J Grid Computing (2015) 13:19–34
DOI 10.1007/s10723-015-9324-0

Revisiting the Anatomy and Physiology of the Grid

Chris A. Mattmann · Joshua Garcia · Ivo Krka · Daniel Popescu · Nenad Medvidovic

Received: 7 July 2013 / Accepted: 6 January 2015 / Published online: 29 January 2015

Abstract A domain-specific software architecture (DSSA) represents an effective, generalized, reusable solution to constructing software systems within a given application domain. In this paper, we revisit the widely cited DSSA for the domain of grid computing. We have studied systems in this domain over the last ten years. During this time, we have repeatedly observed that, while individual grid systems are widely used and deemed successful, the grid DSSA is actually underspecified to the point where providing a precise answer regarding what makes a software system a grid system is nearly impossible. Moreover, every one of the existing purported grid technologies actually violates the published grid DSSA. In response to this, based on an analysis of the source code, documentation, and usage of eighteen of the most pervasive grid technologies, we have significantly refined the original grid DSSA. We demonstrate that this DSSA much more closely matches the grid technologies studied. Our refinements allow us to more definitively identify a software system as a grid technology, and distinguish it from software libraries, middleware, and frameworks.

Keywords DSSA · Physiology · Anatomy · OODT · Software architecture

C. A. Mattmann · N. Medvidovic
Computer Science Department,
University of Southern California,
Los Angeles, CA 90089, USA

N. Medvidovic
e-mail: neno@usc.edu

C. A. Mattmann (✉)
Jet Propulsion Laboratory,
California Institute of Technology,
Pasadena, CA 91109, USA
e-mail: chris.a.mattmann@nasa.gov; mattmann@usc.edu

I. Krka · D. Popescu
Google Inc., 1333 2nd Street, Santa Monica, CA, USA

I. Krka
e-mail: krka@google.com

D. Popescu
e-mail: popescudm@gmail.com

J. Garcia
George Mason University, 4400 University Drive, MS 4A5,
Fairfax, VA 22030-4444, USA
e-mail: joshuaga@usc.edu

1 Introduction

Over the past half-century, computing has undergone several transformations that have fundamentally changed the manner in which humans use computers and the nature of problems that can be solved with computers. Grid computing [1, 2] is a recent advance that shows promise of enabling another such transformation. The grid allows virtually any person or organization to solve a variety of complex problems by utilizing the computing resources beyond those of just a small cluster of computers.

Today, grids have been used successfully in several domains, including cancer research [3], planetary science [4], earth science [5], and astrophysics [6, 40, 44].

Over the past several years, a number of technologies have emerged, claiming to be grid technologies or grid platforms (e.g., see Table 1). Our own work resulted in two related such technologies. The first, Apache OODT [7], is a data grid platform currently in use at NASA and National Cancer Institute's Early Detection Research Network. The second, GLIDE [8], is a mobile grid platform. The early literature also resulted in several "big picture" publications that tried to establish the underlying principles of the grid: its "anatomy" [2] and "physiology" [1], which describe the grid's software architecture, as well as its overarching requirements [9–15]. Even though their authors likely did not view them that way, these reference requirements and architecture together comprised a domain-specific software architecture (DSSA) [16] for the domain of grid computing.

However, the publications in this area have had some common shortcomings. The specific technologies have been unclear as to what traits make them suitable for grid computing. This is in part because the "big picture" publications have invariably lacked rigor and have been open ended in their characterization of the grid; additionally, many were inspired by their authors' experience drawn from a single approach whose broader applicability is questionable. We have experienced first-hand the potential confusion stemming from this: our initial attempt at publishing the work behind GLIDE [8] yielded reviews stating that GLIDE is not a grid platform, but rather "a simple object-oriented framework".

We were puzzled by this as GLIDE shares many concepts with the highly successful OODT. In order to be able to refute such criticisms, however, we needed to understand intimately what constitutes a grid platform. In other words, we needed to have a precise understanding of the grid's DSSA. To that end, nine years ago we commenced a pilot study [17] in which we attempted to extract the subset of the architectural principles underlying the grid from the grid's published "anatomy" [2]. We then recovered, using the source code and existing documentation, the architectures of five widely used grid technologies, and compared those architectures to the anatomy. While it was difficult to draw definitive conclusions given the scope of that study, we observed a number of discrepancies that suggested that the published anatomy of the grid is not reflective of the existing grid systems.

These discrepancies served as an impetus to spend the next six years significantly expanding the study. We report on our results in this paper. We first elaborated the proposed grid DSSA by revisiting the architecturally relevant aspects of the grid's published "anatomy" [2] as well as its "physiology" [1]. We then analyzed the source code, documentation, and usage of eighteen widely deployed grid technologies, including the five from our pilot study, in order to recover their architectures and compare them to the published grid DSSA. As in our pilot study, these technologies departed heavily from the grid's DSSA. This reinforced our suspicion that proposing a DSSA before sufficient experience with constructing systems in the domain is amassed, as the progenitors of the grid have done, is risky and error-prone. At the same time, the study enabled us to generalize from the specific experience [18] of the analyzed grid technologies and to propose a more accurate reference architecture for the grid. We argue that our DSSA is a much better fit for the domain of grid computing. Furthermore, the new DSSA enables us to categorize grid technologies and identify the extent to which a candidate technology can be considered a "grid". Though recent studies by Rimal in 2011 [41], Montes in 2012 [42] and Shamsi [43] in particular have attempted to pinpoint grid (and cloud) architectural requirements our study in particular advances the state of the art by not just identifying similar requirements, but actually using the requirements (and recovered architectural techniques) to prescribe a accurate canonical grid architecture representative of the grid technologies studied.

The remainder of the paper is organized as follows. Section 2 highlights related studies in understanding grid technologies, clouds, and architectural recovery. Section 3 summarizes the results of our analysis of grid technologies. Section 4 presents our new grid DSSA and demonstrates that it provides a significantly better fit than the previously published DSSA for the systems we studied. Section 5 presents the lessons we

Table 1 The studied grid technologies

Technology	PL	KSLOC	#ofModules	URL
Alchemi	C# (.NET)	26.2	186	http://www.alchemi.net
Apache Hadoop	Java, C/C++	66.5	1643	http://hadoop.apache.org
Apache HBase	Java, Ruby, Thrift	14.1	362	http://hadoop.apache.org/hbase/
Condor	Java, C/C++	51.6	962	http://www.cs.wisc.edu/condor/
DSpace	Java	23.4	217	http://www.dspace.org
Ganglia	C	19.3	22	http://ganglia.info
GLIDE	Java	2	57	http://sunset.usc.edu/~softarch/GLIDE/
Globus 4.0 (GT 4.0)	Java, C/C++	2218.7	2522	http://www.globus.org
Grid Datafarm	Java, C	51.4	220	http://datafarm.apgrid.org/
Gridbus Broker	Java	30.5	566	http://www.gridbus.org/
Jcgrid	Java	6.7	150	http://jcgrid.sourceforge.net/
OODT	Java	14	320	http://oodt.jpl.nasa.gov
Pegasus	Java, C	79	659	http://pegasus.isi.edu
SciFlo	Python	18.5	129	http://sciflo.jpl.nasa.gov
iRODS	Java, C/C++	84.1	163	https://www.irods.org/
Sun Grid Engine	Java, C/C++	265.1	572	http://gridengine.sunsource.net/
Unicore	Java	571	3665	http://www.unicore.eu/
Wings	Java	8.8	97	http://www.isi.edu/ikcap/wings/

learned in the process. Finally, Section 6 concludes the paper.

