Build, Configuration, Integration and Testing Tools for Large Software Projects: ETICS

Marc-Elian Bégin\(^2\), Guillermo Diez-Andino Sancho\(^2\), Alberto Di Meglio\(^2\), Enrico Ferro\(^1\), Elisabeta Ronchieri\(^1,\ast\), Matteo Selm\(^1\), and Marian Żurek\(^2\)

\(^1\) INFN, Italy
\(^2\) CERN, Switzerland
elisabetha.ronchieri@cnaf.infn.it

Abstract. Software development within geographically dispersed and multi-institutional projects faces challenges in the domain of validation and quality assurance of software products. Experience in such projects, especially in the area of Grid computing, has shown that the lack of appropriate tools and procedures may cause high overall development costs and delays in the deployment, development and maintenance of the software. In this paper, we introduce ETICS, an integrated infrastructure for the automated configuration, build and testing of Grid and distributed software. The goal of the infrastructure is to provide a service for software projects by integrating well-established procedures, tools and resources in a coherent framework and adapting them to the special needs of distributed projects. A set of versatile tools and best-practice guidelines for quality assurance implementation are also provided to maximize the project’s chances of delivering reliable and interoperable software.

1 Introduction

Several large-scale open-source software projects have to deal with the need to organize complex software life cycle management infrastructures and processes in order to guarantee required levels of quality, interoperability and maintainability. Often these projects have to face resource, skill, time and budget constraints that may lead to the risk of releasing software difficult to deploy, maintain, understand and integrate with other applications. Fixed-term research projects such as DILIGENT [1], [2] and EGEE [3], [4] have to focus on developing software of increasing functionality through their lifetime, but cannot always guarantee that the software will still be accessible, maintainable and documented after the conclusion of their mandate. In such distributed development environments, ensuring that components developed by different developers, in different languages, on different platforms and with non homogeneous tools and processes is often a daunting challenge that may lead to software difficult to manage. Furthermore permanent projects such as QUATTOR [5] suffer from the lack of well-defined build procedures and this makes it difficult for other institutes to

\* Corresponding author.
adopt them. Under the pressure of short deadlines and large requirement sets, project managers may have to face the decision of cutting testing and quality assurance verifications, which can be a cause of delaying the release or impairing the usability of the software because of the excessive number of undetected problems. Even when functional tests are performed, the nature itself of complex middleware, such as that developed for the computational Grid\textsuperscript{1}, render costly the provision of adequate hardware and network resources. When middleware and applications are deployed on tests or certification testbeds a lot of time is usually spent trying to make middleware suites and applications to interoperate due to the different configuration assumptions and different versions of common libraries.

In this work, we introduce ETICS, an integrated infrastructure for the automated build, configuration, integration and testing (BCIT) of software [6], specifying its requirements and architecture. ETICS aims to support such research and development initiatives by integrating existing procedures, tools and resources in a coherent infrastructure, additionally providing an intuitive access point through a Web portal and a professionally-managed, multi-platform capability based on Grid technologies [7]. Consequently, developers and software managers will be able to integrate their code, libraries and application, validate the code against standard guidelines, run extensive automated tests and benchmarks, produce reports and improve the overall quality of the software. ETICS goal is not to develop new software but to adapt and integrate already existing capabilities, chiefly open source, providing other research projects with the possibility of focusing their efforts in their specific research field and to avoid wasting time and resources in such required, but expensive, activity. Nevertheless, ETICS also adds any missing features, such as a consistent schema in order to configure, build and test software projects with different characteristics (e.g., platforms, development languages).

This paper is organized as follows. Section 2 describes the requirements of the system, whilst Section 3 documents the related work, explaining what ETICS can add to the state of the art. Section 4 details the certification process. Section 5 describes the architecture. Section 6 presents a rigorous definition of the basic concepts, whilst Section 7 reports useful user's operations supported in the ETICS infrastructure. Few use cases are provided in Section 8. Section 9 reports the conclusion and future activities.

