Specification and Implementation of the SAGA Python Language Binding

Computer Science Master Thesis

P.F.A. van Zoolingen
1284657, pzn400@few.vu.nl

December 19, 2008

Abstract

This thesis describes how we created a Python language binding for SAGA, the Simple API for Grid Applications, and how we implemented the language binding on top of the Java reference implementation. Using this functionality, Python programmers can use SAGA to program grid-aware applications and shield themselves from all the details which come with grids. The language binding and its implementation add to the adoption of SAGA in a world of with many different APIs and middleware layers. The specification is a set of classes and methods called PySaga and the implementation on top of the Java reference implementation is called JySaga. PySaga is available in source code and in HTML documentation. JySaga is has been tested and runs on top of the SAGA Java reference implementation. JySaga is available as source code, together with the unit tests.

Contents

1 Introduction

This section contains the introduction to this thesis. It is written in both English and Dutch.

1.1 Introduction English

SAGA stands for Simple API for Grid Applications [10], and was developed to offer users a simple tool to program applications for heterogeneous grids. These grids often consist of different types of hardware, operating systems and middleware software and are hard to program. SAGA is developed to be independent of any underlying hardware or software and it shields the user from all the details, and lets him focus on programming grid aware applications themselves.

To use the SAGA API, the functionality described by SAGA [9] has to be implemented by another piece of software: the SAGA implementation. Currently, there are two different reference implementations which are programmed in the programming languages Java [27] and C++ [17]. In a general sense, only Java and C++ applications can use the SAGA implementations to access the grid in an easy way. This thesis describes how we added another language to that list, namely Python [22]. Python is partially supported by the C++ reference implementation, but there is no specific Python language binding available. A language binding is a set of classes and methods which describes the SAGA functionality in a Python-specific way, independent of the chosen reference implementation. During the course of this master project we have specified the Python language binding and implemented the language binding for the Java reference implementation.

This thesis is divided into different parts. First we will describe and explain in Section 3 what SAGA is, where it comes from and how it is implemented. Then we will continue in Section 4 with a description of Python and a special implementation of Python called Jython, followed by the specification of the language binding, Section 5 and its implementation in Section 6. After that we will describe the testing processes of the language binding in Section 7, the discussion of the project in Section 7.4, the future work is Section 8 and we will conclude with the conclusion in Section 9.

1.2 Introduction Dutch

SAGA staat voor Simpele API voor Grid Applicaties en is ontwikkeld als een simpel stuk gereedschap om het programmeren op heterogene grids te vergemakkelijken. Dit soort grids bestaan vaak uit verschillende hardware, besturingssystemen en middleware software en het is vaak lastig om hier grid applicaties voor te programmeren. SAGA is ontwikkeld als een aanspreekpunt voor het grid, onafhankelijk van de onderliggende hard- en software. Tevens houdt de API de programmeur weg bij de onderliggende details, die per platform zeer kunnen verschillen. De programmeur kan zich hierdoor bezighouden met het programmeren van een hogere abstractie niveau voor zijn applicatie.

Om de API te kunnen gebruiken moet de functionaliteit beschreven door SAGA geïmplementeerd worden door andere software, ook wel de SAGA implementatie genoemd. Momenteel bestaan twee verschillende referentie implementaties die gemaakt zijn in de programmeertalen Java en C++. Dit houdt globaal in dat het alleen in de talen C++ en Java mogelijk is om een applicatie te programmeren die door middel van SAGA het grid te gebruikt. Deze master thesis beschrijft hoe daar een derde taal aan toe is gevoegd, namelijk Python. Python word op dit moment al deels ondersteund door de C++ referentie implementatie, maar er is nog geen Python 'language binding' gespecificeerd die de syntax voor Python applicaties vastlegt. De language binding specificeerd een set van klassen en methodes die de SAGA functionaliteit beschrijft in een Python specifieke manier, onafhankelijk van de onderliggende referentie implementatie. Tijdens mijn master project heb ik een Python language binding voor SAGA gespecificeerd en

geïmplementeerd bovenop de Java referentie implementatie. Deze implementatie zou in theorie moeten werken op elke Java implementatie van SAGA.

Deze thesis is onderverdeeld in verschillende delen. Eerst zal ik uitleggen wat SAGA is, waar het vandaan komt en hoe het geïmplementeerd is. Ik zal doorgaan met een beschrijving van Python en een specifieke implementatie van Python genaamd Jython, gevolgd door de specificatie van de language binding en zijn implementatie. Daarna zal ik het testen van de language binding bespreken, de discussie van het project, het mogelijke vervolg onderzoek na dit project en besluiten met de conclusie.

