: Overview of the generated annotations for the Montage, Epigenomics, and SoyKB workflows using the WICUS ontology network (yellow rectangles represent the workflow component; blue squashed rectangles represent the Montage workflow; green bevelled rectangles represent the Epigenomics workflows; and red hexagons represent the SoyKB workflow).

Storm⁸⁸ and Multiprocessing DCIs. It also provides data provenance and monitoring functionalities, as well as an information registry, which can be accessed in order to ensure consistency between runs and improve user collaboration.

As the specification of a dispel4py workflow is written in Python, we have to study the corresponding Python files, detecting the dispel4py primitives in order to define the workflow steps. In order to identify the dependencies of these workflows, it is necessary to analyze the header section of those files, which include the imported modules and libraries required by the interpreter to execute the process.

The WMS itself is composed by a Python package that can be deployed using *pip*⁸⁹, the installation tool recommended by the Python Packaging Authority⁹⁰ working group. As depicted in Figure 6.8, both dispel4py and *pip* require Python for being installed.

⁸⁸<https://storm.apache.org/>, last accessed 2016

⁸⁹<https://pypi.python.org/pypi/pip>, last accessed 2016

⁹⁰<https://www.pypa.io>, last accessed 2016

Figure 6.8: dispel4py WMS software dependency graph.

Figure 6.9: dispel4py version of the Internal Extinction of Galaxies workflow

In the following sections we will describe how we reproduced two different dispel4py workflows belonging to the astrophysics and seismology scientific domains.

6.2.2.1 Internal Extinction of Galaxies

Taking as entry point a Virtual Observatory (VO) service, published under the International Virtual Observatory Alliance⁹¹ (IVOA) specification, the *int_ext* workflow (Filgueira et al., 2015) calculates the internal extinction factor of galaxies. Using the galaxy catalog provided by AMIGA⁹², this workflow, which has been also implemented using Taverna, calculates this property, which represents the dust extinction within a galaxy and is used as a correction factor for analyzing its optical luminosity.

Figure 6.9 shows the main structure of the workflow, developed as a pipeline composed by four different PEs. The process starts by reading the input coordinates containing the right ascension and declination values of 1051 galaxies. Then, a VO service is queried for each of these galaxies using the ascension and declination values. The results of these queries are filtered in the third step, removing those that do not correspond to the morphological type and apparent flattening features of the studied galaxies. Finally, the last PE calculates the internal extinction value of each galaxy.

⁹¹<http://www.ivoa.es/>, last accessed 2016

⁹²<http://amiga.iaa.es/>, last accessed 2016

Figure 6.10: Internal Extinction of Galaxies workflow software dependency graph.

Two main modules are required by the workflow in order to carry out this process. Both are Python modules that are imported by the main file in order to retrieve the information about the galaxies and process it. This workflow uses `requests`⁹³ and `astropy`⁹⁴, which are used for downloading and parsing the VO tables.

The installation of both modules is carried out using *pip*, thus, both packages depend on it, as depicted in Figure 6.10. They also depend on Python (version 2.7.3) and its related developers package for building Python modules.

The execution of the workflow in the reproduced environments, defined and enacted by the WICUS system, produced the same results as generated by the original workflow. Even when these results were equal, an exact match might have not occurred, as the input data is retrieved from an online repository. Thus, the obtained results are subject to potential changes and updates on this data. As the data provided by the AMIGA catalog is mature and stable enough, we were able to obtain the exact same results in this experimentation process.

The materials and instructions for reproducing this experiment are available online⁹⁵.

6.2.2.2 Seismic Ambient Noise Cross-Correlation

The Seismic Ambient Noise Cross-Correlation workflow (also referred as *xcorr*) (Figuera et al., 2014) was developed as part of the VERCE project⁹⁶ for processing and cross-correlating time series (traces) from seismic stations. These traces include many geophysical properties, such as wave velocities or event rates, which have to be processed in nearly real-time for hazard forecasting.

⁹³<http://www.python-requests.org/en/latest/>

⁹⁴<http://www.astropy.org/>, last accessed 2016

⁹⁵<http://w3id.org/idafensp/ro/wicusdispel4pyastro>

⁹⁶<http://www.verce.eu/>, last accessed 2016

Figure 6.11: Seismic Ambient Noise Cross-Correlation workflow.

xcorr can be divided into two main phases, namely preprocess and cross-correlation. The first one applies a set of treatments to the different traces from a given station. The second stage of the process pairs the seismic stations and calculates the cross-correlation of those pairs. Five main PEs define this workflow, four simple ones and another one composite, as depicted in Figure 6.11. The first and last PEs only perform read and write operations, whereas the main phases are implemented by the second and fourth ones.

This implementation relies on Obspy (Beyreuther et al., 2010), a Python framework for processing seismological data that provides a set of parsers, clients and processing tools for seismology analysis routines of time series. This is the only identified dependency of this workflow.

To evaluate it we performed an execution of the workflow using the data from seven different stations. In this case, the input data consumed in both, the baseline execution and the ones executed on the reproduced environments, is static and therefore the obtained results can be compared directly. The output of the *xcorr* is written to a file containing the cross-correlation calculations between a pair of stations. In all executions the results were equal, thus guaranteeing the correct reproduction of the execution environments.

The experimental materials for obtaining a new execution environment and executing the *xcorr* workflow can be accessed online⁹⁷.

⁹⁷<http://w3id.org/idaensp/ro/wicusdispel4pyxcorr>

6.2.3 Makeflow

Makeflow is a WMS where workflows are defined and implemented in a very similar way to the Make⁹⁸ files syntax. In Makeflow, transformations are defined by a set of input files to be consumed, the set of files to be generated, and the executable file or command that is in charge of carrying out the transformation. From small pipelines grouping a few commands to more complex applications containing hundreds to thousands of tasks, Makeflow implements a very flexible syntax that allow to defining workflows in a portable way.

This WMS is distributed as part of the Cooperative Computing Lab (CCL) software suite⁹⁹, a collection of tools for large scale distributed computing on DCIs such as grids, clusters and clouds. These tools, including Makeflow, are designed for science and engineering tasks and include computational abstractions, virtual file and storage systems. CCL tools are available as a bundle, which is the software artifact we will use for deploying this WMS. External tools, such as HTC Condor, SGE and Torque can be used alongside Makeflow for task management. In this work, however, we have used Work Queue¹⁰⁰, the built-in solution provided by the WMS and distributed with it, which fits our necessities for executing the Makeflow workflows studied in this work.

In this syntax, both data files and executable files components are mixed together at the same level in the workflow definition. As no specific indications about whether a file is an executable or a regular file, we have to manually classify them in order to identify software dependencies of Makeflow workflows. We define a pre-filtering stage, where a user extracts the files representing libraries, commands, scripts or binary files. These elements are the input of the software annotation tool of the WICUS system described in Section 5.1.6.

6.2.3.1 BLAST

The Basic Local Alignment Search Tool (BLAST) (Altschul et al., 2011) algorithm is a bioinformatics implementation for comparing sequence information. Different types of genetic information, such as nucleotides or amino-acids sequences, can be queried using a database to find commonalities between several sequences.

⁹⁸<https://www.gnu.org/software/make/>, last accessed 2016

⁹⁹<http://ccl.cse.nd.edu/software/>, last accessed 2016

¹⁰⁰<http://ccl.cse.nd.edu/software/workqueue/>, last accessed 2016

Figure 6.12: Makeflow version of the BLAST workflow.

This algorithm, and the different implementations that have been developed over the last decades, have improved the performance of sequence searching process. The use of heuristics methods for avoiding to compare sequences entirely, increases the processing speed and makes this problem affordable when dealing with large amounts of data.