2 Background and Related Work

In this section, we first discuss existing studies of the grid. We then provide an overview the architectural recovery techniques used in this research.

2.1 Studies of the Grid and Cloud

Two seminal studies that have tried to underpin and motivate grid technologies have been by Kesselman and Foster, highlighting the grid's *anatomy* [2] and *physiology* [1]. The two comprise the grid's DSSA. In the interest of brevity, we will only summarize the key facets of the DSSA.

The anatomy of the grid is defined as a five-layer architecture with several over-arching requirements.

(1) *Application* – The top-most layer houses custom applications that plug into the common services of an underlying grid infrastructure.
(2) *Collective* – The next layer aggregates underlying *Resource* layer services, agglomerating information such as resource monitoring statistics, job status, and metadata for a given grid application.
(3) *Resource* – This layer encapsulates underlying heterogeneous computing resources (such as files, disks, I/O, etc.) and provides a standard interface for communicating with grid services.
(4) *Connectivity* – This layer is responsible for providing security, communication, and coordination of access from grid resources to underlying physical resources present in the bottom-most grid layer.
(5) *Fabric* – The bottom-most layer's elements include low level DBMS, disk I/O, threading and other OS-like resources, available from individual nodes in a grid.

As described in [2], the anatomy disallows "upcalls", i.e., inter-layer interaction initiated by a lower layer. The anatomy is ambiguous as to whether it is possible to "skip" layers, i.e., whether interactions can involve non-neighboring layers; the most widely referenced diagram from [2] implies that the layers are opaque. Finally, the nature of inter-layer interactions

(i.e., connectors [19]) is not elaborated; all interactions are treated as direct (local or remote) procedure calls.

The physiology of the grid focuses its attention on the *Resource* layer the grid reference architecture. It defines the core requirements and canonical services for grid resources. Each grid service defines five core interfaces: (1) *service registration*, which allows a grid service to register itself with a service registry; (2) *service location*, which enables service/resource discovery; (3) *service lifecycle management*, defining a core set of stages in a grid service; (4) *introspection*, which allows grid service capabilities to be dynamically discovered; and (5) *service creation*, allowing new grid services to be dynamically created and made available at runtime.

Although the published anatomy and physiology do much to lay the groundwork for grid software system architectures and have been very widely, often uncritically, cited, they do not readily make the distinction between grids on the one hand and traditional software libraries, middleware, and frameworks on the other [17] – though recent studies e.g., [41, 42] have indeed distinguished some of the differences in grids and *cloud computing* at various phases (requirements) and elements (components) of software.

Cloud computing involves distributed resource sharing and on-demand elastic scalability in a multi-tenant computing environment – similar to grid computing. Rimal et al. [41] identify these similarities and discuss clouds both from the vendor [46] and standards perspective [45]. The authors identify several areas in which grids and clouds are different: (1) focus; (2) resource pattern; (3) management; (4) business model; (5) interoperability, and (6) middleware. Rimal et al. state that the cloud application-programming interface (API) is still the biggest concern area, similar to the early days of grids – and further study is required in areas such as data management; maximization of bandwidth, interoperability, and QoS. Similar to our study, Rimal et al. extract requirements common to clouds and grids, categorize them as functional (measurable), or non-functional (qualitative), and map the requirements to components in clouds and grids. Our study complements and expands Rimal et al.'s work by further identifying a new architecture derived from the examined code of software systems, and by providing a basis for further study of architectural styles for grids and clouds, which were only anecdotally discussed in Rimal et al.'s paper.

Even though differences between grids, middleware, clouds, etc. have been studied there remains significant overlap of functionality and concerns between different grid layers, *Resource* and *Fabric* being the most notable example. Finally, as we have demonstrated previously [17], existing grid technologies regularly violate the reference architecture.

A handful of other studies have been conducted in this area. Unlike our study, however, they have been based entirely on system documentation, usage data, and technical papers. They corroborate several of our observations, although they have tended to contradict one another.

Shamsi et al. [43] provide a thorough systems survey and classification of the extensive requirements of data intensive clouds. Their study identifies challenges and requirements for data-intensive systems and further studies how compute clouds can support these environments. The authors derive fifteen requirements of data intensive systems / clouds including: (1) Scalability; (2) Availability and Fault Tolerance; (3) Flexibility and Efficient User Access; (4) Elasticity; (5) Sharing; (6) Heterogeneous Environment; (7) Data placement and locality; (8) Effective data handling; (9) Effective storage; (10) Support for Large data sets; (11) Privacy and access control; (12) Billing; (13) Power Efficiency; (14) Efficient Network Setup; (15) and Efficiency. About 67 % of these requirements exhibit overlap with the requirements identified in Table 1 from our prior study [17] whereas 33 % of them have no direct mapping present and are either directly related to cloud computing advances (e.g., Billing) or to advances related to data and compute intensive increases (e.g., Power Efficiency; Support for Large Data sets). Similar conclusions related to grid and cloud requirements come from Begeman et al. [40] who are concerned with grid data provenance and data locality, and encapsulation of a researcher's data within a grid for reproducibility purposes.

Many of the recent studies on grids and clouds identify increased Quality of Service (QoS) as one of the key foci for clouds as different from grids, for example the study by Montes et al. [42]. The authors explain that over the years many of the issues in grid computing are derived from technical and political underpinnings – the political issues stem from agen-

cies and organizations sharing and trusting resources; and the technical issues can be summed up as relating to "complexity". Clouds are suggested as one means of addressing this complexity issue in grids. The authors define a service-level system management model for clouds and grids pinpointing a key difference related to grids focusing on the *structure* element of that model (e.g., virtual organizations), whereas clouds focus on the *function* element of the model, e.g., with a focus on particular services, and applications - this conclusion is consistent with our own prior study [47], as well as the outcomes identified in this paper.

Finkelstein et al. [20] study data grid systems, which deal mainly with large-scale data management, processing, and dissemination. In the study, thirteen data grid systems are compared along the dimensions of five *architectural styles*, such as client/server and peer-to-peer [19]. The authors conclude that, of the thirteen systems examined, only two appear to support the layered architectural style (globus2 [2] and the European Data Grid [21]). In addition, Finkelstein et al. observe that, while the non-functional grid requirements are fairly well specified, the functional grid requirements are quite broad. This is consistent with Rimal et al. who identify 14 non-functional requirements, 3 functional/non-functional requirements (overlapping) and 5 functional requirements in Table 5 of their paper [41].

Venugopal et al. [22] identify eight key characteristics that data grid systems must support: *proliferation of data, geographical distribution, single source, uni?ed name space, limited resources, local autonomy, access restrictions*, and *heterogeneity*. In contrast to Finkelstein et al., the authors argue that each of these characteristics is naturally mapped to a four-layer data grid reference architecture. This architecture is conceptually similar to the grid's five-layer DSSA, except that it delegates the functionality of the *Resource* layer to the grid *Fabric* layer. In addition, this mapping is significantly coarser grained than requirements mappings efforts present in the studies by Shamsi et al. [43] and by Rimal et al. [41]. Our own study also significantly expands this classification by introducing a more accurate grid architecture.