2 List of Frequently Used Terms

API: Application Programming Interface. A set of variables, methods and classes that is offered by an operating system or software library to support requests made by computer programs.

Grid: A collection of interconnected computers consisting of different hardware, placed in different locations and belonging to different organizations.

Grid-aware: Applications which are grid aware are designed to run on a grid and use the possibilities of the grid, such as distributing workload between available nodes in the grid.

Language Binding: An API in a specific programming language which gives access to a library or service

SAGA reference implementation: First working software that implements the functionality described by SAGA. New applications can link to this software and call methods described in SAGA to use the grid.

SAGA: Simple API for Grid Applications

3 SAGA

In this section we will describe what SAGA is, how it was created and the packages it consists of. In the last part we will discuss the Java and C++ reference implementations and how they work.

3.1 Short History of SAGA

SAGA stands for Simple API for Grid Applications. SAGA came as an idea in a time when multiple middleware projects and application groups were looking for higher-level programming abstractions and the simplification of programming for the grid [10]. A SAGA research group (SAGA-RG) was founded within the Global Grid Forum (GGF), which later merged into the Open Grid Forum (OGF). The aim of the group has been to identify a set of basic grid operations and derive a simple, consistent API, which eases the development of applications that make use of grid technologies.

To poll the needs of users, the research group sent out a call for use cases. In these use cases users described many subjects such as their application area, the desired look and feel of the API, and resource, performance and security considerations. The majority of use cases which were returned came from scientific users [14], which probably biased SAGA in the analysis of the use cases towards scientific applications. In this analysis, the research group focused on the identification of the SAGA API scope, on the level of abstraction wanted and needed by the application programmers. Non-functional requirements and requirements from other projects, such as GAT [3] and CoG [30] were also considered.

With 24 use cases available, the requirements from the users could be distilled [15]. A design team was formed to use these requirements to design and develop the API. A few general design issues were considered and agreed upon.

- The API would be designed and developed in an object-oriented manner using a language-neutral representation.
- Grid subsystems should be specified independently from each other to allow independent development and implementation of parts of the API.
- Sessions and security should be an essential part of SAGA since applications often run across administrative domains and security boundaries.
- Data management, like remote file access and replica catalogs are an important part of grid applications and should therefore be part of SAGA.
- Remote jobs and asynchronous operations are a common requirement for grid applications and must be supported in the API.
- Asynchronicity is preferred to be handled by a polling mechanism rather than a subscribe/listen mechanism to simplify implementations in non multi-threaded environments.
- SAGA should support inter-process communication as a stream concept, similar to BSD sockets.

Ultimately, the purpose of SAGA is to provide an simple API that can be used with much less effort compared to the vanilla interfaces of existing grid middleware. A guiding principle for achieving this simplicity is the 80/20 rule: serve 80% of the use cases with 20% of the effort needed for serving 100 % of all possible requirements and to provide a standardized, common interface across various grid middleware systems and their versions.

After determining the requirements, a so-called SAGA Strawman API was developed to accommodate the requirements and after some iterations the SAGA API was published in January 2008 [9]. SAGA is described in a document called *"A Simple API for Grid Applications (SAGA)"* or GFD.90. GFD.90 specifies the core components of SAGA. It has formed the basis of specification of the Python language binding, which will be explained in Section 5, and the reference implementations. It is aimed at implementors of the API and not directly at end users. The implementors of the reference implementations can supply the end users with the documentation and specific language bindings.

3.2 SAGA API

SAGA is divided into two parts. The first part is the Look and Feel part which contains the base classes and interfaces. The second part is the API part which represents explicit entities and actions of some backend system.

3.2.1 Look and Feel

The SAGA Look & Feel is defined by a number of classes and interfaces which together ensure the non-functional properties of the SAGA API. Non-functional requirements are requirements that specify criteria that can be used to judge the operation of a system, rather than specific behaviors.

The interfaces and classes from the Look and Feel are intended to be used by the functional SAGA API packages

Error The Error package contains all the exceptions which can raised or thrown by SAGA API calls. GFD.90 also describes an `error_handler` which allows a user of the API to query for the latest error associated with a SAGA object. Error handlers should not be included in language bindings of languages which have exception handling capabilities of their own, such as Python.

Object The `Object` package provides mechanisms which are needed by all SAGA objects, such as cloning and getting the type, ID and Session of the object. The `Object` class is also called the *base object*

URL The `URL` object is used to reference local and remote resources. Using a separate `URL` object simplifies the construction, parsing and checking of URLs in applications and unifies the signatures of SAGA method calls that accept URLs.