Several workflows for different WMS have been designed to implement the BLAST algorithm, including the version studied in this work. The Makeflow version of the BLAST algorithm, depicted in Figure 6.12, is composed by four steps, being the second one the main processing activity of the sequence searching process. This workflow relies on a set of Perl and Bash¹⁰¹ scripts that invoke other binaries and commands.

As depicted in Figure 6.13, four main dependencies are required by the BLAST workflow, two Perl scripts and two Bash ones. Each of the Perl scripts depend on a binary file which are part of the BLAST distribution, as well as on the Perl interpreter.

The execution of the workflow on the reproduced environments produced the same results as the ones obtained in a baseline environment. The input data to be analyzed in both cases is the same, a sample gene database with more than 18Gb of data. As the

¹⁰¹<https://www.gnu.org/software/bash/>, last accessed 2016

Figure 6.13: BLAST workflow software dependency graph.

data consumed, and thus retrieved and stored, and the data generated are high space consuming, the hardware requirements of this workflow were specified to state a large disk and RAM memory capacity.

To reproduce the execution environment and executing the Makeflow BLAST workflow we provide an RO with the required materials¹⁰².

6.3 Results discussion

For all the experiments introduced above, we went through an analytical process, in which we tried to extract the underlying knowledge about the necessary computational assets, in order to properly describe them. This process includes gathering information from the corresponding documentation files or sites when available, studying the different files that are distributed, and usually contacting the workflow developers and the associated scientific communities. In all the workflows evaluated in this work we had the support from the staff related to these workflows and systems.

An exploratory process like this works in a trial and error fashion, as many times identifying dependencies and their proper configuration requires the workflow to be executed to analyze where and why errors occur. This iterative process is the first stage of understanding which are the required annotations to be generated. Based on this experience, the input files of the process are complemented and refined to include those

¹⁰²<http://w3id.org/idaferisp/ro/wicusemakeflowblast>

elements that are not correctly defined or that were not included. Usually, a new WMS or workflow can be studied and documented in a period of one to two weeks.

The workflows included in this work have been all analyzed after they were developed and published. Some of them are more recent, such as the ones from the `dispel4py` WMS, developed less than a year before the time of this work, while other were originally published several years before, such as the case of the Pegasus Montage workflow, developed originally in 2003. Ideally, scientists or workflow developers would use the WICUS system to generate the documentation of their experiments while publishing them, so the information provided would come straight from the adequate source and could be verified seamlessly during the development process.

In this section we have shown how each of the workflows, and their related systems, evaluated in our work have been studied and analyzed, in order to achieve their reproducibility. Based on the results obtained from this experimentation process, and how they relate to the original ones, we have verified that the generated environments were equivalent to the former infrastructures. Thus, the conservation and reproduction of those experiments can be considered as achieved under the conditions stated in this work.

As introduced on the previous sections, we have published online the experimental materials associated to the different experiments, following the Research Object (RO) specification. Alongside with these materials, instructions and examples on how to execute the workflows studied in this section are provided in the corresponding ROs introduced for each experiment of the evaluation.

Chapter 7

Conclusions and Future Work

“We can only see a short distance ahead, but we can see plenty there that needs to be done.”

Alan Turing, *Computing Machinery and Intelligence*, 1950

With the contributions described in this thesis we can claim having achieved our initial objectives. We have validated the overall hypothesis, stating that it is possible to describe the execution environment of a computational experiment in terms of its properties, and use this description to carry out a reproduction process able to generate an equivalent computational environment by means of virtualization techniques.

Based on our experimentation process performed, as well as on the development process of this thesis, we can identify several outcomes, in the form of conclusions, that summarize the main ideas we have extracted. Besides, we spot several points that are open to discussion, and which we will endorse in this chapter. We will finally introduce which are the next steps related to this work that may be addressed in the near future.

7.1 Conclusions

Two main contributions have been introduced in this work in order to achieve the conservation and reproducibility defined goals. We first introduced the set of models, implemented as an ontology network, for representing the properties of execution environments, which focus on achieving their conservation. Then we introduced the

WICUS framework, where these ontologies are integrated, for obtaining a reproduced infrastructure of a given scientific workflow.

7.1.1 Goals Achievement

Each of the goals defined in Section 3.3 has been successfully fulfilled by the contributions of this work, as well as verified by the experimentation process we have conducted. These goals validate, in turn, the different hypotheses stated in Section 3.2. Below we discuss how each of the goals has been achieved:

Goal 1 *Create a model able to conceptualize the set of relevant capabilities that describe a Computational Infrastructure.*

We have introduced in this work a set of semantic models, arranged into an ontology network, to describe the different conceptual domains related to the execution of a scientific process. Using the WICUS ontology network, detailed in Chapter 4, we are able to describe the requirements of a workflow, relating each one to the corresponding fragment of the workflow, in terms of their associated software components, as well as the hardware specifications required by the workflow. It also defines means for documenting the computational resources that are available to the user aiming to reproduce an experiment, and that serve as the basis for building the reproduced environment.

The terms and relations of the WICUS ontologies offer a structured way of stating the capabilities of the computational infrastructures, in a way they can be shared and understood, using the models and their documentation as a reference framework.

Goal 2 *Design a framework to provide means for populating these models, collecting information from the materials of a Computation Scientific Experiment and generating structured information.*

As part of the WICUS system described in Section 5.1, we have developed several tools for extracting information from the identified sources of each workflow system and transforming it into structured data. These tools use the ontologies from WICUS network to generate the set of required individuals, belonging to the different classes included in the models, and relating them to each other by means of the properties defined on them.

In Section 5.1.2 and Section 5.1.6 we introduced two parsing tools that read the workflow specification and software registry files respectively, and based on them generate the annotations describing steps of a workflow and their requirements, as well as the dependencies associated to these requirements. These tools have been developed to adapt to the different workflow system syntaxes studied in this work.

In some cases we were not able to identify an input source of information, such as for the workflow system itself or the available computational resources to be used. For those situations, we provided manually generated files, which are integrated with the aforementioned tools. As well, due to the fact that sometimes dependencies are not explicit or properly stated in the input sources, we allow manual interaction from the user annotating the catalogs, through configuration files specifying them.

These modules of the WICUS system allow populating the information repositories in a semi-automated way, generating structured and shareable datasets.

Goal 3 *Propose an algorithm that, based on the description of a former Computational Infrastructure, is able to define an equivalent infrastructure specification.*

We describe in Section 5.2 how our approach carries out the specification process, which based on the annotations generated before, obtains a new specification defining an equivalent infrastructure. The module in charge of this process is the Infrastructure Specification Algorithm (ISA), an algorithmic process described in Section 5.2.1 that consumes the descriptions of the workflow requirements and the available resources, and matches them accordingly.

Depending on the criteria that we want to prioritize, different implementations of this algorithm would be developed. The goal of the variant we introduce in this work is to minimize resources consumption and deployment effort, by combining different requirements into a single machine and avoiding the installation of already available components. As evaluated in this work, this implementation meets our requirements, being able to generate equivalent infrastructures.

The main outcome of this process is the Abstract Deployment Plan described in Section 5.2.2, that contains the specification of the infrastructure itself, in terms of the resource identifiers and how they are going to be deployed. This plan can be later translated into the corresponding syntaxes of the different enactment systems when necessary.

This process, which relies on the contributions previously described, generates the specification of an infrastructure that is equivalent in the context of the execution of a given workflow, ensuring that the required features are properly set and tuned.