2 Requirements for the Design of the BCIT Framework

In this section, we describe the requirements for the design of the build, configuration, integration and testing framework for distributed software. First of all, it is fundamental to establish an international and well-managed capability for software configuration, integration, testing and benchmarking for the scientific community (for what concerns the software configuration in a complex testbed,
it is of major importance to keep the configuration as simple as possible). Second, it is also important to deploy, and if necessary to adapt, engineering tools and support infrastructures developed by other projects such as EGEE, NMI [8], LCG [9] and other open-source or industrial entities and organize them into a coherent easy-to-use set of on-line tools. The creation of a repository of libraries is also a requirement in order to allow the ETICS framework to link against and consequently to validate their software in different configuration conditions. A distributed infrastructure of computing and storage resource to support the software in different configuration conditions is a crucial ETICS objective. Collecting, organizing and publishing middleware and application configuration information is another requirement in order to facilitate interoperability analysis at the early stages of development and implementation. The collection from the scientific community of sets of test suites is necessary to help users to validate deployed middleware, applications and their products for specific uses. Another requirement is to increase the awareness of the need for high-quality standards in the production of software and to promote the identification of common quality guidelines and principles and their application to software production in open-source academic and research organization. The international collaboration between research projects and establishment has to be promoted and a virtual community in the field of software engineering has to be established in order to contribute to the development of standards and advancement in the art of quality assurance.

Via the ETICS service, users can explore meaningful metrics pertaining to the quality of their software. Furthermore, the ETICS service also offer a repository of ready-built components, services and plug-ins, with a published-quality level. The quality metrics provided by the ETICS services and available for each package in the repository help guide the user in selecting reliable software dependencies. Finally, the repository also contain pre-built packages for specific hardware platforms and operating systems, which will help the developers to assess the platform independence of their entire service, including each and every dependency the service is relying on.

The task of building Grid applications that run reliably and efficiently on Grid resources remains extremely difficult. In general those applications consist of a heterogeneous collection of sub-applications that are handled together in order to form a large distributed application. From the perspective of Grid application developers, the ETICS service should allow them to automate their build and test procedures, and to verify that all the pieces work together. Developers should not be equipped with a programming framework in order to have details of most Grid services. In addition they should work by using a consistent, non-complex model in which their application could be composed from well tested, reliable sub-units. Most Grid and distributed software projects invest in a build and test system in order to build and test automatically their software and monitor key quality indicators. ETICS takes requirements from many Grid and distributed software projects (e.g., QUATTOR, EGEE, DILIGENT) and offers a generic yet powerful solution for building and testing software. Building software via such a systematic
process can provide a rich pool of published quality components, services and plugins, which the next generation of Grid and distributed applications could be based on and composed of.

ETICS aims to establish a distributed and managed infrastructure providing common software engineering tools and processes. Therefore, integrated pools of resources also have to be maintained and managed for running automated builds and test suites. For this reason, a centre of exchange for software configuration information and documentation is required in order to allow projects to best organize their software. In addition, a repository of standard benchmarks and interoperability information that new projects can use to validate their products has to be defined.

To fulfill ETICS requirements, firstly, it is fundamental to identify a number of existing resource pools with adequate network, hardware and software capabilities that can be federated to run builds and test suites. The next step consists of promoting the definition and adoption of common configuration management and quality assurance guidelines to foster software interoperability and reliability. This activity lead to the establishment of the "quality certificate" for the software developed by the research community. Then, it is mandatory to run a professionally managed service in support of software projects and offload them from the need of setting up dedicated infrastructures and help improve the overall quality of the software. Finally, it is required to set up a repository of software configuration information, documentation, benchmarking data and reference test cases that projects can use to validate their products and ultimately produce increasingly better and more efficient software. The ETICS services must be rapidly and efficiently accessible by users from different locations and platforms. A Web-based portal is therefore considered to be the right choice as a means of accessing the services and resources provided by ETICS.

3 Related Work

In the area of software development management, we consider Gump and Maven [10] sponsored by the Apache software Foundation [11]. Gump is a distributed software build service, whilst Maven is a software build and project management tool. These projects are often referred as "social experiments", since they try to address not only the technical problems but also the issues that arise when software is developed by several communities. Gump supports the build of software for a particular technology (i.e., Java), and does not support testing tools.

Tools that provide customizable portals for software project management are also available such as Sourceforge [12], Savannah [13] and GForge [14]. Sourceforge is provided as a service, whilst Savannah (like GForge) is supplied as a software package that can be installed and customized by users. Moreover, GForge automatically creates a repository and controls access to it depending on the role settings of the project. These projects give a Web-based portal that users can access to register their software, track its evolution and interact with the user community. They do not provide, for example, automated software build and testing.
ETICS, compared with Gump, integrates in a single framework of build, testing and reporting functionality. It also gives the provision of software engineering functionalities such as the software build and testing, compared with Sourceforge, Savannah and GF orge. Within EGEE, the gLite Configuration and Build System [15], [16] was set up to build automatically the software and support the integration activities. OMII Europe [17] (acronym for Open Middleware Infrastructure Institute) identifies, proposes and promotes middleware and applications that want to improve their quality and be part of a coherent development effort at pan-European and international level. ETICS profits from the gLite experience to lay the foundations of a more general configuration, integration and build system to match the needs of distributed software development. OMII and ETICS projects are to complement each other. The latter, in fact, can provide the underlying development process and quality assurance facility.