Finally, Yu and Buyya [23] focus on grid workflow systems. According to the authors, grid workflow systems can be classified along five dimensions, namely, support for *workflow design, information retrieval, workflow scheduling, fault tolerance*, and *data movement*. The authors classify ten grid workflow technologies along these dimensions and their sub-dimensions (omitted for brevity). In contrast to Finkelstein et al.'s study, one of Yu and Buyya's key conclusions is that QoS requirements for grid workflow applications are ill defined and rarely addressed in the systems they studied – though this topic is somewhat covered by Begeman et al. [40] in a single domain (astronomy).

2.2 Architectural Recovery

Architectural recovery is the process of elucidating a software system's architecture, most frequently from source code, but also from other available artifacts [24, 25]. A full treatment of architectural recovery is beyond the scope of this paper. Here, we will briefly discuss the applicability of architectural recovery and the techniques we have employed in our work.

Numerous automated architectural recovery techniques deal with code dependency analysis (i.e., static analysis), recovering relationships such as association, composition, and generalization for object-oriented (OO) code, and recovering software trace dependency relationships, such as function calls, ownership relations, and module dependencies for procedural code. Two representative static analysis architectural recovery techniques are Rigi [26] and PBS [27]. In our work, we leverage these techniques for understanding code-level dependencies in the grid systems we studied.

In order to recover a software system's *architecture*, static code dependency analysis is typically too fine-grained and requires abstraction into higher-level architectural components. Numerous such techniques have been developed [25]. For the purposes of our study, we are leveraging Focus [28], which is particularly well-suited for capturing complex system interactions, such as those found in the grid, in the form of software *connectors*, in addition to first-class *components* and architectural *styles*. Focus clusters system modules recovered via static analysis using a set of rules based on coupling and cohesion properties such as (a) two-way dependencies, (b) aggregation, (c) association, and (d) the identification of classes with large numbers of incoming

and outgoing dependencies. As each of these four types of relationships is identified, Focus suggests how the relevant modules can be grouped into architectural components. Focus also suggests methods for examining the interaction mechanisms (i.e., connectors) that exist between the architectural components. Finally, Focus allows one to assess the fit of a candidate architectural style to a partially recovered architecture .

3 Deconstructing Grid Technologies

In order to improve our understanding of the grid and try to determine what constitutes a grid technology, we examined the available information from eighteen widely used grid technologies, summarized in Table 1. In the table, the column *PL* indicates the grid technology's primary implementation language(s); *KSLOC* identifies the size of the technology computed using the software lines of code (SLOC) counting tools *CCCC* [29] and *Sloccount* [30]; and *#ofModules* indicates a count produced by *CCCC* of all classes and any other modules for which member functions could be identified. *CCCC* provides basic code counting metrics (lines versus comments; information flow, etc.) and *Sloccount* expands on this providing time estimates and other relevant data. While we collected a tremendous amount of data in the process, we can only summarize our findings here; the interested reader can find the unabridged results of our study in [31].

We studied these technologies in light of the published DSSA [1, 2]. Similar to Klaus et al. [32], in our selection we tried to distinguish between:

(1) *Computational Grid Systems* – These types of grid systems traditionally focus on complex, large-scale computational problems, such as scientific workflows, distributed image processing, earthquake analysis, and the like. Examples of such technologies include *Alchemi, Hadoop, Condor, Globus, Gridbus Broker, Jcgrid, Pegasus, SciFlo, Sun Grid Engine, Unicore and Wings*. Requirements addressed by these technologies based on classification by Shamsi et al. [43]: *Scalability*; *Availability and Fault Tolerance*; *Flexibility and Efficient User Access*; *Elasticity*; *Sharing*; *Heterogeneous Environment*; *Power Efficiency*; *Efficient Network Setup*; *Efficiency*.

(2) *Data Grid Systems* – These types of grid systems regularly collect, manage, and disseminate large amounts of data and metadata. Data grid systems may have a compute element to them (e.g., small amount of processing to convert/transform data), as may compute grids deal with data (messages passed between components, not requiring of large and concurrent data access or movement), however, here we simply differentiate their respective foci. Examples of such technologies include *HBase, DSpace, GLIDE, Grid Datafarm, iRODS,* and *OODT*. Requirements addressed by these technologies based on classification by Shamsi et al. [43]: *Data placement and locality*; Effective*data handling; Effective storage; Support for Large data sets; Privacy and access control;*

(3) *Grid Monitoring Systems* – These types of systems provide capture, analysis, logging, and visualization of monitoring data aggregated from grid resources, such as web services (state information) and *Fabric* layer resources (e.g., hardware characteristics) across a set of nodes available via the grid. Grid monitors may themselves be present in data and/or compute grids – for example we studied one such *pure* grid monitoring system, *Ganglia*, although several other grid systems (e.g., Hadoop and Hbase) also contain monitoring components. Other systems including *Splunk* [48] are also similar examples, though due to page and time constraints are not covered in this paper. Requirements addressed by these technologies based on classification by Shamsi et al. [43]: *Scalability*; *Availability and Fault Tolerance; Power Efficiency*; *Efficient Network Setup*; *Efficiency*

We realize that our three-category classification is coarse grained at this stage, but we note it represents an initial step derived from studying many grid systems and from our prior [17] study of compute/data grid requirements along with classifications derived from Shamsi et al. [43]. We will revisit the issue of grid categories in Section 4.

To select grid technologies across each of these three areas, we applied four criteria: (1) the system should be open source, so that we could analyze its

source code; (2) the system should have available documentation to supplement source code analysis; (3) the technology should be actively used; and (4) the system should claim to be a grid. Because of the well known problem of architectural erosion [19] and the demonstrated unreliability of system documentation alone (e.g., [33]), we decided to reconstruct each system's as-implemented architecture from its source code and compare it with the published grid DSSA.

Since each grid technology we examined was open source, we were able to analyze and visualize its source code using static analysis tools. One challenge we faced was that, as can be seen in Table 1, the studied grid systems were implemented in a number of programming languages, including C/C++ (present in 44 % of the technologies), Java (83 %), Ruby (5 %), Thrift (5 %), and Python (5 %). Another challenge was that the static analysis tools vary in quality and tend to give incomplete results (e.g., see [34]). We typically applied multiple tools on the same grid system to ensure that we are correctly extracting as many static relationships as possible. In the process, we used or attempted to use over 20 static analysis tools (including, e.g., Rigi [26], PBS [27], and SHrIMP [35]). We found that four tools were able to extract the bulk of the static dependencies for each grid technology: Rational Software Architect (RSA) [36], ArgoUML [37], Understand [38], and DoXYGen [39]. A detailed evaluation of the respective strengths of each static analysis tool is outside of this paper's scope.

To fill in the "gaps" in our understanding of a given grid technology's architecture, whenever necessary we supplemented the information obtained via automated analysis with available documentation and manual inspections of the source code. Finally, we applied Focus [28] to identify the system's architectural components, connectors, and style(s), as outlined in Section 2.2. The above process resulted in architectural models for each of the eighteen grid technologies.

Our ensuing step involved "shoe-horning" each of the recovered grid components and their interactions (i.e., connectors) into the five-layered grid architecture, using the grid's anatomy and physiology [1, 2] as a guide. Figure 1 shows the results of this step for four grid technologies: *Wings*, *Pegasus*, *Hadoop,* and *iRODS*. Due to space constraints, we will illustrate our findings with results from these four technologies; for the details of the architectural recovery of all eighteen grid technologies are given in [31]. Shoe-horning was performed manually by: (1) inspecting the as-stated requirements for each grid's documentation; (2) using requirements classified by each layer of the grid architecture to place recovered components and interactions; and (3) observing behavior by running/testing out the studied grid technologies. A complete treatment of architectural recovery is outside the scope of this paper though we recommend [24] for further study. The shoe-horning process was empirically validated through the suggestion of a new grid architecture that reduces style violations, and that we believe more accurately represents the nearly twenty real-world grid systems studied.