Goal 4 *An integrated system able to deploy virtual machines on several Virtualized Infrastructure providers, meeting a certain hardware specifications and install and configure the proper software stack, based on the deployment plan specified by the afore-mentioned algorithms.*

As described in section 5.3, in this thesis we integrate several tools for enacting resources and configuring them, according to the specification generated in the Abstract Deployment Plan. This plan provides a generic description of the required deployment steps and their associated configuration information, which allow us to integrate different deployment tools by adding translation modules.

In this work we provide two different enactment solutions, PRECIP and Vagrant, which are described in Sections 5.3.2.1 and 5.3.2.2. These tools create computational resources in the form of virtual machines, meeting the hardware specification required by the workflow. Once these machines are created, the software components are installed on them, using a proper configuration.

As proved by our experimentation process, these tools are capable of generating an equivalent infrastructure, thus achieving this last goal.

7.1.2 Hypothesis Validation

By achieving the aforementioned goals we have validated the main hypothesis of this work. The experimentation process we have envisioned and conducted in this research, which applies them to a set of representative use cases from different areas of science, allows us to conclude that the different hypotheses stated in this work have been verified.

Hypothesis 1 *Semantic technologies are expressive enough to describe the Execution Environment of a Computational Scientific Experiments.*

We have generated semantic-based descriptions of several execution environments, each of them including different workflow systems and scientific tools. These information assets have been used as the basis of the reproduction process, proving that it is indeed

possible to describe the properties of a given execution environment in a way that they can be later reproduced.

The use of semantic technologies provides us with a reference framework for information representation, which is expressive enough to describe these environments, and allows to query it in a seamless way, regardless of which workflow system we are targeting. These insights, obtained by the achievement of goals **G1** and **G2**, verify the **H1** hypothesis introduced in Section 3.2.

Hypothesis 2 *An algorithmic process can be developed that, based on the description of the main capabilities of an Execution Environment, is able to define an equivalent infrastructure for executing the original Computational Scientific Experiment obtaining, equivalent results.*

The implementation of the algorithm described in Section 5.2.1 and its execution over the annotations describing the execution environment of the six scientific workflows reproduced in this work, proves that such algorithm is feasible. The definitions generated by the ISA have been enacted and successfully used for supporting new executions of these experiments.

As stated in Section 3.5, we define the equivalence between two infrastructures in terms of their capability of executing a scientific workflow obtaining consistent results. In this way, we have been able to obtain such results for all the experiments studied, which allows us to conclude that this contribution, defined by goal **G3**, proves the feasibility of hypothesis **H2**.

Hypothesis 3 *Virtualization techniques are capable of supporting the reproduction of an Execution Environment by creating and customizing computational resources such as Virtual Machines that fulfill the requirements of the former experiment.*

In this work we argue that virtualization, as a mature and widespread solution for providing computational resources, allows tackling the long term conservation and reproducibility of infrastructures. The milestones achieved by goal **G4** show that, by using virtualization techniques in conjunction with tools for configuration management, it is possible to create equivalent duplicates of former environments. These new infrastructures are created and configured on the fly, enacting virtual machines which are tuned to fulfill the requirements of the workflow being reproduced.

Thus, hypothesis **H3** is verified by this goal, as resources that were configured and used by the time of the first execution of the workflow, which in the case of some of the experiments happened several years ago, have been recreated and used for executing it again successfully.

7.2 Discussion and Future Work

This thesis addresses one of the dimensions of scientific reproducibility, in the context of computational experiments. As explained throughout the different chapters of this work, our approach fulfills the objectives and hypotheses envisioned at the beginning. The contributions on this thesis solve current problems in the context of scientific experiments and open, as well, new lines of work that may be considered in the future.

As any other process attaining the conservation of an object of interest, either physical or digital, an extra effort when producing scientific results is required for achieving their reproducibility. Why and how it should be done, as well as who should take care of it, is not a trivial question, as many interests conflict, and guaranteeing its success also depends on that. More and more often funding agencies are requiring the results of scientific projects to be reproducible, and scientific publication stakeholders have started to promote means for providing extra materials that ease this issue.

Encouraging authors to make their research reproducible, just by invoking their good will and moral duty towards a better science is of course advisable, but relying only on that is not sufficient. A culture on reproducibility must be grown among scientific communities, what requires significant parties of the scientific society to promote it. However, it is not clear whether or not scientists should be the ones in charge of conserving their work. If we consider classical areas of conservation, such as cultural heritage and literature, it would not be reasonable to expect authors of these kind of contributions to be also the ones providing a conservation plan.

External entities, specialized in conservation, are usually required in such cases, taking care of the study of the objects to be conserved. Nevertheless, in order to state a fair comparison, not all pieces of cultural heritage go through a conservation process, as not all of them are of interest, thus not being required to make them last forever. Whereas conservation of cultural heritage is only required for prominent objects, in the context of scientific contributions, usually most of them should be conserved in order

to allow their reproducibility. Therefore, due to the intrinsic need of scientific results to be reproducible, some involvement of authors producing these results is required.

Following this line of thought, one of the future contributions that are still open is the definition of the roles and responsibilities over the conservation and reproducibility process of scientific experiments. These responsibilities would be probably divided into the different parties and stages of the publication process. On the one hand, editors and publishing entities would provide the means for authors to identify and supply the required materials for conserving the experimental process related to a publication. On the other hand, those authors should take into account the future reproducibility as part of the design of their experiments, and include the conservation process as part of their routine.

The contributions stated in this approach would allow providing the required information, providing a reference framework that allows stating the materials for experimental conservation. As well, the tools described in this work would help authors to generate the information assets related to their experiments. In the future we aim to identify the different tasks related to the conservation and reproducibility, as well as the different participants that must be involved on them, and assign the corresponding responsibilities, producing a methodology on scientific conservation and reproducibility.

As introduced before in this thesis, we have analyzed and reproduced several representative workflows that were already developed and deployed, in some cases even several years ago. We have followed a sort of forensic approach for studying them, aiming to find evidences of which components were really required for reproducing them, avoiding those which were redundant and spotting the ones that were not directly identifiable. This process required the execution of the experiments, analyzing the causes of the failures of each invocation for refining and fixing the annotations describing them.

It is still open to study how our approach would be applied during the design, development and publication process of a computational experiment. We envision such integration as a beneficial addition, as during the development stage of a workflow, the software dependencies and hardware specifications are usually identified, thus being easier to describe them. In the near future we aim to apply our approach to a real case experiment during its development process in collaboration with computational scientists and authors. We will study the immediate reproducibility of the experimental environment as well as how it holds in the long term.

Our approach automates the deployment and configuration of new reproduced environments, based on the generated specifications, by relying on the scripting capabilities of the enactment tools we have included on our system. Even when most of the tools and resources can be managed by this kind of processes, during the development of this work we found some components that only provided installation processes based on Graphical User Interfaces (GUIs). We have not covered this type of deployment in this work, since the workflows evaluated did not contain tools using it. Defining a way of providing installation instructions in a detailed and understandable way, so they can be later installed using human interaction, is a challenging task that we will study in the future, providing natural language statements, generated as automatically as possible, to guide the process.

Another concern on scientific conservation is the period during which the experiment being considered must be able to be reproduced. It is not straightforward to define how long it must be conserved. The larger the period the harder to conserve the experiment environment will be, as more potential threats to their executability may appear. The tradeoff between the conservation effort and the sufficient reproducibility must be identified, as it makes no sense, and is even almost impossible, to make an experiment be reproducible in the very long term. In this way, we aim to study how this period can be defined, identifying the period of potential interest of the conserved experiment. Experiments of high interest for the community would be at some point reproduced, adapted or optimized, potentially going through a new conservation process that will incrementally extend their reproducibility.

