Abstract. Grid computing is the computing infrastructure of the next century where unlimited hardware and software resources are delivered to user’s fingertips. Much of the power delivered by grid computing is realized through application software made readily available to its users. The process of application deployment and deliverance to the end-users though is perplexed with options and requirements not readily available to application deployers. In this paper, we present a new grid language called Application Specification Language (ASL), which allows application developers to describe their individual applications. Application descriptions include general application information, installation requirements, invocation requirements, as well as any additional hints or suggestions that are applicable for the application. We present an ASL schema and provide examples of language use showing how adoption of ASL alleviates some of the deployment and runtime difficulties.

Keywords: application specification, language, grid

1 Introduction

Grid Computing [1] has emerged as the next-generation computational infrastructure for developing and deploying distributed high performance computing (HPC) applications. Created grid environments are perplexed with heterogeneous and dynamic resource availability, inherently leading to resource and application dependencies as well as constraints [2]. Application deployment process in such environment is a non-trivial task and the user has to first determine what resources are available and then decide which is the most suitable resource for that particular application. It is important to differentiate the application development process from the application deployment process. Typically, HPC applications are developed using a specific programming language and parallel programming paradigm (e.g., compiler directive-based, threads, message-passing, combination of threads and message-passing) and often times the programming paradigm chosen decides the application deployment platform. If the application uses a shared-memory programming paradigm then the application can be only deployed on a shared memory system whereas an application developed using the message-passing paradigm can be
deployed on both distributed memory and shared memory systems. Furthermore, applications might require specific processor architecture, amount of memory, disk space, etc. to deliver desired performance and scalability. An application developer could describe these application requirements and dependencies using some sort of application descriptors and also add hints about various performance implications and space/time tradeoffs. For commercial software packages, information about licensing and subscription could also be provided by the descriptors. A resource broker could also take advantage of such descriptors during resource selection and job scheduling.

While the task of application deployment on a grid is quite challenging, the task of using various applications deployed on the grid is not any simpler (e.g., [3]). Users have a variety of applications to choose from for performing a specific task. Many applications exist that provide the same or similar functionality, yet often there are subtle differences among those such as programming paradigms used (e.g., shared memory, distributed memory), algorithms employed (e.g., search, sort methods), resource requirements (e.g., memory size, number of processors), numerical accuracy, scalability, and performance. While these differences are interesting to a domain-expert, for an end user who is interested in getting a problem solved with a particular quality of service (QoS) requirement (say, find the most accurate solution for this problem, or find the fastest solution to this problem) there are too many options to choose from. In order to make the best possible application selection that matches users QoS requirement, a typical user would require significant domain expertise, HPC expertise, as well as application expertise.

To address these goals of user accessibility, there is a need to standardize and simplify the process of application deployment and use on the grid. In this paper, as a step in that direction, we present a new language specification along the lines of JSDL [4] called the Application Specification Language (ASL) that can be used to describe details of a given application. ASL allows an individual application to be represented in heterogeneous world of grid computing by capturing its purpose, functionality, and options. Using ASL, application descriptions can be made available for immediate use or further advancements among applications such as application deployers, automated interface generators, job schedulers, and application-specific on-demand help provisioning. ASL can also be used to describe how an application is compared and/or combined with other matching services and software, thus also supporting idea of workflow systems. This ability to specify the composition of services can facilitate the creation of new and added functionality, as well as enabling further advancement of existing tools that can take advantage of the provided information.

The rest of the paper is organized as follows. Section 2 motivates the problem and introduces relevant background information. Sections 3 and 4, respectively, present description and structure of the presented language. Section 5 provides examples of use and future work while Section 6 summarizes the paper.