During the shoe-horning process, we repeatedly encountered four types of discrepancies between the as-implemented architectures of the grid technologies and the grid's DSSA:

(1) *empty layers* – layers identified in the grid's anatomy that contained no recovered components, an example of which is highlighted for *Wings* in Fig. 1a;
(2) *skipped layers* – components in one layer make calls to components at least two layers below or above, such as the example for *Pegasus* highlighted in Fig. 1b, which spans the entire architecture;
(3) *upcalls* – calls made from components in a lower ("servicing") layer to components in a higher ("client") layer, such as the example for *Hadoop* highlighted in Fig. 1c; and
(4) *multi-layer components* – components that provided services which, according to the published grid DSSA, belong to two or more layers, e.g., as shown highlighted for *iRODS* in Fig. 1d.

In addition to these four types of discrepancies, in several cases we also identified *orphaned components*, whose exact location in the grid architecture we were unable to determine based on the available information. An example is shown for *Pegasus* on the right side of Fig. 1b. Orphaned components typically indicated the presence of test classes, templates, and other functionality not intended to be part of the core system.

As can be seen from Table 1, the studied systems' sizes varied widely, from 2 KSLOC to over 2 MSLOC, and from 22 to over 3600 identifiable modules. Each grid technology also varied widely in the number of violations of the grid DSSA. As an example, in *Wings* (Fig. 1a), which has 97 original modules and around 9 KSLOC, we observed three types of discrepancies: 6 upcalls, 10 skipped layers, and 1 empty layer (*Connectivity*); in addition, *Wings* also had an orphaned component (*WingsUtil*). On the other hand, in *Pegasus*, a much larger system (almost 10x as many SLOC and 7x as many modules), we noted about half as many discrepancies: 3 upcalls and 5 skipped layers; *Pegasus* also had six orphaned components, which we attribute to the fact that, as a larger system, it had more utility modules than *Wings*.

We have found a similar lack of correlation between a grid system's size and its adherence to the grid DSSA throughout our study. Another example is *Hadoop*, with 66 KSLOC and over 1643 modules: it had only 2 upcalls and 6 skipped layer violations. On the other hand, *iRODS*, which was of comparable size (84 KSLOC) but had 10x fewer code-level modules (163), had 35 upcalls, 51 skipped layer violations, and 2 multi-layer components. In fact, *iRODS* had the most problematic architecture of the eighteen systems we studied, as can probably be gleaned from Fig. 1: it had at least 2x as many upcalls and 4x as many skipped layer violations as any of the other technologies.

Overall, the most prevalent discrepancies identified in all grid technologies were upcalls and skipped layers (a total of 242), as indicated in Table 2. Each studied grid technology's architecture used upcalls (a total of 98 across the eighteen technologies) and all but one (*Condor*) skipped layers (a total of 144 across the eighteen technologies). This suggests that, as conceived in the grid DSSA, the layers share concerns and are ultimately less orthogonal than intended. Furthermore, as described in [1, 2] and summarized in Section 2, the layered architecture is conceptually abstract, failing to document both the types of grid components that should reside in each layer and the many complex and important interactions between those components that would be required for a sound analysis of grid properties. While not as pronounced, the presence of other identified violations—multi-layer components (a total of 18 across the eighteen technologies) and empty layers (a total of 5)—served to reinforce our conclusion that the existing grid architecture required further refinement.

In the next section, we describe our refinement of the grid DSSA and present the results of shoehorning the existing grid technologies into the new DSSA.

4 Reconstructing the Grid DSSA

When Foster, Kesselman, and others introduced the anatomy and physiology of the grid, they, in fact, presented a prescriptive architecture [19] based on their experiences with the Globus toolkit. Even though the Foster and Kesselman papers were lacking in low level implementation detail, by recovering the architectures of eighteen different grid technologies and comparing them to the original prescriptive architecture, we provide a gap analysis and additionally a concrete mapping between Foster and Kesselman's high level prescription, what was implemented over the last decade, and ultimately where we are today. From this, we are able to create a new DSSA that more faithfully describes and effectively differentiates grid technologies. Section 4.1 explains the building blocks and interconnections of this new DSSA. Section 4.2 describes the different architectural styles and interaction types of grid systems. Section 4.3 provides a classification of grid systems with respect to the new DSSA.

4.1 Structure of the Proposed DSSA

Figure 2 shows a structural view of the new grid DSSA. The main logical building blocks of the DSSA are software components that belong to four subsystems: *Application*, *Collective*, *Resource*, and *Fabric*. The new DSSA borrows and, as appropriate, modifies the terminology of the different grid subsystems (layers in the original DSSA) in order to better reflect the knowledge we extracted from the as-implemented grid architectures. On the other hand, we have departed from the layered organization, as it clearly was not an appropriate fit for the grid. Each identified component within the four subsystems in Fig. 2 has distinct responsibilities, which together form the grid. Our DSSA also describes the allowed interaction (i.e.,