In this work, we have demonstrated that an equivalent computational environment, which fulfills the requirements of a workflow, can be obtained from simple base VM images, which are already available in most cloud platforms. Consequently, there is no need to create and store a new VM image for each workflow. Therefore, this approach significantly diminishes the costs and efforts in maintaining a VM image that may have limited usage. We acknowledge that this approach may be limited to the online public availability of the software, however most of the scientific tools developed nowadays are hosted as an open-source project.

The results of this work also show how components, such as the workflow system, can be annotated once and then reused among workflows. We envision a library of workflow descriptions in which components and tools can be easily reused, even during

the development process of the workflow. Many workflows are built upon previous workflows, especially within the context of a scientific domain, and hence having such type of library would be helpful. We plan to study how and when those libraries can be built, analyzing their degree of reuse. As introduced in this work, our descriptions of the environment, generated using the WICUS ontology network, are compliant with the Linked Data principles. This would allow its future publication and integration with other information sources, which fit the idea of a distributed and structured library of descriptions that could be publicly available worldwide.

In the near future we are also willing to apply our approach to a wider range of scientific applications, including new workflows belonging to scientific areas not covered in this work. Similarly, we aim to extend it in order to support the execution of workflows over more complex and larger execution infrastructures, such as multi-node clusters and topology-specific network platforms. In order to face the long term conservation of the experiments conserved using our approach, we have to integrate the different solutions on virtualization that are appearing and that will appear in the future, such as the use of lightweight containers. We will also work on increasing the degree of automation of the semantic annotation process to describe both the workflow application and the workflow management system. Similarly, we plan to extend the ontology network to include new concepts and relations such as software variants, incompatibilities, and user policies for resource consumption.

7.3 Selected Publications

Finally, we introduce the most relevant publications generated during the development of this thesis.

- Santana-Perez, Idafen and Pérez-Hernández, María , “Towards Reproducibility in Scientific Workflows: An Infrastructure-Based Approach” *Scientific Programming*, vol. 2015, Article ID 243180, 11 pages, 2015. doi:10.1155/2015/243180.
- Doug James, et. al. (including Santana-Perez, Idafen), “Standing Together for Reproducibility in Large-Scale Computing: Report on reproducibility@XSEDE” reproducibility@XSEDE workshop, 2014.

- Santana-Perez, Idafen, Ferreira da Silva, Rafael, Rynge, Mats, Deelman, Ewa, Pérez-Henández, María, Corcho, Oscar , “A Semantic-Based Approach to Attain Reproducibility of Computational Environments in Scientific Workflows: A Case Study” 1st International Workshop on Reproducibility in Parallel Computing (REPPAR14) in conjunction with Euro-Par 2014 (August 25-29), Porto, Portugal.
- Santana-Perez, Idafen, Ferreira da Silva, Rafael, Rynge, Mats, Deelman, Ewa, Pérez-Henández, María S, Corcho, Oscar , “Leveraging Semantics to Improve Reproducibility in Scientific Workflows” reproducibility@XSEDE workshop, 2014.
- Santana-Perez, Idafen and Pérez-Hernández, María.; , “A Semantic Scheduler Architecture for Federated Hybrid Clouds” Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on , vol., no., pp.384-391, 24-29 June 2012.

ANNEX A

WICUS ORSD

WICUS Ontology Requirements Specification Document
1. Purpose
The purpose of building the WICUS ontology is to provide a consensual knowledge representation framework for describing the domain of the execution environment involved in the execution of computational experiments, to be used for reproducing them.
2. Scope
This ontology has to focus on the reproducibility aspects related to the execution environment of a computational experiment and how it relates to the workflow defining its process. The level of granularity is directly related to the competency questions and terms identified.
3. Implementation Language
The ontology will be implemented using the OWL language
4. Intended End-Users
User 1. Computational scientist aiming to make their experiment reproducible. User 2. Computational scientist reproducing an experiment. User 3. Software agent carrying out a process for defining a new infrastructure specification.
5. Intended Uses
Use 1. Describing the software and hardware execution requirement of a computational experiment. Use 2. Understanding how the original infrastructure was set up. Use 3. Retrieving the necessary information for deploying the software components properly. Use 4. Retrieving the necessary information for enacting the required computational nodes.

Table A.1: WICUS ORSD Slots 1 to 5

6. Ontology Requirements
a. Non-Functional Requirements
<p>NFR 1. The ontology must follow a <i>Camel Case</i> convention.</p> <p>NFR 2. The ontology be documented and support dereferenceable URIs that point either to its documentation (in case of a human asking for it) or RDF information (in the case of a software agent).</p> <p>NFR 3. The ontology must be written in english, both for the URIs and the ontology annotations.</p> <p>NFR 4. The ontology must be published under a open and non-comercial license.</p>
b. Functional Requirements: Groups of Competency Questions
<p>CQ1. What is the most common form of modeling a computational scientific experiment? A scientific workflow.</p> <p>CQ2. How dependencies are formalized when describing a scientific workflow? By defining its execution requirements.</p> <p>CQ3. What kind of components are required for executing a workflow, from the point of view of its infrastructure? Software components and hardware specification.</p> <p>CQ4. How a workflow is defined? By a set of steps and relations between them.</p> <p>CQ5. How a software requirement is defined? By its software components and their dependencies, both software and hardware.</p> <p>CQ6. How a hardware requirement is defined? By the specification of the components required, including their features.</p> <p>CQ7. Which elements are required for automatically installing a piece of software? Its configuration information and its installation script.</p> <p>CQ8. How can a user enact a hardware specification? Using virtualization.</p> <p>CQ9. Which kind of hardware elements are involved on a workflow execution? CPU, Memory, Network, Storage.</p> <p>CQ10. How is a hardware component defined? By its inner features, defining the value and unit of each one.</p> <p>CQ11. How are software components defined? By its binary file, its configuration info, version, and its dependencies.</p> <p>CQ12. What defines the configuration information of a software component? The set of configuration parameters that compose it.</p> <p>CQ13. What defines a configuration parameter? A pair composed by a name and a value.</p> <p>CQ14. What defines which software components can be used? Its licence.</p> <p>CQ15. Which types of licenses can a software component define? Open and non-open ones.</p> <p>CQ16. How a requirement relates to a workflow? It can be related to a single step, to the whole workflow, or to a subworkflow of it.</p> <p>CQ17. How a software component is installed? By following a deployment plan.</p> <p>CQ18. How the configuration information of a software component is involved on its deployment? It relates to the different steps of the deployment plan.</p> <p>CQ19. How can a user enact a resource using virtualization? By accessing a cloud computing provider.</p> <p>CQ20. Which elements define the resources provided by a cloud computing provider? The virtual machine images of their catalogue.</p> <p>CQ21. How those virtual machine images are identified? By their public virtual machine images id.</p>

Table A.2: WICUS ORSD Requirements

7. Pre-Glossary of Terms			
a. Terms from Competency Questions + Frequency			
software 8 software component 6 workflow 5 hardware 4 configuration information 2 elements 2 cloud 1	components 1 configuration parameter 1 dependencies 1 deployment 1 experiment 1 hardware component 1	hardware requirement 1 hardware specification 1 infrastructure 1 licenses 1 machine images 1 provider 1 requirement 1	resource 1 resources 1 software requirement 1 virtualization 1 workflow execution 1
b. Terms from Answers + Frequency			
licence 3 step 3 dependencies 2 deployment plan 2 features 2 hardware specification 2 software components 2	value 2 workflow 2 cloud 1 c. information 1 c. parameters 1 version 1 ex. requirements 1	hardware 1 installation script 1 machine images 1 machine images id 1 name 1 non-open 1 Open 1	provider 1 relations 1 software 1 subworkflow 1 unit 1 virtualization 1
c. Objects			
No objects were identified.			