2 Motivation and Background

A goal of an application developer is to see their solution adopted quickly by a large group of end-users. However, the typical path to adoption requires frequent
interaction with end-users to address numerous questions, such as: how to install the application, how to invoke it, the purpose and function of the available options and arguments, as well as how to improve performance for a specific platform configuration. Through the lifetime of the application, additional documentation is created to address such common questions. However, with a new version of the application, much of this work needs to be discarded and redone. Avoiding this pipeline of events is challenging, but through gradual and systematic adoption of the technique proposed in this paper, it is possible to provide more automated support.

Computing has evolved from a user-centric context (where every detail of code execution needed human attentiveness and interaction) to a global service-driven view (such as Web Services, where complex, data dependent, goal seeking interactions and computations of independently developed components can be achieved with absolutely no human intervention). From this progression, we observe the common approach where automatic communication between programs is enabled. This is generally accomplished through definition of standard languages that specify protocols observed between applications. The following languages represent efforts within the grid community to automate specific deployment tasks:

- **Job Submission Description Language (JSDL)** [4] – a completed standard from the Job Submission Working Group [5] within the Open Grid Forum (OGF) [6]. The JSDL is a specification of an abstract and independent language used for describing requirements of computational jobs in grid environments. It contains a vocabulary and normative XML schema that builds on the idea of standardizing a language to accommodate a variety of job management systems, alleviating the problem of heterogeneity in the grid. By having a standard language available, a job submission description can enable diverse job management systems to communicate and complement job description in a more simplified manner.

- **Resource Specification Language (RSL)** [7] – RSL preceded JSDL but still provides a common interchange language to describe resources and jobs to run on them. It is a language developed by the Globus Toolkit Project [8] and is represented by various `<name, value>` pairs that are used to perform complex resource descriptions in cooperation with other components in the system.

- **Resource Description Language (RDL)** [4] – a language that is part of the JSDL standard document for describing underlying grid resources in terms of CPU speed, number of CPUs, and main memory. Even though not yet realized, the concept of this language has been propagated through other tools such as Condor and ClassAds [9] and the necessary information (e.g., current resource status) can be obtained from MDS [10].

The above-mentioned languages have been developed to enable standardized exchange of information between grid resources and provide support for direct and concrete communication between these resources. By defining documents specified by these language constraints, one can rely on automated negotiation during the job submission process without regard for the heterogeneity of the underlying hardware and software. When mapping the above-mentioned languages to individual grid user categories (i.e., end-users, resource owners, application developers and deployers), the JSDL and RSL map most favorably to the end-user category (where the user is mainly interested in adopting the grid as a pool of resources). The JSDL and RSL enable users to limit and perform resource selection. At the same time, RDL can be
classified as a language for allowing resource owners to describe the capabilities of their resources and advertise that information for wider use through mechanisms such as MDS and Ganglia. What is evidently missing is the need to support the application developer. Once an application is developed, there is no standardized way to publicize the name and capabilities of the application in a manner that can be accessed by other tools (WSDL [11] is the most relevant technology but it focuses on web service and does not capture necessary level of detail at the application level). Rather, the application developer and the end-user are forced to interact in an interrogative manner, perhaps using a wiki or similar tools (e.g., readme files, FAQ) to advertise application’s availability.

By supporting a method for capturing the core purpose of the application, requirements, and options, the end-user is provided with specific information that describes the application. After successful installation, the second most important feature enabling application use is the interface that the application provides to its users. With respect to grid applications, the most appropriate way to interact with an application is through a web-based interface that requires no local installation of the application. A web-based interaction may also provide special tools and knowledge to access available resources [12, 13]. By providing a default standardized interface to the given application automatically, a resource owner may reuse the interface rather than implementing their own. The benefit for the end-user is that the interface stays constant across different providers. An additional benefit is a reduction of possible errors in interface generation originating from the resource provider due to possible misunderstanding or lack of application knowledge.