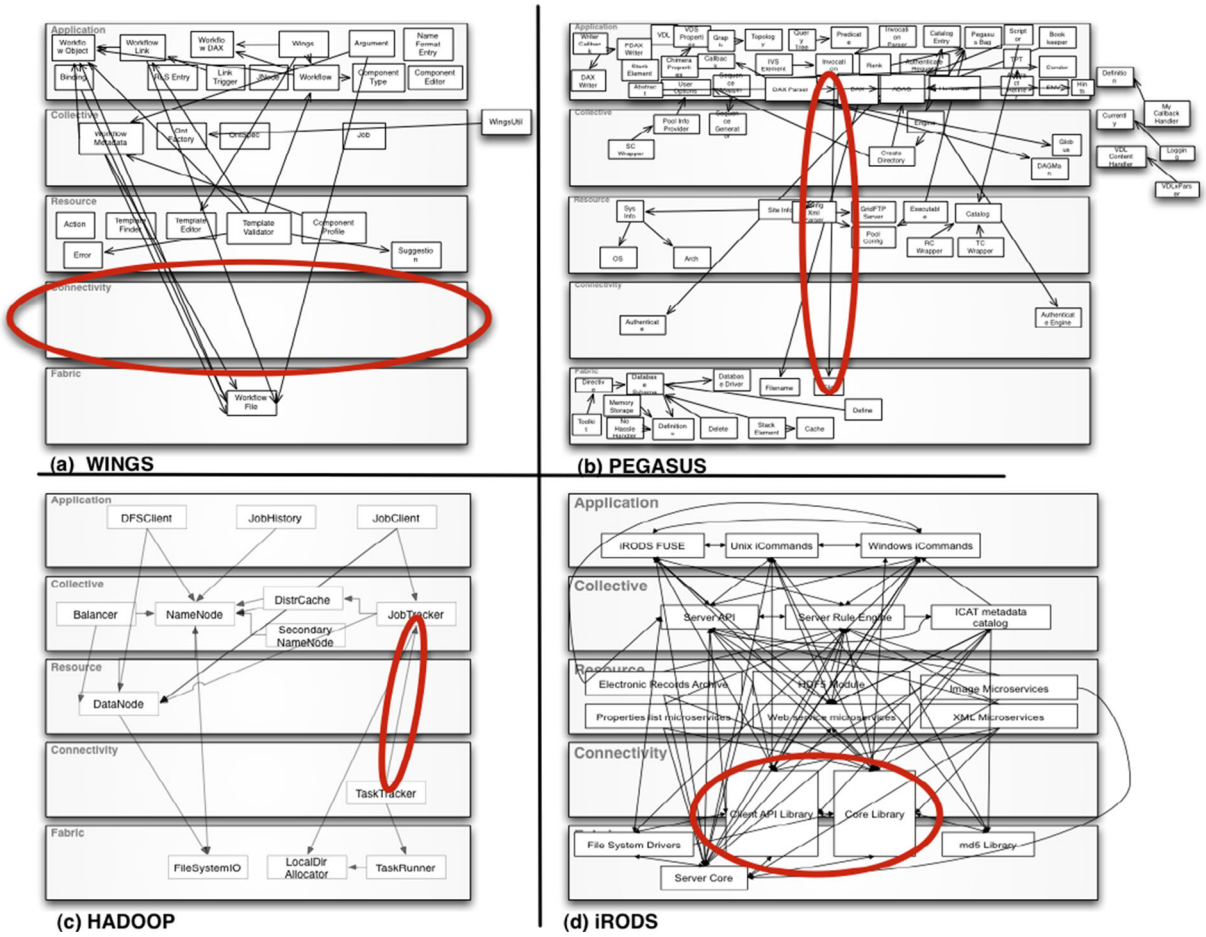

Fig. 1 Representative discrepancies identified in four of the studied grid technologies

connector) types between the different component types.

Application Components An *Application* can be any client that needs grid services and is able to use an API that interfaces with *Collective* or *Resource* components.

Collective Components The *Collective* components are the unique feature of the grid, distinguishing it from other service-oriented architectures – the *Collective* is the *core layer* that allows for the creation of virtual organizations. Other service-oriented architectures have *Resource* layer components, but they are confined to a single organization. The *Collective* components are used to orchestrate and distribute data and grid jobs to the various available resources in a manner consistent with the security and trust policies specified by the institutions within a grid system (i.e., the virtual organization [2]), the computing resources they share, and the sharing policies specified by them. *Collective* components can provide the following services: *planning* (e.g., where the data should be stored), *query federation*, job *scheduling* (e.g., to which nodes should individual tasks be assigned), maintaining *grid metadata* (e.g., storing addresses of individual nodes in the grid), collection and aggregation of *monitoring* data (e.g., measuring average utilization of the grid), and *resource registration* and *discovery*.

Resource Components The *Resource* layer of the original grid DSSA was defined as a set of protocols that build on the protocols of the *Connectivity* layer for the purpose of managing individual resources. However, components in the recovered

architectures do not manifest themselves as protocols. Therefore, in the new DSSA, *Resource* components are defined as the components that perform individual operations required by a grid system by leveraging available lower-level *Fabric* components (e.g., storing part of a file submitted to a DFS using capabilities of the local file system). Specific services of *Resource* components include managing creation and destruction of grid service instances, performing inspection on grid service instances, and providing data being monitored by *Collective* or *Application* components.

Fabric Components *Fabric* components offer lower-level access capabilities to computational and data resources on an individual node (e.g., access to file-system operations). *Fabric* components often do not provide any grid-specific functionality and, therefore, can also be used by traditional software systems.

Our examination of the recovered architectures revealed that the *Connectivity* layer described in the original grid DSSA does not actually exist as such. As described, the *Connectivity* layer provided communication and authentication protocols such as TCP/IP or Transport Layer Security (TLS) to facilitate data exchange between *Fabric* components. However, the grid architectures that we recovered indicate that *Fabric* components never interact across distributed nodes. On the other hand, the *Application*, *Resource*, and *Collective* components all use the *Connectivity* protocols, causing a large number of violations of the layered style. Given these observations, the new DSSA eliminates the *Connectivity* layer as a subsystem, encapsulating its functionality in the explicit connectors, as discussed in Section 4.2.

The new DSSA offers further design guidance by specifying a mapping of the conceptual components described above to the hardware nodes that house them and their associated multiplicities. It was unclear

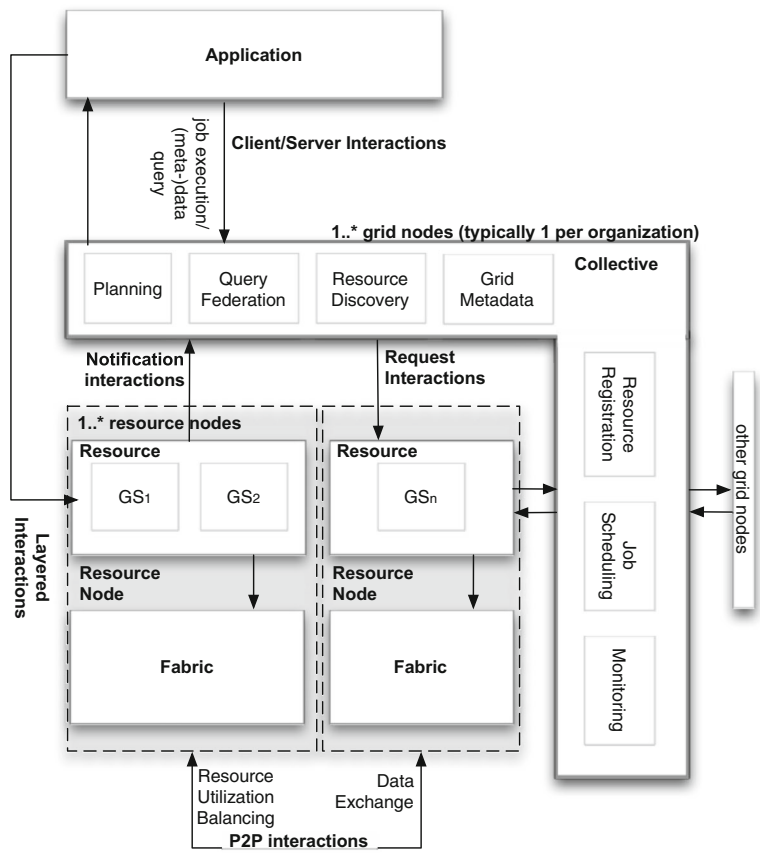

Fig. 2 Structural view of the new grid reference architecture

Table 2 Grid reference architecture discrepancies in the studied technologies

Technology	*UC*	*SL*	*ML*	*EL*
Alchemi	2	2	2	0
Apache Hadoop	2	6	0	1
Apache HBase	5	7	0	0
Condor	2	0	0	0
DSpace	1	4	1	0
Ganglia	2	1	0	1
GLIDE	2	5	0	0
Globus 4.0	6	4	5	0
Grid Datafarm	12	16	0	0
Gridbus Broker	4	3	0	1
Jcgrid	4	7	0	0
OODT	3	6	1	0
Pegasus	3	6	6	0
SciFlo	1	1	0	0
iRODS	35	51	2	0
Sun Grid Engine	2	2	0	1
Unicore	6	13	0	0
Wings	6	10	1	1
Totals:	98	144	18	5

UC = upcalls; SL = skipped layers; ML = multi-layer components; EL = empty layers

in the original reference architecture how the five layers map to physical nodes. The studied grid technologies show that *Resource*, *Collective*, and *Application* components are typically deployed on different physical nodes, but that other deployments are also allowed.