4 Quality Assurance Certification Process

As with other major issues, the ETICS project must deliver a feasibility study for the implementation of a Quality Assurance (QA) certification process for Grid and distributed software projects which want to adopt a quality label and promote their products. In fact, software products need to be certified in order for them to be accepted by other projects and user communities. In most cases, it is very difficult to assess the quality of software and most standards address the problem by specifying various statistical methods of calculating the defect density of the software or by setting the following equation: quality of the software = quality of the company/institute. Although such metrics [18] are certainly of great importance in the evaluation of software quality, there are no agreed definitions of such concepts based on usability or suitability principles. Such a task is of great difficulty and the ETICS infrastructure is in a strategic position to participate in and promote discussions on the definition of more qualitative rather than quantitative standards. For example, “is the software good enough for what I need to do?”, rather than, “it has passed a set of reference tests?”, or “it has less than 0.9 defects per thousands lines of code”.

Some useful metrics are: 1. the number of implemented requirements based on the requirements that can be tracked in a project tracker. Since the percentage is relative to the number of requirements in the tracker, which is supposed to increase in time, a small increase in percentage may still imply a large number of implemented requirements. This metric must therefore be accompanied also by the real number of received and implemented requirements; 2. defect and usage correlation: this is a metrics that tries to correlate the usage of a package or a system (e.g., number of users, number of days between updates, number of packages depending or using this package, number of bugs per package, trends) with the number of defects found before releasing; 3. the number of lines, classes, methods counts and stats fragility, complexity, hierarchy size. The idea is to demonstrate in practice that the application of proper software engineering and QA techniques has a measurable impact on the final quality and usability of the software.
The ETICS study is a full assessment of what procedures, tools and rules maximize the project's chances of delivering reliable [19], interoperable software based on the real data collected in two years of activity. ETICS wants to propose the results of this study as the starting point of a coordinated QA Certification activity in the context of a more permanent Grid infrastructure initiative. The investigation is conducted using as benchmarks the software developed by some of the ETICS partners in other projects. Currently, the EGEE-gLite software, the DILIGENT software and the VDT [20] software use the ETICS service. Additional software packages (e.g., QUATTOR) have already been selected in order to start using ETICS services and to be deployed by using ETICS infrastructure. The ETICS system performs both static and dynamic analysis: the former consists of evaluating the adherence to user defined coding conventions and the source code, and making custom analysis (e.g., IPv6 compliance); the latter consists of for example evaluating the code coverage for unit and system tests, compliance tests, stress tests, performance tests.

5 ETICS System Architecture

The ETICS system architecture is based on the requirements described in Section 2. It is split into several entities, as shown on Figure 1: Web Service; Web Application; command line interfaces; data model and storage; job execution engine. The Data Model and Storage is designed to organize a software project by using the high-level entities such as the project structure, the build configuration, the security information, the build and job result set. The data model describes explicitly the objects and the relationships between objects. In addition, the model allows representing the results of running a build and test job in a way that can be consumed by the Web Application to generate reports. The

![Fig. 1. The ETICS System Architecture](image-url)
data storage back-end holds the persisted data model and supports different deployment models. For example, as the number of requests increases, with more users using the service, a potential use cases require database to be hosted on different machines via a load balancing algorithm and/or several instances of the Web Service to be deployed in parallel on multiple hosts. The Web Service is the entity providing business logic for the entire service, used by both the client and the Web Application. An important goal of the Web Service is to abstract the data storage backend, which holds the persisted version of the ETICS data model. For simplicity and better scalability, the Web Service is stateless. This means that it does not use a stateful Web Service paradigm, such as Web Services Resource Framework (WSRF), which still has to prove itself in high-availability applications. The Web Application is responsible for allowing the user to view, monitor, configure and execute automated builds and tests. It is stateful in order to maintain the security credentials and session information, which improves the ergonomics of the interface. The Command Line Interfaces provide a similar functionality as the Web Application and makes use of the same Web Service interface for simplicity and symmetry (i.e., they have some tasks in common). The command line interfaces can be used directly by the user on local resources (e.g., a developer machine). Furthermore, the same client is used in an almost identical context by the NMI build and test framework. This similarity is crucial to avoid context switching between local and remote builds/tests\(^2\), which would reduce the usability and reliability of the system. The Job Execution Engine allows the ETICS service to offer the user the automation of builds and tests, possibly on a regular schedule, on a large set of different resources and platforms. The engine is provided by the NMI build and test framework, which builds on top of Condor [21], a specialized workload management system for computing intensive jobs.