Table A.3: WICUS ORSD Pre-Glossary of terms and objects.

ANNEX B

Terminology

Some extra concepts, related to the conservation and reproducibility aspects of this work are included below, complementing the list provided in Section 1.2

- **Scientific Domain:** also referred as scientific area, can be defined as the set of scientific practices belonging to a certain branch of science. Practitioners of a scientific domain share the same goals and tend to accept and use the same procedures for solving scientific problems.
- **Computational Science:** the use of computational procedures, such as mathematical model and simulations, to conduct processes for analyzing and solving scientific problems.
- **Computational Experiment:** the process and set of elements required for evaluating the validity of a given hypothesis by means of computational resources.
- **Scientific Workflow:** a precise and executable description of a scientific procedure designed for carrying out the necessary activities for solving a certain problem, in the context of an experiment.
- **Scientific Infrastructure:** the set of elements required for carrying out an experiment, which includes the tools involved on the experimentation process, the individuals used as part of it, and the experimental conditions in which it is performed.

- Execution Environment: in the context of computational science, the equipment involved on performing an experiment is defined by its execution environment, which is defined by the set of software components and hardware specifications that support the overall process.
- Dependency: any element required by a scientific workflow in order to be executed, including data assets, software components, and hardware resources. In the context of this work the word dependency is mainly use to denote software dependencies.
- Software Components: any piece of executable software, such as scripts and binary files, able to compute, process, generate and/or transform data. Software components are usually grouped into stacks that pile conceptually related components belonging to different levels of abstraction and that rely on each other.

Bibliography

- Afgan, E., Baker, D., Coraor, N., Chapman, B., Nekrutenko, A., and Taylor, J. (2010). Galaxy cloudman: delivering cloud compute clusters. *BMC Bioinformatics*, pages 4–4. 32
- Albrecht, M., Donnelly, P., Bui, P., and Thain, D. (2012). Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, SWEET '12, pages 1:1–1:13, New York, NY, USA. ACM. 36
- Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. (2011). Basic local alignment search tool. *IJournal of Molecular Biology*, 215(3):403–410. 116
- Atkinson, M., Liew, C., Galea, M., Martin, P., Krause, A., Mouat, A., Corcho, O., and Snelling, D. (2012). Data-intensive architecture for scientific knowledge discovery. *Distributed and Parallel Databases*, 30(5-6):307–324. 111
- Azarnoosh, S., Rynge, M., Juve, G., Deelman, E., Niec, M., Malawski, M., and Ferreira da Silva, R. (2013). Introducing PRECIP: an API for managing repeatable experiments in the cloud. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 2 of *CloudCom*, pages 19–26. 94, 96
- Baggerly, A. and Coombes, K. R. (2010). Deriving chemosensitivity from cell lines: Forensic bioinformatics and reproducible research in high-throughput biology. *Annals Applied Statistics*, pages 1309–1334. 2
- Barker, A. and Van Hemert, J. (2008). Scientific workflow: a survey and research directions. In *Proceedings of the 7th international conference on Parallel processing*

- and applied mathematics*, PPAM'07, pages 746–753, Berlin, Heidelberg. Springer-Verlag. 38, 39
- Beckman, P. H. (2005). Building the teragrid. In *A 363 (2005) 1715D 1728*, DOI: 10.1098/rsta.2005.1603. <http://www.teragrid.org>. 26
- Begley, C. G. and Ioannidis, J. P. (2015). Reproducibility in science: Improving the standard for basic and preclinical research. *Circulation Research*, 116(1):116–126. 2
- Belhajjame, K., Corcho, O., Garijo, D., Zhao, J., Missier, P., Newman, D., Palma, R., Bechhofer, S., Garcia Cuesta, E., Gomez-Perez, J. M., Klyne, G., Page, K., Roos, M., Ruiz, J. E., Soiland-Reyes, S., Verdes-Montenegro, L., De Roure, D., and Goble, C. (2012). Workflow-centric research objects: First class citizens in scholarly discourse. In *Proc. Workshop on the Semantic Publishing (SePublica)*, Proc. Workshop on the Semantic Publishing (SePublica), Crete, Greece. 8, 39
- Belhajjame, K., Goble, C., Soiland-Reyes, S., and De Roure, D. (2011). Fostering scientific workflow preservation through discovery of substitute services. In *E-Science (e-Science), 2011 IEEE 7th International Conference on*, pages 97–104. 38, 44
- Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284(5):34–43. 25
- Berriman, G. B., Deelman, E., Good, J. C., Jacob, J. C., Katz, D. S., Kesselman, C., Laity, A. C., Prince, T. A., Singh, G., and Su, M.-H. (2004). Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand. In *Astronomical Telescopes and Instrumentation*, volume 5493, pages 221–232. International Society for Optics and Photonics. 105
- Berthold, M. R., Cebron, N., Dill, F., Gabriel, T. R., Kötter, T., Meinl, T., Ohl, P., Sieb, C., Thiel, K., and Wiswedel, B. (2007). KNIME: The Konstanz Information Miner. In *Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007)*. Springer. 33
- Beyreuther, M., Barsch, R., Krischer, L., Megies, T., Behr, Y., and Wassermann, J. (2010). Obspy: A python toolbox for seismology. *Seismological Research Letters*, 81(3):530–533. 115

- Bizer, C., Heath, T., and Berners-Lee, T. (2009). Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(3):1–22. 40
- Blankenberg, D., Kuster, G. V., Coraor, N., Ananda, G., Lazarus, R., Mangan, M., Nekrutenko, A., and Taylor, J. (2010). Galaxy: a web-based genome analysis tool for experimentalists. *Current protocols in molecular biology*, pages 19–10. 37
- Bonnet, P., Manegold, S., Bjørling, M., Cao, W., Gonzalez, J., Granados, J., Hall, N., Idreos, S., Ivanova, M., Johnson, R., Koop, D., Kraska, T., Müller, R., Olteanu, D., Papotti, P., Reilly, C., Tsirogiannis, D., Yu, C., Freire, J., and Shasha, D. (2011). Repeatability and workability evaluation of sigmod 2011. *SIGMOD Rec.*, 40(2):45–48. 43
- Brammer, G. R., Crosby, R. W., Matthews, S. J., and Williams, T. L. (2011). Paper mache: Creating dynamic reproducible science. *Procedia Computer Science*, 4(0):658 – 667. Proceedings of the International Conference on Computational Science, {ICCS} 2011. 43
- Bresnahan, J., Freeman, T., LaBissoniere, D., and Keahey, K. (2011). Managing appliance launches in infrastructure clouds. In *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, TG '11, pages 12:1–12:7, New York, NY, USA. ACM. 29
- Buncic, P., Sanchez, C. A., Blomer, J., Franco, L., Harutyunian, A., Mato, P., and Yao, Y. (2010). Cernvm: a virtual software appliance for lhc applications. *Journal of Physics: Conference Series*, 219(4):042003. 30, 44
- Carpen-Amarie, A., Rougier, A., and Lbbe, F. (2014). Stepping stones to reproducible research: A study of current practices in parallel computing. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8805 of *Lecture Notes in Computer Science*, pages 499–510. Springer International Publishing. 6
- Cerf, V. G. (2011). Avoiding "bit rot": Long-term preservation of digital information [point of view]. *Proceedings of the IEEE*, 99(6):915–916. 7, 22, 41