The Pasteur Institute Software Environment (PISE) [14] is an example of previous work where this idea has been adopted in practice. PISE is a transformation tool that receives as input a PISE-DTD compliant XML document and interprets the document to create any of the suite of interfaces ranging from HTML to CGI and IPSH [14]. A scalable core is also provided by PISE that can be extended to add additional interface interpreters as needed. PISE currently contains a database of over 200 XML documents corresponding to interfaces for various applications that are primarily focused on bioinformatics. By leveraging ideas and technologies such as those provided by PISE, many of the accidental complexities associated with grid application deployment could be removed.

3 Description of the Language

To address the challenges raised in the previous sections, we present a new language called the Application Specification Language (ASL). Comparable to packaging tools in Linux environment (e.g., APT [15], OPIUM [16]), ASL is a new approach toward application specification in the grid that focuses on the needs of grid application developers and their respective grid applications and captures essential application information. Through standardized protocols, tools can thus access and process ASL document extracting needed information about an application. ASL is a language for describing any application’s requirements, attributes, and options. The ASL directly supports the ability to capture application-specific information that is

27
not necessarily found in the general pool of available description tags. Using ASL, factors such as software and hardware requirements, data constraints, and algorithm complexity can be provided to a user. As can be seen in Fig 1, the ASL may be composed with other groups of established grid languages (e.g., JSDL/RSL, RDL). The interactions implied in the triangle connect all perspectives and mentioned user categories of a grid environment, which enable communication to take place over well-designed paths to facilitate further communication, refinement, contract creation and the possibility of higher QoS for all participants.

![Fig 1. ASL-RDL-JSDL/RSL triangle showing direct communication paths between corresponding user categories.](image)

ASL is applicable before and during installation, during job scheduling, during job execution, and even after the job has completed. It can be complemented and modified as knowledge about an application increases. The ASL can be used with legacy applications (requiring adaptation), or with newly developed applications designed specifically for the grid (often called ‘Smart Applications’). By providing a standardized way to describe application requirements, the ASL enables an automated capability to compare applications (e.g., versioning, installation/runtime requirements, ownership, invocation procedure). Without ASL, such comparisons are very hard to perform manually because of their subjectivity. Such comparisons can be useful in numerous cases, such as application scheduling and software cost estimation [17].

In essence, ASL is an extended application version of RSL. It provides a set of specialized tags used to capture application-specific details. By providing the appropriate set of tags, ASL enables application comparison and interface generation. Because every grid application is custom built to meet a certain need, the implementation details may be difficult to describe. Many of the options available during application specification often require significant human intervention as well as use of human language descriptions that cannot be modeled and captured by a general purpose computer language. Providing a standardized set of tags to capture information about an application in a concise and precise manner is difficult. The requirements imposed when selecting a given set of tags must focus on capturing the core set of characteristics describing any application and then providing an extended set of tags that allow unique application components to be specified. The starting point in defining this set of tags considered existing languages such as JSDL and RSL, which capture job submission requirements that map onto resource and application requirements. Examples of such tags include numerical values (e.g., CPU speed and amount of main memory required), as well as a predefined set of values (e.g., operating system and CPU architecture type).
Additional tags, although not required for application invocation (e.g., max number of CPUs an application scales to), were created by a systematic analysis of characteristics that describe an application and are based on our experience with grid-enabling applications on the UABGrid and SURAgird. Many of these tags are simple in nature allowing the definition of a range of valid values that can be used to validate data entered by the end-user. The more difficult set of requirements deals with values that are dependent on each other, but can be viewed individually as containers of simple values. Thus, these tags were organized in groups where sub-elements define individual pieces of the larger component. An example is the operating system requirement. An application may be suitable for many operating systems as well as different versions of an individual operating system. Thus, creating higher level elements that contain equivalent sub-elements allows different version dependencies to be specified.