5.1 Security in ETICS

The underlying security infrastructure is based on digital certificates. Both the Web Application and the client authenticate themselves using standard x.509 [22] certificates. Users are modeled as fully qualified x.509 principal names as they appear in standard x.509-compliant certificates. The Web Service verifies the user certificate Distinguished Name (DN) in the database of existing users involved in a project, and it allows or denies the operation based on the roles (summarized in Figure 2) assigned to the users. From that point onwards, the Web Service uses a service certificate to interact with other internal services. The access control list on the persisted data will be enforced by the Web Service.

The identified roles are described in the following list: Administrator (A) is a super user enabled to perform all the operations allowed in the ETICS

\(^2\) A local build/test is performed, for instance, by the developer on his/her personal workstation whenever he/she wants. A remote build/test is submitted, for instance, by the developer on a remote system that will process it when possible (i.e., the developer does not have a total control of the resources that may be used by other users).
Fig. 2. The Hierarchy of Roles. Figure has to be read from left to right. The role G is the lowest important respect to the role A. The arrow points out that the destination role can perform the same action of the source one.

infrastructure; Module Administrator (MA) is responsible for handling the project by using the ETICS services; Developer (D) works on the implementation of the software; Integrator (I) runs software verifying if it works and register packages; Tester (T) submits and stores test; Release Manager (RM) is responsible for defining the release candidate of the project, publishing packages, creating release notes and other documentation; Guest (G) has only read access.

6 A Rigorous Definition of the Basic Concepts

In this section we introduce in a rigorous way the basic concepts and the definitions of the ETICS data model. The schema has been designed to model a generic software project, its internal structure and its relationships and the operations required to build and test such a project. The model is inspired by the Common Information Model (CIM) Application model [23] and the Object Management Group (OMG)'s Model Driven Architecture [24], but it adds definitions for the operations of software construction (build) and verification (testing) which are missing from the above mentioned models. The ETICS data model is composed by several elements which can be organized in software structure, build configuration and security information.

6.1 Software Structure Definitions

We provide the software structure definitions, characterized by the concepts of component, subsystem and project. We start by defining the concept of component that is different from the same definition used in the object-oriented programming:

Definition 1. A component \( c \) is defined as a collection of objects providing a well-defined more limited functionality within the system architecture.

A proper understanding of this definition requires investigation of the concepts of object and functionality. The former can be at least a source file, or a
configuration file or a document, whilst the latter is the sum or any aspect of what a software application can do for a user\textsuperscript{1}. The software application can be formalized in the following way:

Definition 2. A software application $SA$ is a set of non-ordered components $SA = \{c_1, c_2, \ldots, c_m\}$ with $m \in \mathbb{N}$.

The set of all subset of $SA$ is called the power set of $SA$ and specified as $\mathcal{P}(SA)$. Taking into consideration the previous two definitions, we introduce the subsystem, a logical portions of the overall architecture in more specific subsets of functionalities, as follows:

Definition 3. A subsystem $SS$ is a non empty subset of the software application, defined as a set of non-ordered components $SS = \{c_{s_1}, c_{s_2}, \ldots, c_{s_n}\}$, where $c_{s_i}$ ($s_i=1, \ldots, n$ with $n \in \mathbb{N}$) are components defined in $SA$.

Definition 4. Let $SSS$ be a set of subsystems $SSS = \{ss_1, ss_2, \ldots, ss_n\}$ where $ss_i$ ($i=1, \ldots, n$ with $n \in \mathbb{N}$) are subsystems defined in $SA$, so that $SSS \subseteq \mathcal{P}(SA)$.

For the sake of simplicity in the definition of subsystem we neglect the fact that it could be composed not only of components but also of other subsystems. Taking into account the component, software application and subsystem definition, we formalize the project, a complete software package providing well-defined high level functionalities according to predefined user requirements, as follows:

Definition 5. A project $P$ is a non empty subset of subsystems and components of the software application, defined as a set of non-ordered elements $P = \{p : p \in SSS \lor p \in SA\}$.