- Chirigati, F., Shasha, D., and Freire, J. (2013). Reprozip: Using provenance to support computational reproducibility. In *Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance*, TaPP '13, pages 1:1–1:4, Berkeley, CA, USA. USENIX Association. 45
- Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. (2001). Web Service Definition Language (WSDL). 32
- Collberg, C., Proebsting, T., and Warren, A. M. (2014). Repeatability and Benefaction in Computer Systems Research: A Study and a Modest Proposal. Technical Report TR 14-04, Department of Computer Science, University of Arizona. 42
- Crick, T., Hall, B. A., Ishtiaq, S., and Takeda, K. (2014). "share and enjoy": Publishing useful and usable scientific models. *CoRR*, abs/1409.0367. 44
- De Roure, D. (2005). The semantic grid: Past, present and future. In Gomez-Perez, A. and Euzenat, J., editors, *The Semantic Web: Research and Applications*, volume 3532 of *Lecture Notes in Computer Science*, pages 726–726. Springer Berlin Heidelberg. 25
- Deelman, E., Singh, G., Su, M.-H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G. B., Good, J., Laity, A., Jacob, J. C., and Katz, D. S. (2005). Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237. 34, 37
- Drake, N. (2014). Cloud computing beckons scientists. *Nature*, 509(7502):543–544. 23, 30
- Eker, J., Janneck, J. W., Lee, E. A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., and Xiong, Y. (2003a). Taming heterogeneity - the ptolemy approach. In *Proceedings of the IEEE*, pages 127–144. 38
- Eker, J., Janneck, J. W., Lee, E. A., Liu, J., Liu, X., Ludvig, J., Sachs, S., Xiong, Y., and Neuendorffer, S. (2003b). Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144. 33
- Elsevier (2011). The executable paper grand challenge. 43

- Errington, T. M., Tan, F. E., Lomax, J., Perfito, N., Iorns, E., Gunn, W., and Evans, K. D. (2015). Reproducibility Project: Cancer Biology. osf.io/e81xl. 37
- Fahringer, T. (2011). Askalon grid environment Ñ user guide, version 1.0. 32
- Fahringer, T., Prodan, R., Duan, R., Nerieri, F., Podlipnig, S., Qin, J., Siddiqui, M., Truong, H.-L., Villazon, A., and Wieczorek, M. (2005a). Askalon: A grid application development and computing environment. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing, GRID '05*, pages 122–131, Washington, DC, USA. IEEE Computer Society. 32
- Fahringer, T., Qin, J., and Hainzer, S. (2005b). Specification of grid workflow applications with agwl: an abstract grid workflow language. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, volume 2, pages 676–685 Vol. 2. 32
- Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine. AAI9980887. 32
- Filgueira, R., Klampanos, I., Krause, A., David, M., Moreno, A., and Atkinson, M. (2014). dispel4py: A python framework for data-intensive scientific computing. In *International Workshop on Data Intensive Scalable Computing Systems (DISCS-2014)*. 111
- Filgueira, R., Krause, A., Atkinson, M., Klampanos, I., Spinuso, A., and Sanchez-Exposito, S. (2015). dispel4py: An agile framework for data-intensive escience. In *11th IEEE International Conference on e-Science*. 113
- Filguiera, R., Klampanos, I., Krause, A., David, M., Moreno, A., and Atkinson, M. (2014). Dispel4py: A python framework for data-intensive scientific computing. In *Proceedings of the 2014 International Workshop on Data Intensive Scalable Computing Systems, DISCS '14*, pages 9–16, Piscataway, NJ, USA. IEEE Press. 35, 114
- Fisher, P., Noyes, H., Kemp, S., Stevens, R., and Brass, A. (2009). A systematic strategy for the discovery of candidate genes responsible for phenotypic variation. In *Cardiovascular Genomics*, pages 329–345. Springer. 37

- Forum, M. P. (1994). Mpi: A message-passing interface standard. Technical report, Message Passing Interface Forum, Knoxville, TN, USA. 35, 111
- Foster, I. (2005). Globus toolkit version 4: Software for service-oriented systems. In *Proceedings of the 2005 IFIP International Conference on Network and Parallel Computing*, NPC'05, pages 2–13, Berlin, Heidelberg. Springer-Verlag. 25
- Foster, I. and Kesselman, C., editors (1999). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 24
- Foster, I., Kesselman, C., and Tuecke, S. (2001). The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222. 24
- Foster, I., Zhao, Y., Raicu, I., and Lu, S. (2008). Cloud Computing and Grid Computing 360-Degree Compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10. IEEE. 27
- Gaizauskas, R., Davis, N., Demetriou, G., Guod, Y., and Roberts, I. (2004). Integrating biomedical text mining services into a distributed workflow environment. In *Proceedings of UK e-Science All Hands Meeting*. 37
- Gamble, M., Goble, C., Klyne, G., and Zhao, J. (2012). Mim: A minimum information model vocabulary and framework for scientific linked data. In *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pages 1–8. 40
- Garijo, D. and Gil, Y. (2011). A new approach for publishing workflows: abstractions, standards, and linked data. In *Proceedings of the 6th workshop on Workflows in support of large-scale science*, WORKS '11, pages 47–56, New York, NY, USA. ACM. 41
- Garijo, D., Kinnings, S., Xie, L., Xie, L., Zhang, Y., Bourne, P. E., and Gil, Y. (2013). Quantifying reproducibility in computational biology: The case of the tuberculosis drugome. *PLoS ONE*, 8(11):e80278. 40, 42
- Gavish, M. and Donoho, D. (2011). A universal identifier for computational results. *Procedia Computer Science*, 4(0):637 – 647. Proceedings of the International Conference on Computational Science, {ICCS} 2011. 41

- Giardine, B., Riemer, C., Hardison, R. C., Burhans, R., Elnitski, L., Shah, P., Zhang, Y., Blankenberg, D., Albert, I., Taylor, J., Miller, W., Kent, W., and Nekrutenko, A. (2005a). Galaxy: a platform for interactive large-scale genome analysis. *Genome research*, 15(10):1451–1455. 37
- Giardine, B., Riemer, C., Hardison, R. C., Burhans, R., Shah, P., Zhang, Y., Blankenberg, D., Albert, I., Miller, W., Kent, W. J., and Nekrutenko, A. (2005b). Galaxy: A platform for interactive large-scale genome analysis. *Genome Res*, 15:1451–1455. 37
- Gil, Y., Ratnakar, V., Deelman, E., Mehta, G., and Kim, J. (2007). Wings for pegasus: Creating large-scale scientific representations of computational workflows. In *IN PROCEEDINGS OF THE 19TH ANNUAL CONFERENCE ON INNOVATIVE APPLICATIONS OF ARTIFICIAL INTELLIGENCE (IAAI)*, pages 22–26. 38
- Gil, Y., Ratnakar, V., Kim, J., Gonzalez-Calero, P. A., Groth, P., Moody, J., and Deelman, E. (2011). Wings: Intelligent workflow-based design of computational experiments. *IEEE Intelligent Systems*, 26(1). 34, 37
- Goddard, W. and Melville, S. (2004). *Research methodology: An introduction*. Juta and Company Ltd. 54
- Goecks, J., Nekrutenko, A., Taylor, J., and Team, T. G. (2010). Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biol*, 11(8):R86. 31
- Gorp, P. V. and Mazanek, S. (2011). Share: a web portal for creating and sharing executable research papers. *Procedia Computer Science*, 4(0):589 – 597. Proceedings of the International Conference on Computational Science, {ICCS} 2011. 43
- Grossman, R. L., Greenway, M., Heath, A. P., Powell, R., Suarez, R. D., Wells, W., White, K. P., Atkinson, M. P., Klampanos, I. A., Alvarez, H. L., Harvey, C., and Mambretti, J. (2012). The design of a community science cloud: The open science data cloud perspective. In *SC Companion*, pages 1051–1057. 22
- Henderson, R. L. (1995). Job scheduling under the portable batch system. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPSP ’95, pages 279–294, London, UK, UK. Springer-Verlag. 25, 36