One challenge with adopting ASL, especially when viewed from the perspective of a developer creating the ASL document, is the requirement of the document syntax to be specified correctly (i.e., equivalent tags may have a different meaning when placed in different element groups). The most difficult part of describing an application concisely occurs with tags that cannot be constrained to a set of predefined values (e.g., tags that represent a human readable text string, such as copyright policy). The obvious impediment with such tags is the lack of precision needed for formal interpretation. However, the additional information provided within these tags can benefit end-user understandability of the application. The use of the tags for all types of descriptors (e.g., simple, complex groups and natural language) helps to partition the entire document and provide guided help for natural language descriptors. A further benefit of such tags is the possibility of developing additional interpreters to generate application web documentation automatically.

The completed ASL document consists of several parts (discussed in subsequent sections) each focusing on a particular portion of an application deployment lifecycle. With the blend of the formal tags (i.e., computer readable) and informal tags (i.e., end-user understandable), the ASL assists in application description from different perspectives and provides user support in multiple formats. Examples of interpretation include an application description web page with installation and invocation instructions, script generation for automated application installation, as well as optimal system requirements for job submission.

4 Structure of the language

An ASL document consists of only four distinct yet related sections that are described in XML. By dividing the document into these distinct sections, an ASL document is modular and allows for easier initial generation and subsequent modification. With the use of appropriate tools, each section of the document can have its own permissions, which allows the document to be modified independently and securely. As the application receives a wider user base, additional information may become available from its users. As a result of multiple executions of an application, additional information can be gathered, such as profiles of application
performance, unexpected behavior, or suggestions for future enhancements. Beyond
the collection of application information, the segregation of the document into
appropriate sections allows for shorter search times among users allowing them to
focus on sections of the document of most interest. To provide segregation of
information collection and retrieval, an ASL document consists of application name
and description, installation requirements, job invocation requirements, and hints.

The sections are not directly connected to each other, but the data is stored only
once per ASL document. Because of this, inadvertent references to information
provided in other parts of the document may exist. These sections are correspondingly
mapped to XML with appropriate tags. Each of the sections is described below and
the schema is provided for the given section.

4.1 Application name and description

The application name and description section contains the most basic information
about an application and acts as the application identification component. It specifies
the name and version of the application as can be found in an application repository.
This section also contains sub-elements such as the application description describing
the application in a human readable format. The description identifies the problem the
application solves and maps the application to an application category. The
application category element is limited to a predefined set of values as described
below. It is intended to offer better understanding of the application deployment
process on the grid and it is essential in classifying different types of applications
deployed in such environment:

1. Sequential applications – Traditional applications developed to execute on a
single node machine, where grid provides redundancy, fail-safe capability, and
excess capacity (e.g., more memory, disk, or faster CPUs).

2. Parametric sweeps - Multiple copies of sequential jobs using different input
datasets or parameters. These applications are often submitted independently by a
single user in an effort to reduce overall task execution time. Benefits of using
the grid are the same as sequential applications with the addition of multiple
instance coordination performed by grid tools and middleware.

3. Master-Worker applications – Master-worker or bag-of-tasks or embarrassingly
parallel model, where a master process distributes work (either statically or
dynamically) to a set of worker processes and aggregates the results at the end.
The main differences between parametric sweeps and master-worker applications
is that the individual tasks do not have to be executing the same code, but a
workflow system can be in place with the master-worker model possibly
delivering a more complex application functionality. The coordination between
the worker nodes and task assignment must be handled by the master process.

4. All-Worker applications – Similar to the master-worker model, except that each
process, including the master, share the workload and data is exchanged between
individual processes in some pattern (point-to-point or group communication).

5. Loosely coupled parallel applications – Parallel applications (e.g., coupled fluid
flow and wave models) that exchange data occasionally through files during
execution (e.g., beginning and ending of an outer iteration).
6. **Tightly coupled parallel applications** – A single Message Passing Interface (MPI) [18] application distributed across multiple systems sharing data during the execution, possibly at frequent intervals, through passed messages. It requires interoperability between MPI libraries or a MPI library such as MPICH-G2 [19].