Definition 6. Let $PRJ$ be the set of projects $PRJ = \{prj_1, prj_2, \ldots, prj_h\}$ where $prj_j$ ($j=1, \ldots, h$ with $h \in \mathbb{N}$) are projects defined in $SA$, so that $PRJ \subseteq \mathcal{P}(SA)$.

For instance, a project can be composed of one or more components, of one or more subsystems, of a combination of components and subsystems.

Definition 7. A module $m$ is a generic entity, where entity can be a project, a subsystem or a component. Let $M$ be the set of modules $M = \{m_1, m_2, \ldots, m_n\}$ where $m_i$ ($i = 1, \ldots, n$ with $n \in \mathbb{N}$) are modules defined in $M$ and $M = \{PRJ \cup SSS \cup SA\}$.

6.2 Build Configuration Definitions

In this section we provide the build configuration definitions, composed of platform, resource, commands, and configuration concepts. An example of platform is $slc4\_x86\_64\_gcc345$ where $slc4$ is the operating system, $x86\_64$ is the machine architecture and $gcc345$ is the compiler.

\textsuperscript{1} Szyperksi [25] gives the definition of software component, commonly used in the object-oriented programming.
Definition 8. Let OS be the set of operating systems. Let CMP the set of compilers. Finally, let MA be the set of machine architectures. PLT = OS × CMP × MA is a set of platforms.

Definition 9. Let \( R \) be the set of resources \( R = \{r_1, r_2, ..., r_b\} \) with \( b \in \mathbb{N} \).

We now define the applications \( f \) and \( g \). The former enables to define the concept that it is possible to have several resources with the same platform, and exactly one platform for each resource. The latter enables to define the concept that a project runs on several resources, and a resource can run more than one projects.

Definition 10. The application \( f: R \rightarrow \text{PLT} \) from \( R \) to \( \text{PLT} \) is a correspondence between \( R \) and \( \text{PLT} \) such that for each resource \( r \in R \), \( \exists \) exactly one platform \( pt \in \text{PLT} \) such that \( f(r) = pt \).

We observe that given the resources \( r_j \) and \( r_k \) with \( j \neq k \) (i.e., \( r_j \neq r_k \)) and \( r_j, r_k \in R \) and the platform \( pt \in \text{PLT} \), it could be that \( f(r_j) = pt \) and \( f(r_k) = pt \), so that the application \( f \) is not injective.

Definition 11. Consider the application \( g: R \times \text{PRJ} \rightarrow \{0, 1\} \) such that

\[
g(r, prj) = \begin{cases} 
1 & \text{if \( prj \) runs on \( r \)}, \\
0 & \text{otherwise}. 
\end{cases}
\]

We observe that given:

- the resources \( r_j \) and \( r_k \) with \( j \neq k \) (i.e., \( r_j \neq r_k \)) and \( r_j, r_k \in R \) and the project \( prj \in \text{PRJ} \), it could be that \( g(r_j, prj) = 1 \) and \( g(r_k, prj) = 1 \);
- the projects \( prj_j \) and \( prj_k \) with \( j \neq k \) (i.e., \( prj_j \neq prj_k \)) and \( prj_j, prj_k \in \text{PRJ} \) and the resource \( r \in R \), it could be that \( g(r, prj_j) = 1 \) and \( g(r, prj_k) = 1 \).

\( EV \) environment variables are a set of dynamic values that can affect the way running processes will behave.

Definition 12. A configuration \( \text{conf} \) collects some informations needed to download, build and test a subset of software for each supported platform. Let \( \text{CONF} \) be the set of configurations.

The \( \text{PRP} \) property is a set of custom attributes that a configuration requires at build-time such as compilation flags.

Definition 13. Let \( \text{VCSC} \) be the set of version control system commands. Let \( \text{BC} \) be the set of build commands. Finally, let \( \text{TC} \) be the set of test commands. \( \text{CMN} = \text{VCSC} \times \text{BC} \times \text{TC} \) is a set of commands.