In grid systems there is typically a central *Collective* component deployed onto a single reliable node that is responsible for maintaining the grid metadata. This central component may also perform most of the planning, scheduling, monitoring, etc. efficiently and with a low-likelihood of failure. To decrease the utilization of the central node, grid system designers can also choose to deploy remaining *Collective* components on other system nodes (e.g., the job execution component can be deployed on a different node than the scheduling component). *Resource* components are typically deployed onto individual nodes in computational clusters, servers, workstations, or PCs where each node provides the computational and/or data storage resources that grid applications will use. To increase efficiency, cacheable versions of *Collective* components may be deployed onto *Resource* nodes or clusters of nodes as needed. Any component may use the *Fabric* components available on its local node, but not the *Fabric* components of remote nodes.

4.2 Interactions in the New DSSA

The original DSSA was described using only the layered architectural style. Our investigation indicated that there are four prevalent architectural styles in the grid technologies we studied: (a) client/server, which is the main interaction style between *Application* and *Collective* components as well as between *Application* and *Resource* components; (b) peer-to-peer, where the peers of the grid system are typically *Resource* components; (c) layered, where the *Fabric* subsystem of a node provides low-level services to grid-specific components on that node; and (d) event-based, which is largely represented by event notifications such as job completion or heartbeats from *Resource* components to *Collective* components. These styles more accurately separate and capture the key interactions occurring across the subsystems of a grid system and,

as we will elaborate in Section 5, reduce the violations induced by the originally prescribed layered architectural style.

The client/server style accurately describes the interaction dependencies originating from *Application* components. *Application* components may submit grid jobs to the *Collective* components, request monitored data from *Collective* components, or issue metadata queries to help efficiently perform an operation. For instance, an *Application* component may need to request from a *Collective* component the location of *Resource* components that the *Application* wants to access directly. Once this information is received, the *Application* can directly connect to the needed resource to achieve better network performance.

The need for distributed *Resource* or *Collective* components to communicate with each other across nodes in the form of load balancing, data replication, etc. is an afterthought in the original grid. The new DSSA, on the other hand, captures this communication as peer-to-peer interactions. *Resource* components need to send metadata queries to *Collective* components to discover information about other *Resource* components. This location information is then used by a *Resource* component to make requests to other *Resource* components (e.g., one *Resource* component requesting portions of files from another in *Grid Datafarm*). *Collective* components also communicate with other *Collective* components in a peer-to-peer manner (e.g., in *Hadoop*, the component *JobTracker*, which is responsible for distributing computational jobs, calls *NameNode*, which is responsible for data distribution, to obtain data needed for planning tasks).

The event-based style naturally captures the need for components to send asynchronous messages across the network to achieve parallel execution. *Resource* components, in particular, may need to send messages to notify *Collective* components of service state changes (e.g., in the form of heartbeats and resource utilization data in *Alchemi*).

Finally, the layered style is retained solely on individual nodes where components can request services from the *Fabric* components of their local machines. For example, both *Hadoop* and *iRODS* have filesystem *Fabric* components that are used by *Collective* and *Resource* components.

4.3 Classification of Grid Systems

Unlike the original grid DSSA, our reference architecture describes all the components and services needed to create a grid. In particular, all the *Collective* and *Resource* components (with one exception, discussed below), plus an API for *Application* components to interface with these components, are required to create a complete grid – and in turn this work contributes further understanding into Grid and cloud APIs as identified by Rimal et al. [41]. These two kinds of components directly enable the concept of virtual organizations, which distinguishes grids from other types of distributed computing platforms (e.g., frameworks and middleware). Variations of the *Collective* and *Resource* components in the grid technologies we studied result in three grid categories: *computational*, *data*, and *auxiliary* grids (e.g., monitors and other grid supporting components). The computational and data grid categories denote complete grids, while the auxiliary grids category denotes systems that use and augment grids, but are not complete grids by themselves.

Computational and data grids are distinguished mostly by the instances of their *Resources* components and, to a lesser degree, by the types of *Collective* components they instantiate. *Resource* components of computational grids provide services for execution of parallel jobs or tasks (e.g., executors in *Alchemi*), while *Resource* components of data grids provide services for accepting operations to be performed on segments of a single conceptual repository (e.g., filesystem daemons in *Grid Datafarm*). All *Collective* component services depicted in Fig. 2 must be implemented in the case of computational grids (e.g., *Alchemi* and *Wings*). Data grids, on the other hand, do not have schedulers, but are highly reliant on a metadata repository for the entire grid system, as shown in Fig. 2.

Some grid systems claim to be both computational and data grids, e.g., *Gridbus Broker* and *iRODS*. In that case, *Resource* components must be capable of providing services either to perform operations on a storage repository or to execute a job or task. For example, *Gridbus Broker* achieves this mutability of *Resource* components by having them be wrappers around other grid systems such as *Alchemi*

or *Globus*, while *iRODS Resource* components provide "microservices" that can be workflow-related or related to data storage and retrieval.

All services of the *Collective* components in the new DSSA must be implemented in order for a technology to be classified as a complete hybrid (data and computational) grid. *Gridbus Broker* implements all such services. However, *iRODS* is not a full computational grid because it has no means of creating execution-independent workflow instances provided by planning and scheduling components.

The final grid category, auxiliary grids, does not contain complete grid systems. Auxiliary grids implement a small number of *Collective* components, but do not implement all the services described in Section 4.1. For example, *Ganglia* implements a monitoring component, contains no other *Collective* components, and uses other grid nodes as its *Resource* components.

5 Evaluation of the New Grid DSSA

Our suggested reference architecture eliminated 85 % of the architectural discrepancies described in Section 3 and shown in Table 2 by (1) identifying style-related interactions as first-class connectors, (2) removing the superfluous *Connectivity* layer, (3) explicitly addressing deployment, and (4) properly reclassifying components according to the new reference architecture's subsystem definitions.

A great majority of the upcalls and skipped-layer calls we discovered in the grid technologies (242 total as shown in Table 2) were rectified by identifying client-server and event-based styles as key parts of a grid architecture. Client-server interactions allow efficient communication of data between an *Application* component and a *Resource* component (e.g. *DFSClient* requesting data from a *DataNode* in *Hadoop* as depicted in Fig. 3). Likewise, *Resource* components acting as clients may request data from *Collective* components acting as servers. For example, any of the *iRODS Microservices* shown in Fig. 3 may request rule execution information or other metadata from the *Server Rule Engine*. This metadata provides *Resource* components with the necessary location information to facilitate connections to other *Resource* components. In the case of *iRODS*, most of the communication from *Resource* components to *Collective* components (note that these were most of the upcalls according to the original DSSA) results from requests for metadata.

While client-server interactions explained away some upcalls, event-based communication explained away others. Many upcalls originating from *Resource* components to *Collective* components represent event notifications in the form of registration events, heartbeats, monitoring events, and service state changes. Examples of such notifications include the *TaskTracker* component sending monitoring events to the *JobTracker* component in *Hadoop* (see Fig. 3) and heartbeat messages sent from *Resource* components to *Collective* components in *Alchemi*.