7. **Workflow applications** – A model connecting many individual applications executing at different geographically distributed locations, which are chained together to perform a complex simulation. For an application to be classified as a workflow application, additional information is needed to identify dependencies with other applications in terms of input and output data streams.

---

Some additional considerations must be made when selecting the application category for an ASL specification. Applications that belong to categories (1) and (2) can be distributed and scheduled across any available computational resource on the grid because there is no synchronization or coordination required between individual tasks. However, applications in categories (3) and (4) must be scheduled on a single computational resource and cannot be distributed across multiple resources. This does not imply that the application may not use additional, distributed resources. If an application has been developed specifically for the grid, it can utilize middleware components to enable cross-resource task execution. In that case, task scheduling is the responsibility of the application itself because it would be deployed on a dedicated resource. Applications in categories (5) and (6) expect that the individual applications are distributed and assume that the individual tasks are scheduled to execute at the same time (through advanced reservation or mutual agreement with the resource providers). Workflow applications (category 7) assume that the scheduler can trigger the execution of one or more applications as described by the workflow. Because most of the existing schedulers [20-23] do not handle advance reservation, the use of workflow applications is limited and intended mostly for future grid developments.

The remaining elements in this section are illustrated in Fig 2. **Category** element set has a predefined set of values a user must choose from (*i.e.*, as provided earlier in this section). The remaining elements found in this schema section do not have their values predefined, but can be defined by the person creating the document enabling desired application description. Among these tags are some that are required, while others are optional.
4.2 Installation requirements

The installation requirements section of an ASL document contains a set of required elements that describe the pre-installation requirements as well as the installation procedure. Some of the examples of this type of element set include minimum processor speed, processor architecture, minimum amount of disk needed for installation of the application, libraries, applications required for the installation procedure (e.g., compilers, (un)packaging tools), licenses needed for application installation, network requirements, and required amount of disk space for the installation. The tags used are simple declarations that specify the value of a predefined type (e.g., string, integer). Even though this model may result in unnecessary inconsistencies between application descriptions (i.e., resulting from lack of standard vocabulary between persons composing ASL documents), we believe at this stage of ASL development and definition this variability is necessary to allow for the correct words to be selected from a constrained set of choices. The full schema of the installation requirements section is given in Fig 3.

![Fig 3. Application Installation section schema.](image)

Among the elements defined in the installation category are SoftwareDependencies and Applications Required tags. The information these tags contain is intended strictly to be used during the installation procedure. The SoftwareDependencies tag refers to any other software that will be needed for the application execution. This can be viewed as a prerequisite for the installation; i.e., in case software packages declared within this tag are not installed, the application cannot be expected to execute. Examples of such software dependencies would include Perl [24] with certain libraries and Postgres database [25]. With respect to installation, the ApplicationsRequired tag refers to other complete applications required to perform the installation. These applications may be invoked during the installation procedure, such as un-packaging tools (e.g., Ant), and installation tools (e.g., make).
4.3 Job invocation requirements

The job invocation requirements section focuses on providing a user with the information needed to execute the application. Starting with the executable name, it also provides the available switches and minimum hardware requirements, as well as allows the developer to specify the number of input and output files with examples of their respective formats. This section does not represent a duplication of effort found in JSDL/RSL, but it is alternatively used to specify requirements for the entire application. Such specification is needed not only when executing a single job, but to describe the available options and how to use them. Rather than specifying exact input files and other job-specific parameters, the category defines application requirements, such as: the required input files, required format of those files, any output files and corresponding format, libraries required to invoke the application, and licenses needed to run it. This capability can be viewed as a more detailed version of man pages in UNIX. This category allows the developer to be shielded through a contract-like document enforcing application requirements before correct execution.