Subversion [26] and Control Version System [27] are examples of version control systems. An example of version control system command is \textit{cvs co component}. The build commands are used to configure, build, package, create documentation, remove generated build files (e.g., \textit{make clean}, \textit{make doc}), whilst the test commands are involved in running, for example, specialized unit test, coverage test, coding conventions test, functional test, stress test and performance test. We introduce the formal definition of relationship between platform and configuration:
Definition 14. Consider the case \( h : \text{CONF} \times \text{PLT} \rightarrow \text{CMN} \times \text{EV} \times \text{PRP} \) such that

\[
h(\text{conf}, \text{plt}) = \begin{cases} 
    (\text{cmn}, \text{ev}, \text{prp}) & \text{if plt is defined for conf}, \\
    (\text{cmn}_d, \text{ev}, \text{prp}) & \text{if plt is not defined for conf but plt}_d \text{ is associated with conf}, \\
    \emptyset & \text{if plt is not defined for conf and plt}_d \text{ is not associated with conf}, 
\end{cases}
\]

where \( \text{plt}_d \) is the default platform, and \( \text{cmn}_d \) is the default command and \( \{\text{CMN} \times \text{EV} \times \text{PRP}\} \cup \{\emptyset\} \).

Definition 15. Consider the case \( k : \text{CONF} \times \text{PLT} \times \text{M} \rightarrow \{0, 1\} \) such that

\[
k(\text{conf}, \text{plt}, \text{m}) = \begin{cases} 
    1 & \text{if m is defined for conf and m}, \\
    0 & \text{otherwise} 
\end{cases}
\]

where \( \text{m} \) represents the dependency by which \( \text{conf} \) depends on.

We formalize the relationship between modules and configurations as follows:

Definition 16. The application \( w : \text{CONF} \rightarrow \text{M} \) from \( \text{CONF} \) to \( \text{M} \) is a correspondence between \( \text{CONF} \) and \( \text{M} \) such that for each configuration \( \text{conf} \in \text{CONF}, \exists \) exactly one module \( \text{m} \in \text{M} \) such that \( w(\text{conf}) = \text{m} \).

We observe that a module can have more than one configuration.

6.3 Security Information Definitions

Finally, we provide definitions for security information. Let \( \text{U} \) be the set of users. Let \( \text{RL} \) be the set of roles \( \text{RL} = \{\text{A}, \text{MA}, \text{D}, \text{I}, \text{T}, \text{RM}, \text{G}\} \). We now define four applications \( v, z, t, q \) that enables a user to act on a configuration, project, subsystem, and component respectively with a specific role.

Definition 17. Consider the case \( v : \text{U} \times \text{RL} \times \text{CONF} \rightarrow \{0, 1\} \) such that

\[
v(u, \text{rl}, \text{conf}) = \begin{cases} 
    1 & \text{if the user u has the role rl for the configuration conf} \\
    0 & \text{otherwise} 
\end{cases}
\]

Definition 18. Consider the case \( z : \text{U} \times \text{RL} \times \text{PRJ} \rightarrow \{0, 1\} \) such that

\[
z(u, \text{rl}, \text{prj}) = \begin{cases} 
    1 & \text{if the user u has the role rl for the project prj} \\
    0 & \text{otherwise} 
\end{cases}
\]

Definition 19. Consider the case \( t : \text{U} \times \text{RL} \times \text{SSS} \rightarrow \{0, 1\} \) such that

\[
t(u, \text{rl}, \text{ss}) = \begin{cases} 
    1 & \text{if the user u has the role rl for the subsystem ss} \\
    0 & \text{otherwise} 
\end{cases}
\]
Definition 20. Consider the case \( q : U \times RL \times SA \rightarrow \{0, 1\} \) such that

\[
q(u, rl, sa) = \begin{cases} 
1 & \text{if the user } u \text{ has the role } rl \text{ for the component } sa \\
0 & \text{otherwise.}
\end{cases}
\]

7 Supported Operations

The operations supported by the existing ETICS implementation are classified in two main categories: read only and edit. The former one allows users to get information about software, build it and test it, whilst the latter one allows users to interact with ETICS services, such as data storage. In this section we provide more details on the set of the read only and edit categories that are the most meaningful for users, through two list of operations that are respectively able for instance to build software and to support different actions on the ETICS data elements. Table 1 describes the type of the edit operations and their meanings.

Get a Project(prj). This operation gets information about the project prj.
Prereq: The project prj \( \in PRJ \).
Result: prj is returned back to the user.

Checkout a Configuration(conf). This operation checkouts the configuration conf.
Prereq: The operation Get a Project(prj) has to be performed with success and the configuration conf \( \in CONF \).
Result: conf with its dependencies and subset of software are returned back to the user.

Build a Configuration(conf). This operation builds the configuration conf.
Prereq: The operation Checkout a configuration(conf) has to be performed with success.
Result: The subset of software associated to that conf is built.

Test a Configuration(conf). This operation tests the configuration conf.
Prereq: The operation Checkout a configuration(conf) has to be performed with success.
Result: The subset of software associated to that conf is tested.