- Hey, A. J. G., Tansley, S., and Tolle, K. M. (2009). *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research. 22
- Howe, B. (2012). Virtual appliances, cloud computing, and reproducible research. *Computing in Science Engineering*, 14(4):36–41. 44
- Ioannidis, J. P. A. (2005). Why most published research findings are false. *PLoS Med*, 2(8):e124. 2
- Ioannidis, J. P. A., Allison, D. B., Ball, C. A., Coulibaly, I., Cui, X., Culhane, A. C., Falchi, M., Furlanello, C., Game, L., Jurman, G., Mangion, J., Mehta, T., Nitzberg, M., Page, G. P., Petretto, E., and van Noort, V. (2009). Repeatability of published microarray gene expression analyses. *Nat Genet*, 41(2):149–155. 37
- Ioannidis, J. P. A., Haidich, A.-B., and Lau, J. (2001). Any casualties in the clash of randomised and observational evidence? *BMJ*, 322(7291):879–880. 2
- Jones, B. (2015). Towards the european open science cloud. 30
- Joshi, T., Fitzpatrick, M. R., Chen, S., Liu, Y., Zhang, H., Endacott, R. Z., Gaudiello, E. C., Stacey, G., Nguyen, H. T., and Xu, D. (2014a). Soybean knowledge base (soykb): a web resource for integration of soybean translational genomics and molecular breeding. *Nucleic Acids Research*, 42(D1):D1245–D1252. 109
- Joshi, T., Valliyodan, B., Khan, S. M., Liu, Y., Maldonado dos Santos, J. V., Jiao, Y., Xu, D., Nguyen, H. T., Hopkins, N., Rynge, M., and Merchant, N. (2014b). Next generation resequencing of soybean germplasm for trait discovery on xsede using pegasus workflows and iplant infrastructure. In *XSEDE 2014*. poster. 109
- Kacsuk, P. (2011). P-grade portal family for grid infrastructures. *Concurrency and Computation: Practice and Experience*, 23:235 – 245. 35
- Kacsuk, P., Farkas, Z., Kozlovsky, M., Hermann, G., Balasko, Á., Karóczkai, K., and Márton, I. (2012a). Ws-pgrade/guse generic dc gateway framework for a large variety of user communities. *Journal of Grid Computing*, 10:601 – 630. 35
- Kacsuk, P., Farkas, Z., Kozlovsky, M., Hermann, G., Balasko, A., Karóczkai, K., and Marton, I. (2012b). Ws-pgrade/guse generic dc gateway framework for a large variety of user communities. *J. Grid Comput.*, 10(4):601–630. 35

- Keahey, K. and Freeman, T. (2008). Contextualization: Providing one-click virtual clusters. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 301–308. 29
- Kinnings, S. L., Xie, L., Fung, K. H., Jackson, R. M., Xie, L., and Bourne, P. E. (2010). The mycobacterium tuberculosis drugome and its polypharmacological implications. *PLoS computational biology*, 6(11):e1000976. 40, 42
- Konig, I. R. (2011). Validation in genetic association studies. *Briefings in Bioinformatics*, 12(3):253–258. 2
- Kranzlmüller, D., de Lucas, J., and Foster, P. (2010). The european grid initiative (EGI). In Davoli, F., Meyer, N., Pugliese, R., and Zappatore, S., editors, *Remote Instrumentation and Virtual Laboratories*, pages 61–66. Springer US. 26
- Kuhn, T. S. (1962). *The structure of scientific revolutions*. University of Chicago Press, Chicago. 2
- Lankhorst, M., Proper, H., and Jonkers, H. (2010). The anatomy of the archimate language. *Int. J. Inf. Syst. Model. Des.*, 1(1):1–32. 46
- Laure, E., Gr, C., Fisher, S., Frohner, A., Kunszt, P., Krenek, A., Mulmo, O., Pacini, F., Prelz, F., White, J., Barroso, M., Buncic, P., Byrom, R., Cornwall, L., Craig, M., Meglio, A. D., Djaoui, A., Giacomini, F., Hahkala, J., Hemmer, F., Hicks, S., Edlund, A., Maraschini, A., Middleton, R., Sgaravatto, M., Steenbakkers, M., Walk, J., and Wilson, A. (2006). Programming the grid with glite. In *Computational Methods in Science and Technology*, page 2006. 25
- Litzkow, M. and Livny, M. (1990). Experience with the Condor distributed batch system. In *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, Huntsville, AL. 25, 34, 36
- Liu, D., Abdulla, G., Franklin, M., Garlick, J., and Miller, M. (2006). Data-preservation in scientific workflow middleware. In *Scientific and Statistical Database Management, 2006. 18th International Conference on*, pages 49–58. 41

- López, M. F., Gómez-Pérez, A., Sierra, J. P., and Sierra, A. P. (1999). Building a chemical ontology using methontology and the ontology design environment. *IEEE Intelligent Systems*, 14(1):37–46. 60
- Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E. A., Tao, J., and Zhao, Y. (2006). Scientific workflow management and the kepler system: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1039–1065. 33, 37
- LudÄdscher, B., Weske, M., McPhillips, T., and Bowers, S. (2009). Scientific workflows: Business as usual? In Dayal, U., Eder, J., Koehler, J., and Reijers, H., editors, *Business Process Management*, volume 5701 of *Lecture Notes in Computer Science*, pages 31–47. Springer Berlin Heidelberg. 7
- Malone, J., Brown, A., Lister, A., Ison, J., Hull, D., Parkinson, H., and Stevens, R. (2014). The Software Ontology (SWO): a resource for reproducibility in biomedical data analysis, curation and digital preservation. *Journal of Biomedical Semantics*, 5(1):25+. 46
- Manaf, N. A. A., Bechhofer, S., and Stevens, R. (2010). A survey of identifiers and labels in owl ontologies. In Sirin, E. and Clark, K., editors, *OWLED*, volume 614 of *CEUR Workshop Proceedings*. CEUR-WS.org. 60
- Mandal, A., Dasgupta, A., Kennedy, K., Mazina, M., Koelbel, C., Marin, G., Cooper, K., Mellor-Crummey, J., Liu, B., and Johnsson, L. (2004). Scheduling workflow applications in grads. In *Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid*, CCGRID 04, pages 790–797, Washington, DC, USA. IEEE Computer Society. 37
- Mao, B., Jiang, H., Wu, S., Fu, Y., and Tian, L. (2014). Read-performance optimization for deduplication-based storage systems in the cloud. *ACM Transactions on Storage (TOS)*, 10(2):6. 47
- Matthews, B., Shaon, A., Bicarregui, J., Jones, C., Woodcock, J., and Conway, E. (2009). Towards a methodology for software preservation. 43
- Mayer, R., Miksa, T., and Rauber, A. (2014). Ontologies for describing the context of scientific experiment processes. In *10th International Conference on e-Science*. 46