The majority of the application description is provided in this section of the ASL document, so it is natural for a set of tools to be based on this category. An example tool is a translator that formats the appropriate information into a web page allowing the information to be read through a browser, or a correct and application-specific job submission interface. Another example tool serves as a data verification tool that ensures input files are in the correct format. The complete schema is for this section is too long to be included in this paper and is thus available at http://www.cis.uab.edu/ccl/index.php/ASL. Similar to the installation section, the application invocation section has elements SoftwareDependencies and RequiresApplications. In this context, software dependencies refer to any software packages that are necessary and will be used as part of the application during its execution. An example would include a call to a Perl module. The description of the application requirements tag is similar to the description from the application installation section of ASL, where it specifies any other applications that may be invoked during this application’s execution. This tag can be used to specify requirements for a workflow, even though any further enforcement and coordination during execution must be done by given application.

4.4 Hints

Due to the inherent variability of applications, information describing an application may not be adequately captured in the preceding sections of ASL due to non-compliance and uniqueness of the application. Also, the succinctness of available options in ASL tags or already existing data may prevent additional and possibly more complete application information to be stored. In order to accommodate for these possible shortcomings, there is an additional section found in ASL documents, entitled Hints. This section contains instructions and comments, mostly in natural language, providing additional application information.

The purpose of this information is to allow detailed descriptions for areas of high application complexity, either for application users or other developers who may use
this application as a base for development. Another important goal behind this section and its element set is that it can be accessed and edited by a wide user group. Performance information may be stored in this section to specify the optimal parameters on particular hardware architecture. Depending on application type (e.g., sequential, embarrassingly parallel, MPI), certain input parameters (e.g., size or format of input file, number of processors) may alter application performance and thus information found here could be useful for the resource owner, end-user and even the scheduler developer. By giving permissions to a wide range of users, known bugs as well as suggestions for future advancements can be documented. A large portion of its use can be found in troubleshooting an application where expected errors can be explained. Fig 4 shows the current Hints section schema.

Fig 4. Hints section schema.

5 Examples of Language Use

This section provides examples for using ASL within selected projects. Projects were selected based on the potential merit of adopting ASL as well as their applicability to this project. Second subsection considers additional, long-term future work.

5.1 Integration of ASL with current applications and projects

Previous section provided an overview of ASL and its schema. We have created ASL documents using this schema for a range of applications, some of which are available on the earlier noted website while others are under development due to their size and complexity. Three applications whose ASL documents have been fully completed were selected from three different application categories and were used to show applicability of ASL for those application categories. The first application is a sequential application implemented in Perl called *QuerySplit* that performs segmentation of the BLAST [27] input query file. The associated ASL document aims at capturing general purpose of the application and its options during execution. *QuerySplit* application is used by the second chosen application called *Dynamic BLAST* – a master-worker type application [17]. Beyond capturing application description, as needed for this type of an application, associated ASL document shows its capability to capture application dependencies. The third fully described
application for this paper is a parallel implementation of matrix-matrix multiplication using Cannon’s algorithm [28]. Associated ASL document shows ASL’s ability to capture network requirements imposed by the application as well as application’s scalability. Further applications whose ASL descriptions are under development include HPL, BLAST, NAMD, and MatLab.

These examples show sample use for various application categories and, more specifically, various applications. The focus of the examples is to show capability of ASL to capture needed information as well as provide concrete samples for its creation. The use, possibilities, extensions, and limitation of the language will have to be explored over time as its use become more wide-spread and tools emerge. The remainder of this section points at several projects, available today, with different goals and descriptions of how adoption of ASL would improve them.

GridBench [29] is a project that focuses on providing a core set of benchmarks that characterize grid resources. Use of such benchmarks allows prediction of performance and scalability of applications on desired systems. The proposed framework supports collecting, archiving, and publishing of collected data. The goal of GridBench is to provide a collection of kernels representative of applications and application categories that can be used to benchmark and characterize components that affect performance of applications and resources, allowing comparisons to be made. Use of ASL in this context to both, describe applications and subsequently include performance information into ASL document enables automated sharing of performance associated with individual application and resources. Additional project similar to GridBench that could find use of ASL include STAPL [30], Prophesy [31], and application performance predictors such as [32].