Add a User(a). This operation adds a new user a.
Prereq: The user \( u \in U \) who performs this operation has the role A \( \in RL \).
Result: a \( \in U \).

Remove a User(a). This operation remove the user a.
Prereq: The user \( u \in U \) who performs this operation has the role A \( \in RL \), a \( \in U \) and the applications \( v(a, rl, plt) = z(a, rl, plt) = t(a, rl, plt) = q(a, rl, plt) = 0 \), \( \forall \, rl \in RL \) and \( plt \in PLT \).
Result: a \( \not\in U \).
Table 1. Type of edit operations

<table>
<thead>
<tr>
<th>Operations</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td>insert new element in the ETICS data model</td>
</tr>
<tr>
<td><strong>modify</strong></td>
<td>change some element parameters in the ETICS data model</td>
</tr>
<tr>
<td><strong>remove</strong></td>
<td>delete the element in the ETICS data model</td>
</tr>
<tr>
<td><strong>clone</strong></td>
<td>copy an element of the ETICS data model</td>
</tr>
<tr>
<td><strong>prepare</strong></td>
<td>prepare a template file with element parameters</td>
</tr>
</tbody>
</table>

Add a Platform(plt). This operation adds a new platform plt.
Prereq: The user $u \in U$ who performs this operation has the role $A \in RL$.
Result: $plt \in PLT$.

Modify a Platform(plt). This operation modifies the platform plt.
Prereq: The user $u \in U$ who performs this operation has the role $A \in RL$, and $plt \in PLT$.
Result: $plt \in PLT$.

Remove a Platform(plt). This operation removes the platform plt.
Prereq: The user $u \in U$ who performs this operation has the role $A \in RL$, $plt \in PLT$, and the applications $h(\text{conf}, plt) = v(b, rl, plt) = \varepsilon(b, rl, plt) = t(b, rl, plt) = q(b, rl, plt) = 0$, $\forall \text{conf} \in CONF$, $b \in U$ and $rl \in RL$.
Result: $plt \notin PLT$.

Add a Resource(r). This operation adds a new resource r.
Prereq: The user $u \in U$ who performs this operation has the role $A \in RL$.
Result: $r \in R$.

Modify a Resource(r). This operation modifies the resource r.
Prereq: The user $u \in U$ who performs this operation has the role $A \in RL$, and $r \in R$.
Result: $r \in R$.

Remove a Resource(r). This operation removes the resource r.
Prereq: The user $u \in U$ who performs this operation has the role $A \in RL$, $r \in R$ and the application $g(r, plt) = 0$, $\forall plt \in PLT$.
Result: $r \notin R$.

Add a Project(p). This operation adds a new project p. It also associates to $p$ a configuration conf.
Prereq: The user $u \in U$ who performs this operation has the role $A \in RL$.
Result: $p \in PRJ$ and conf $\in CONF$.

Modify a Project(p). This operation modifies the project p.
Prereq: The user $u \in U$ who performs this operation has the role $A \in RL$, and $p \in PRJ$.
Result: $p \in PRJ$. 
Remove a Project($p$). This operation removes the project $p$. It also removes all its configurations $conf_i$, $i=1,...,n$, $n \in \mathbb{N}$.
Prereq: The user $u \in U$ who performs this operation has the role $A \in RL$, and $p \in PRJ$.
Result: $p \notin PRJ$, and $conf_i \notin CONF$, $i=1,...,n$, $n \in \mathbb{N}$.

Add a Module($m$). This operation adds a new module $m$. It also associates to $m$ a configuration $conf$.
Prereq: The user $u \in U$ who performs this operation has either the role $A$ or the role $MA$ with $A$, $MA \in RL$.
Result: $m \in \{M \setminus PRJ\}$ and $conf \in CONF$.

Modify a Module($m$). This operation modifies the module $m$.
Prereq: The user $u \in U$ who performs this operation has either the role $A$ or the role $MA$ with $A$, $MA \in RL$.
Result: $m \in \{M \setminus PRJ\}$.

Remove a Module($m$). This operation removes the module $m$. It also removes all its configurations $conf_i$, $i=1,...,n$, $n \in \mathbb{N}$.
Prereq: The user $u \in U$ who performs this operation has either the role $A$ or the role $MA$ with $A$, $MA \in RL$, and the application $k(conf, plt, m) = 0$, $\forall conf \in CONF$ and $plt \in PLT$.
Result: $m \notin M$, and $conf_i \notin CONF$, $i=1,...,n$, $n \in \mathbb{N}$.