- Merkel, D. (2014). Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239). 96
- Mesirov, J. P. (2010). Accessible reproducible research. *Science*, 327(5964):415–416. 2
- Microsystems), W. G. S. (2001). Sun grid engine: Towards creating a compute power grid. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, CCGRID '01, pages 35–, Washington, DC, USA. IEEE Computer Society. 25, 36
- Milojicic, D., Llorente, I., and Montero, R. S. (2011). Opennebula: A cloud management tool. *Internet Computing, IEEE*, 15(2):11–14. 29
- Moreau, L., Freire, J., Futrelle, J., Mcgrath, R. E., Myers, J., and Paulson, P. (2008). The open provenance model: An overview. 41
- Noy, N. F. and McGuinness, D. L. (2001). Ontology development 101: A guide to creating your first ontology. Technical report, Knowledge Systems Laboratory Stanford University. 58, 60, 64
- Oinn, T., Greenwood, M., Addis, M., Alpdemir, M. N., Ferris, J., Glover, K., Goble, C., Goderis, A., Hull, D., Marvin, D., Li, P., Lord, P., Pocock, M. R., Senger, M., Stevens, R., Wipat, A., and Wroe, C. (2006). Taverna: lessons in creating a workflow environment for the life sciences: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1067–1100. 37, 43
- OSC (2015). Estimating the reproducibility of psychological science. *Science*, 349(6251). 2
- Palat, J. (2012). Introducing vagrant. *Linux Journal*, 2012(220):2. 94, 96
- Pérez, F. and Granger, B. E. (2007). IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29. 35
- Pham, Q. T. (2014). *A framework for reproducible computational research*. PhD thesis, The University of Chicago. 45

- Pordes, R., Petravick, D., Kramer, B., Olson, D., Livny, M., Roy, A., Avery, P., Blackburn, K., Wenaus, T., WÄijrthwein, F., Foster, I., Gardner, R., Wilde, M., Blatecky, A., McGee, J., and Quick, R. (2007). The open science grid. *Journal of Physics: Conference Series*, 78(1):012057. 26
- Qasha, R., Cała, J., and Watson, P. (2015). Towards automated workflow deployment in the cloud using toasca. In *8th IEEE International Conference on Cloud Computing, New York, USA (IEEE CLOUD)*. 46
- Romberg, M. (2002). The unicore grid infrastructure. *Sci. Program.*, 10(2):149–157. 25
- Rossum, G. (1995). Python reference manual. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands. 35
- Roure, De, D., Belhajjame, K., Missier, P., and Al., E. (2011). Towards the preservation of scientific workflows. In *Procs. of the 8th International Conference on Preservation of Digital Objects (iPRES 2011)*, Singapore. 8, 40
- Roure, D. D. and Goble, C. (2007). myexperiment - a web 2.0 virtual research environment. In *International Workshop on Virtual Research Environments and Collaborative Work Environments*. Event Dates: May 2007. 39
- Roure, D. D., Goble, C., and Stevens, R. (2007). Designing the myexperiment virtual research environment for the social sharing of workflows. In *Proceedings of the Third IEEE International Conference on e-Science and Grid Computing, E-SCIENCE '07*, pages 603–610, Washington, DC, USA. IEEE Computer Society. 43
- Roure, D. D., Goble, C., and Stevens, R. (2009). The design and realisation of the virtual research environment for social sharing of workflows. *Future Generation Computer Systems*, 25(5):561 – 567. 33
- Rusbridge, C., Burnhill, P., Ross, S., Buneman, P., Giaretta, D., Lyon, L., and Atkinson, M. (2005). The digital curation centre: a vision for digital curation. In *Local to Global Data Interoperability - Challenges and Technologies, 2005*, pages 31–41. 22
- Santana-Pérez, I. and Pérez-Hernández, M. (2015). Towards Reproducibility in Scientific Workflows: An Infrastructure-Based Approach. *Scientific Programming*, 2015. 64

- Shapin, S. and Schaffer, S. (1985). *Leviathan and the Air-Pump: Hobbes, Boyle, and the Experimental Life*. Princeton University Press,. 1
- Shiers, J. (2007). The worldwide LHC computing grid (worldwide LCG). *Computer Physics Communications*, 177(1-2):219–223. 26
- Staples, G. (2006). Torque resource manager. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA. ACM. 36
- Suárez-Figueroa, M. C., Gómez-Pérez, A., and Fernández-López, M. (2012). The neon methodology for ontology engineering. In Suárez-Figueroa, M. C., Gómez-Pérez, A., Motta, E., and Gangemi, A., editors, *Ontology Engineering in a Networked World*, pages 9–34. Springer Berlin Heidelberg. 58
- Suárez-Figueroa, M. C., Gómez-Pérez, A., and Fernández-López, M. (2015). The neon methodology framework: A scenario-based methodology for ontology development. In *Applied Ontology*, pages 1–39. IOS Press. 58
- Sure, Y., Staab, S., Studer, R., and Gmbh, O. (2003). On-to-knowledge methodology (otkm). In *Handbook on Ontologies, International Handbooks on Information Systems*, pages 117–132. Springer. 60
- Taylor, C. F., Field, D., Sansone, S.-A., Aerts, J., Apweiler, R., Ashburner, M., Ball, C. A., Binz, P.-A., Bogue, M., Booth, T., and Brazma, A. (2008). Promoting coherent minimum reporting guidelines for biological and biomedical investigations: the mibbi project. *Nat Biotech*, 26(8):889–896. 40
- Thain, D., Tannenbaum, T., and Livny, M. (2005). Distributed computing in practice: The condor experience: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4):323–356. 105
- Towns, J., Cockerill, T., Dahan, M., Foster, I., Gaither, K., Grimshaw, A., Hazlewood, V., Lathrop, S., Lifka, D., Peterson, G., Roskies, R., Scott, J., and Wilkins-Diehr, N. (2014). Xsede: Accelerating scientific discovery. *Computing in Science Engineering*, 16(5):62–74. 26

- von Laszewski, G., Fox, G., Wang, F., Younge, A., Kulshrestha, A., Pike, G., Smith, W., Vřckler, J., Figueiredo, R., Fortes, J., and Keahey, K. (2010). Design of the futuregrid experiment management framework. In *Gateway Computing Environments Workshop (GCE), 2010*, pages 1–10. 26
- Wassink, I., van der Vet, P., Wolstencroft, K., Neerincx, P., Roos, M., Rauwerda, H., and Breit, T. (2009). Analyzing scientific workflows: why workflows not only connect web services. In Zhang, L., editor, *IEEE Congress on Services 2009*, pages 314–321, Los Alamitos. IEEE Computer Society Press. 43
- Watson, J. (2008). Virtualbox: Bits and bytes masquerading as machines. *Linux J.*, 2008(166). 96
- Wells, F. (1997). Fraud and misconduct in medical research. *British Journal of Clinical Pharmacology*, 43(1):3–7. 2
- Yelick, K. and andBrent Draney andRichard Shane Canon, S. C. (2011). The magellan report on cloud computing for science. 30
- Zhao, J., Gomez-Perez, J. M., Belhajjame, K., Klyne, G., Garcia-Cuesta, E., Garrido, A., Hettne, K., Roos, M., Roure, D. D., and Goble, C. (2012). Why workflows break - understanding and combating decay in taverna workflows. *2012 IEEE 8th International Conference on E-Science*, 0:1–9. 43
- Zhao, X., Zhang, Y., Wu, Y., Chen, K., Jiang, J., and Li, K. (2014). Liquid: A scalable deduplication file system for virtual machine images. *IEEE Transactions on Parallel and Distributed Systems*, 25(5):1257–1266. 47