GridWay [33] aims at supporting the “submit and forget” ideology where user is abstracted from the grid middleware details by supporting easier and more efficient job execution on the grid. GridWay works on top of the Globus Toolkit and provides a job manager-like interface for its user where a job can be submitted, its status checked, and results retrieved. Use of ASL in this context is broad and could include enablement of application-specific information to be presented to the user when selecting among several application to execute, matching of performance of selected application to available resources, as well as fully automating application invocation parameters and options. Additional examples of grid resource brokers that could benefit from ASL in similar ways as GridWay include Nimrod/G and its parametric job description language [34] and CondorG [35] by incorporating ASL into ClassAds.

The dynamic nature of workflow systems demands the need for grid services to be automatically generated according to the execution environment and resource available for execution. GridDeploy [36] is a toolkit for automatically deploying applications as Grid services, and providing the necessary infrastructure for other applications to invoke them. Integration of ASL into GridDeploy would enable more streamlined generation of needed services because ASL provides needed descriptions of applications and corresponding performance on various resources. Use of standardized protocols to combine information available in ASL with information from JSDL and MDS can enable user and job oriented service composition.
5.2 Future tools and uses of ASL

This section is intended to serve as a window into the future of where ASL could be applied and what benefits could be expected. Before realization of such ideas and associated benefits, much more work needs to be done. This work is largely required as a community effort, where ASL would be accepted by the community and thus further developed, tweaked, and finalized. Authors realize that there is a long road before adoption of ASL by the community, and are thus interested in cooperation with OGF on building ASL into a standard. The standardization process would resolve many of the open questions regarding concrete definition of selected tags, their scope, semantics, as well as overall ASL adoption and applicability. As a step in this direction, a tool enabling generation of ASL documents from a domain specific aspect is under development by the authors [26].

The remainder of this section points at additional research ideas and advancements not directly related to ASL development, but rather pointing at tools that could make use of ASL. Such developments would in turn enable further developments and promotion of ASL:

- **Application Information Services (AIS)**, which represent a suite of services where registration, discovery, and monitoring of an application could take place at the level of a VO. Through a set of interfaces, the users would be able to discover services and applications, and obtain help documentation.
- **Automated HTML application help generator**, an application to interpret ASL documents and generate either static or dynamically generate application-specific and related information.
- **GridRPM**, an application taking advantage of installation and dependency information available through the mesh of ASL documents (as available in AIS) and automatically construct descriptor files before proceeding with automatic application installation and deployment.
- **Grid scheduler applications** could utilize the ASL information to perform application-specific scheduling without the need to keep historical application.
- **Accounting and software license accountability** is an area of grid advancing quickly in response to high demand of grid services from industry providing a vast number of options where relevant information can be stored into ASL documents and later used through automated billing and license verification.
- **Automated application dependency graph** leading to automated application updating similar to application packaging tools available in Linux.

6 Summary

This paper introduced a new language for grid computing environments that was composed with goal of describing individual applications in such environment. The language, called Application Specification Language (ASL), is an XML schema based language that captures application information enabling application developers to provide very specific and targeted goals for their application. Such targeted information enables specific and automated communication that complements already
existing grid languages (e.g., JSDL, RDL) in new ways. Information is captured in ASL documents in four distinct sections. Each of the sections provides scoped information specific to the application description, the installation procedure, the invocation routine and options, and any application specific hints or suggestions users may welcome. Division of the document in such simple but effective sections allows for easier creation, modification, copyright, as well as use. Finally, suggestions for further work were presented identifying the need to standardize current schema and develop support tools that use ASL in current as well as new contexts.

References

32. Smith, W., I. Foster, and V. Taylor. Predicting Application Run Times Using Historical Information. in Workshop on Job Scheduling Strategies for Parallel Processing. 1998: Springer-Verlag.