Add a Configuration($m$, $conf$). This operation adds a new configuration $conf$ to the module $m$.
Prereq: The user $u \in U$ who performs this operation has either the role $A$ or the role $MA$ or the role $D$ with $A$, $MA$, $D \in RL$, and $m \in M$.
Result: $conf \in CONF$.

Modify a Configuration($m$, $conf$). This operation modifies the configuration $conf$ of the module $m$. It also associates to $conf$, if there is at least one plt for that conf, environment ev, property prp, command cmn, dependency.
Prereq: The user $u \in U$ who performs this operation has either the role $A$ or the role $MA$ or the role $D$ with $A$, $MA$, $D \in RL$, $m \in M$, and $conf \in CONF$.
Result: $conf \in CONF$, and if $\exists$ at least one plt in $PLT$ for conf, cmn $\in CMN$, $cn \in EV$ and prp $\in PRP$.

Remove a Configuration($m$, $conf$). This operation removes the configuration $conf$ of the module $m$. It also removes all its environments $ev_j$, properties $pr_j$, commands $cmn_h$, $i=1,...,n$, $n \in \mathbb{N}$, $j=1,...,m$, $m \in \mathbb{N}$ and $h=1,...,c$, $c \in \mathbb{N}$.
Prereq: The user $u \in U$ who performs this operation has either the role $A$ or the role $MA$ or the role $D$ with $A$, $MA$, $D \in RL$, and the application $k(conf, plt, m) = 0$, $\forall m \in M$.
Result: $conf \notin CONF$, and $ev_i \notin EV$, $i=1,...,n$, $n \in \mathbb{N}$, $prp_j \notin EV$, $j=1,...,n$, $n \in \mathbb{N}$, and $cmn_h \notin EV$, $h=1,...,n$, $n \in \mathbb{N}$.
8 Use Cases

A number of use cases have been identified in the framework of the EGEE, DILIGENT and VDT projects. The selection of user community applications that are willing to start using ETICS from the very beginning is an important asset to ETICS in order to gather feedback on the impact and value of the proposed service on the entire project lifecycle. Some of the use cases are: building and testing software locally and remotely by using both the Web Application and the Command Line Interfaces. In addition to perform the remote building and testing ETICS interacts with NMI framework; register resources as public or private. This means users can steer their build and test jobs to their private resources, or add resources in the public pool, in order to share them with other ETICS users; download the source and build a patched package by using different ways: source tarballs, source packages and source rpms, checked out source against checked out binaries; build everything from source; parallel build of software on different platforms and automatic distribution of the packages in different formats (e.g., tarballs, rpms, and debs).

9 Conclusion

In summary, we described how ETICS performs build, configuration, integration and test for Grid and distributed software projects. Requirements for designing the service were also described, taking into account the perspective of application developers and the needs of user communities. We mentioned other systems and products used for managing software development such as Gump and Maven; we briefly talked about projects interested in providing customizable portals for software project management such as Sourceforge and Savannah, and also hinted to other similar initiatives involved in improving the quality of Grid middleware such as OMII Europe, and configuring and build code such as the EGEE gLite Configuration and Build System. The ETICS architecture was explained, together with the formalization of the basic concepts of the ETICS data model and of the edit operations, that belong to the most meaningful category of operations.

Using the valuable feedback from DILIGENT, EGEE, QUATTOR and VDT we will improve ETICS expanding the number and type of metrics and collected data. Future work will include investigation of hardware virtualisation. In particular, the possibility of instantiating on-demand virtual machines tailored to match job requirements is seen as a powerful tool to introduce flexibility and strict reproducibility in the system. In addition, it may help overcome security concerns that arise when test jobs require running as root or Administrator on the target node. An additional area of investigation is the co-scheduling of multiple test jobs onto separate resources. Of particular interest is the case of jobs with dependencies on the availability of external services. In this case, the system should be able to pause a job, deploy the required services, propagate the relevant configuration information to the paused job and resume it, cleaning all resources after the execution of the job. Again the use of virtualisation may help in this task.
Acknowledgement

We would like to thank colleagues from the ETICS project, Peter Couvares, Paolo Fabrini, Istvan Forgacs, Anatoly Karp, Andrea Manieri, and Cristina Vistoli, who have provided useful suggestions to complete this paper. This work is partially funded by the European Commission under contract number INFSOM-RI-026753.

References

15. A. Di Meglio, Developers’ Guide For the gLite EGEE Middleware, https://edms.cern.ch/file/463700/0.7/.