System deployment guidance provided by the new DSSA also plays an important role in reducing the number violations in the recovered architectures. In particular, some systems deploy *Collective* components along with *Resource* and *Fabric* components on the same node. *iRODS*, for example, replicates *Collective* components on each node of the system except for a single reliable node that houses the metadata repository. This deployment of *Collective* and *Resource* components on the same node allows the two types of components to share access to *Fabric* components. Consequently, the fact that *Fabric* components can be called by any local component eliminated all 51 discrepancies originally attributed to skipped layers in *iRODS*.

Even after shoehorning them into the new DSSA, the grid technologies still contained interactions prohibited by the various styles present in the DSSA (these correspond to upcalls and skipped layers in the original DSSA). However, the number of discrepancies was both much smaller than the original 242 (42 total) and we found no discrepancies in nine of the technologies. A recurring example of illegal interactions was a component in the *Fabric* subsystem invoking a component in one of the other subsystems. *iRODS*, with its highly interconnected architecture, was the biggest single culprit, with 13 such discrepancies (which was still significantly lower than the 86 it appeared to have when shoe-horned into the old DSSA). Another, somewhat less common example was a component in the *Application* subsystem

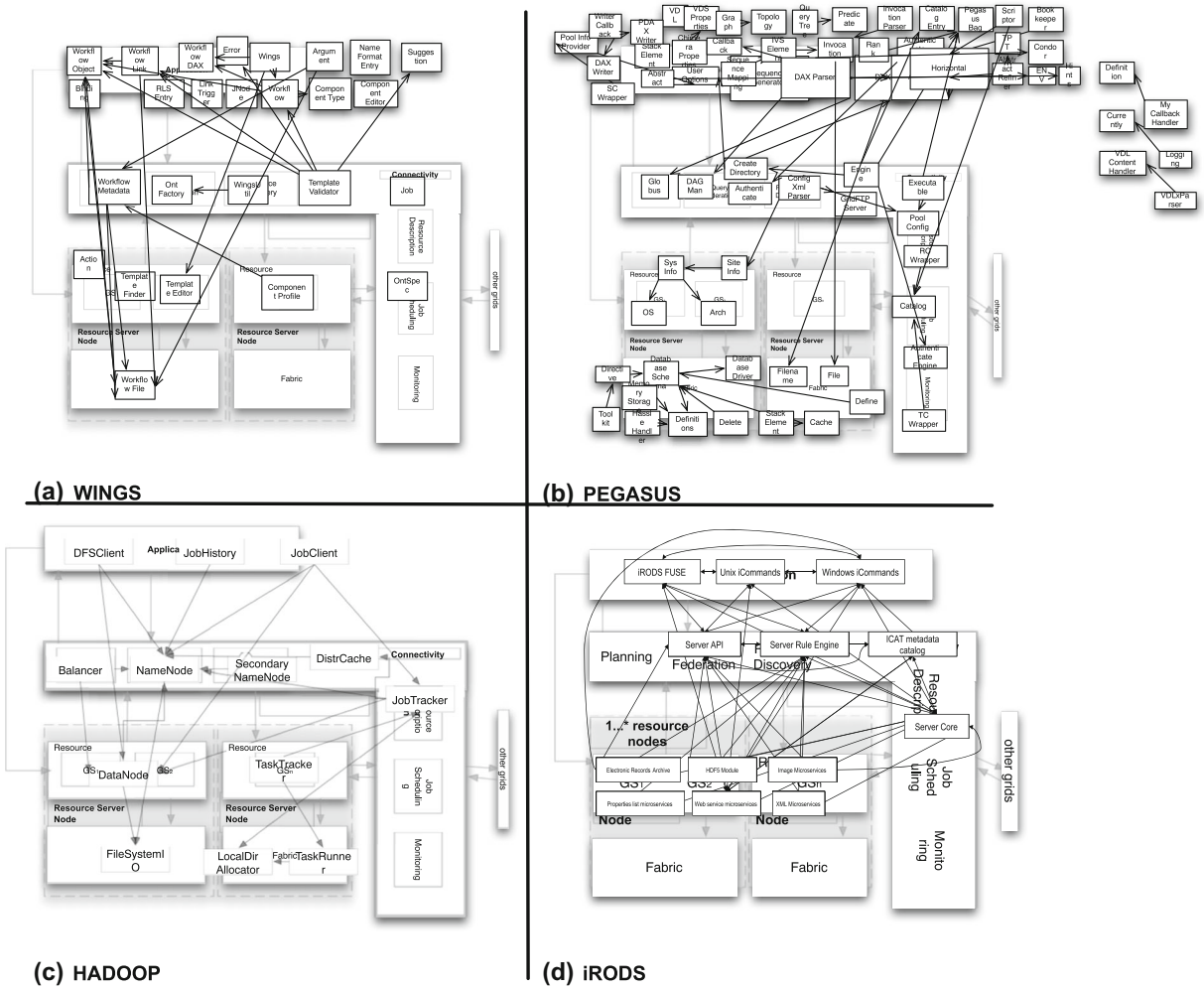

Fig. 3 Shoe-horning grid technologies into the new grid reference architecture

directly communicating with a *Fabric* component on another node. *Wings*, with 4 such interactions, and Jcgrid, with 6, accounted for all instances of this type of discrepancy.

Removing the improperly specified *Connectivity* layer of the original reference architecture eliminated all of the empty-layer and multi-layer component discrepancies identified in Table 2. This layer provided security and communication protocols used by nearly every other layer of the original reference architecture. In fact, a number of the recovered components from the eighteen analyzed technologies could be reasonably classified to be in both the *Connectivity* layer and another layer of the original DSSA due to the pervasive need for security and communication services. In our DSSA, such components were placed in the appropriate subsystem and their interactions made explicit.

Components that were originally placed only inside the *Connectivity* layer are easily reclassified in the new DSSA. For a great majority of the recovered grid architectures, only one or two components were shoe-horned into the *Connectivity* layer of the original DSSA (e.g., recall Fig. 1). These components contained a large number of *Collective* services and, thus, were typically placed in the *Collective* subsystem of the new DSSA (e.g. *Pegasus*'s *Authenticate* and*Authenticate Engine* in Fig. 3).

6 Conclusions

Our study of eighteen widely used grid technologies over the past five years suggested major deviations of their as-implemented architectures from that of the grid's widely cited "anatomy" and "physiology". These deviations ranged from minor undocumented components to significant deviations of the grid's layered reference architecture. Furthermore, the grid's architecture, as documented, was frequently ambiguous and under-specified. This suggested a need for another look at the grid's DSSA and possibly its significant refinement. We have motivated, described, and evaluated our refinement to the grid's DSSA. We have showed that our proposed DSSA captures the as-implemented architectures of grid technologies much more accurately (reducing their apparent structural violations by 85 %), and more closely matches the needs of the domain of grid computing (by describing key interaction styles and deployment characteristics missing from the grid's original specification). Our future work includes analyzing and refining the published grid requirements in light our new grid DSSA and applying our DSSA to additional grid platforms.

Our work represents a basis on which fundamental grid architectures and requirements can now be further defined and studied starting from a point that more closely matches most of the common grid and cloud-based systems in use today. In addition, the approach, data collected, and study evaluation metrics can also serve as an example for which future computing paradigms and architectures can be assessed.

We hope that the data and identified discrepancies from our study can also serve as a model for improvement for many of the grid and cloud systems studied and ultimately that the new grid DSSA can better describe for developers and users of the eighteen grid systems the interactions, uses, expected behaviors, structural elements, and evolution road for those and other similar grid and cloud-based computing platforms.

References

1. Foster, I. et al.: The physiology of the grid: An open grid services architecture for distributed systems integration, Globus research, work-in-progress (2002)
2. Kesselman, C. et al.: The anatomy of the grid: Enabling scalable virtual organizations. J. Supercomputing Applications, 1–25 (2001)
3. Crichton, D.J. et al.: A distributed information services architecture to support biomarker discovery in early detection of cancer. In: Proceedings of e-Science, p. 44 (2006)
4. Hughes, J.S. et al.: An ontology-based archive information model for the planetary science community. In: Proceedings Spaceops, Heidelberg, Germany (2008)
5. Bernholdt, D. et al.: The earth system grid: supporting the next generation of climate modeling research. Proc. IEEE **93**, 485–495 (2005)
6. Deelman, E. et al.: Grid-based galaxy morphology analysis for the national virtual observatory. In: Proceedings IEEE Conference on Supercomputing, Phoenix, AZ (2003)
7. Mattmann, C. et al.: A software architecture-based framework for highly distributed and data intensive scientific applications. In: Proceedings ICSE, Shanghai, China (2006)
8. Mattmann, C. et al.: GLIDE: a grid-based, lightweight, infrastructure for data-intensive environments. In: Proceedings EGC, Amsterdam, the Netherlands (2005)
9. Chervenak, A. et al.: The data grid: towards an architecture for the distributed management and analysis of large scientific data sets. J. Netw. Comput. Appl. **23**, 187–200 (2000)
10. Atkinson, M.P. et al.: Grid database access and integration: Requirements and functionalities, global grid forum GFD-I.13 (2003)
11. Badia, R. et al.: Use-cases and requirements for grid checkpoint and recovery, global grid forum GFD-I.92 (2007)
12. Bhatia, K.: Peer-to-peer requirements on the open grid services architecture framework, global grid forum GFD-I. 049 (2005)
13. Gamiel, K. et al.: Grid information retrieval requirements, global grid forum GFD-I.027 (2004)
14. Jha, S., Merzky, A.: A requirements analysis for a simple api for grid applications, global grid forum GFD-I.071 (2006)
15. Welch, V. et al.: OGSI authorization requirements, global grid forum GFD-I.067 (2006)
16. Tracz, W. et al.: Software development using domain-specific software architectures. ACM SEN, 27–38 (1995)
17. Mattmann, C. et al.: Unlocking the Grid. In: Proceedings CBSE, St. Louis, MO (2005)
18. Taylor, R.N.: Generalization from domain experience: The superior paradigm for software architecture research? In: Proceedings ISAW-2, San Francisco, CA (1996)
19. Taylor, R.N. et al.: Software architecture: Foundations, theory and practice. Wiley (2008)
20. Finkelstein, A. et al.: Relating requirements and architectures: a study of data grids. J. Grid Comput. **2**, 207–222 (2004)
21. The DataGrid Project: http://eu-datagrid.web.cern.ch/eu-datagrid/ (2006)
22. Venugopal, S. et al.: A taxonomy of data grids for distributed data sharing, management, and processing. ACM Comput. Surv. **38** (2006)

23. Yu, J., Buyya, R.: A taxonomy of scientific workflow systems for grid computing. SIGMOD Rec. Spec. Issue Sci. Workflows **34** (2005)
24. Kazman, R., Carriere, S.J.: Playing detective: reconstructing software architecture from available evidence. JASE **6**, 107–138 (1999)
25. Koschke, R.: Rekonstruktion von software-architekturen. Informatik-Forschung und Entwicklung **19**, 127–140 (2005)
26. Storey, M.-A.D. et al.: On designing an experiment to evaluate a reverse engineering tool. WCRE, Monterey, California (1996)
27. The portable bookshelf (PBS). Available online at http://www.turing.toronto.edu/pbs (2008)
28. Medvidovic, N., Jakobac, V.: Using software evolution to focus architectural recovery. JASE **13**, 225–256 (2006)
29. Littlefair, T.: An investigation into the use of softare code metrics in the industrial software development environment. Edith Cowan University, Ph.D, Dissertation (2001)
30. Wheeler, D.A.: More than a gigabuck: estimating GNU/Linux's size (2001). http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html
31. Grid systems software architecture - supplimentary Website, http://sunset.usc.edu/~softarch/grids/ (2008)
32. Krauter, K. et al.: A taxonomy and survey of grid resource management systems for distributed computing. Softw.: Pract. Experience **32**, 135–164 (2001)
33. Bowman, I.T. et al.: Linux as a case study: its extracted software architecture. ICSE, Los Angeles (1999)
34. Sim, S.E. et al.: On using a benchmark to evaluate C++ extractors. In: Proceedings *IWPC* (2005)
35. Storey, M.-A.: ShriMP views: an interactive environment for exploring multiple hierarchical views of a Java program. In: in ICSE 2001 Workshop on Software Visualization (2001)
36. IBM - rational software architect, http://www-01.ibm.com/software/awdtools/architect/swarchitect/ (2008)
37. Robbins, J., Redmiles, D.: Cognitive support, UML adherence, and XMI interchange in Argo/UML. Inf. Softw. Technol. **42**, 79–89 (2000)
38. Understand - source code analysis and metrics, http://www.scitools.com/products/understand/ (2008)
39. Doxygen (2008). http://www.stack.nl/~dimitri/doxygen/
40. Begeman, K.G. et al.: Merging grid technologies. J. Grid Comput. **8.2**, 199–221 (2010)
41. Rimal, Bhaskar, P., et al.: Architectural requirements for cloud computing systems: an enterprise cloud approach. J. Grid Comput. **9.1**, 3–26 (2011)
42. Montes, J., Sánchez, A., Pérez, M.S.: Riding out the storm: How to deal with the complexity of grid and cloud management. J. Grid Comput. **10.3**, 349–366 (2012)
43. Shamsi, J., Muhammad Ali K., Mohammad Ali Q.: Data-intensive cloud computing: requirements, expectations, challenges, and solutions. J. Grid Comput. **11.2**, 281–310 (2013)
44. Mattmann, C., Hart, A., Cinquini, L., Lazio, J., Khudikyan, S., Jones, D., Preston, R., Bennett, T., Butler, B., Harland, D., Glendenning, B., Kern, J., Robnett, J.: Scalable data mining, archiving and big data management for the next generation astronomical telescopes. In: Hu, W., Kaabouch, N. (eds.) Big Data Management, Technologies, and Applications, pp. 196–221. IGI Global (2013). http://www.igi-global.com/book/big-data-management-technologies-applications/
45. Mell, P., Grance, T.: Perspectives on cloud computing and standards. national institute of standards and technology (NIST), information technology laboratory (2009)
46. Kapil Bakshi, K.: Cisco cloud computing-data center strategy, architecture, and solutions. Point of view white paper for U.S. Public Sector (2009)
47. Mohan, T.S., Medvidovic, N., Mattmann, C.: Leveraging domain-specific software architectures for classifying cloud service abstractions. In: Proceedings of the Cloud Futures 2010: Advancing Research with Cloud Computing Workshop, Redmond, WA, April 8–9 (2010)
48. Stearley, J., Corwell, S., Lord, K.: Bridging the gaps: joining information sources with splunk. Proceedings of the 2010 workshop on Managing systems via log analysis and machine learning techniques. USENIX Association (2010)