Buffer SAGA has a generic buffer object that is designed as a container for data. `Buffer` is used in combination with a number of SAGA calls that perform byte-level I/O operations. The data can be either allocated and maintained in application memory or be managed by the SAGA implementation.

Session The Session object provides the functionality to interactively exchange information between two computers and isolates independent sets of SAGA objects from each other. Sessions support the management of security information by using contexts.

Context The `Context` class is a container for security information and is attached to a `Session` object to make the information available to all objects instantiated in that session. Multiple contexts can co-exist in one session for different method calls and can be shared between sessions.

Permission The `permission` package contains an interface to let applications allow or deny specific operations on SAGA objects or grid entities for different types of users. Because it is difficult to anticipate how different types of middleware handle these permissions, applications using the `permission` package are not expected to be fully portable between SAGA applications. In addition, each implementation must specify which permissions it supports and for which operations.

Attributes The `attributes` package provides an interface for storing and retrieving attributes associated with SAGA objects. The supported attributes of an object are included in the description of the object in the language binding.

Monitoring The `monitoring` package provides a mechanism to monitor certain properties of monitorable SAGA objects by exposing metrics to the application. These metrics which represent monitorable entities, such as state or CPU time used. Steerable objects even allow certain metric values to be changed. An example of a monitorable object is a `Task.` The `Task` object has a `task.state` metric which can be monitored. If a special object, called a `Callback`, is attached to the task and the state changes, the SAGA implementation calls the `cb()` method of the `Callback` to respond to the state change of the task.

Task The last package of the Look and Feel is `task`. Tasks are representations of asynchronous operations and each SAGA object that implements the `async` interface is obliged to offer synchronous and asynchronous method calls.

3.2.2 API Packages

The interfaces, classes and methods defined in the API or functional packages of GFD.90 are, in general, representing explicit entities and actions of some backend system. The currently specified packages are shown below, but new packages may be added in the future.

Job The `job` package offers the functionality to submit jobs to grid resources and to monitor and control these jobs. Job submission can be done in batch mode or interactive mode. Jobs are controlled through different method calls such as `run()` and `suspend()`. Status information can be retrieved for both running and completed jobs.

Namespace The `namespace` package describes notions of hierarchical namespace entries and directories. These `NSEntry` and `NSDirectory` objects allow to navigate through a namespace such as a file system. Operations like moving, renaming, copying or removing these namespace entries are also supported.

File The `file` package is an extension of the `namespace` package and, in addition to all the operations from `namespace`, allows access to the contents of the files regardless of their location. It also offers the Scattered, Pattern-Based and Extended I/O paradigms.

Logicalfile The `logicalfile` package describes the interaction with replica systems, especially logical files, logical directories and creating replicas. A logical file is a namespace entry with some metadata and associated set of physical replica files. A replica (or physical file) is a file which is registered on a logical file. In general, all replicas registered on the same logical file are identical, but can be on different machines in the grid.

Stream The `stream` package specifies the functionality to create simple, remote sockets to establish connections between components. These components can then form a distributed application together.

RPC The Remote Procedure Call (RPC) or `rpc` package specifies operations to execute code on other machines. This is done by invoking methods on different machines. A high level API called GridRPC [20] is imported into SAGA and adapted to the SAGA look and feel. Semantically, GridRPC maps to the RPC package.

3.3 Reference Implementations

SAGA currently has two reference implementations, one written in Java and one written in C++. In this section we will explain what they are and how they work.

3.3.1 Java SAGA Reference Implementation

After GFD.90 was released in January 2008, a Java SAGA reference implementation was created at the Vrije Universiteit and released in September 2008 [27]. It is largely based on JavaGAT [29], a toolkit which provides a high-level, middleware-independent interface to grids. JavaGAT is the reference implementation for the GAT API [3], which shares many goals with SAGA. Common goals are the aim to make it easier for grid users to create complex grid applications and to shield them from the underlying middleware. The structure of JavaGAT can be seen in figure 1.

The Java SAGA reference implementation consists of different layers which all have its own responsibility. The top layer is the API that serves as the interface to the user. Below the API resides the engine. The engine is responsible for delegating the API calls to the correct middleware. The Capability Provider Interface is the layer which connects the engine to middleware specific software, called *adaptors*. When the engine receives a request, e.g., to copy a file, it selects the right adaptor with help from the CPI. The adaptor then delegates the copy request to the actual middleware that copies the file.

Principles used from JavaGAT in the SAGA Java reference implementation include intelligent dispatching, nested exceptions and the adaptor writing framework. Intelligent dispatching is the process where the engine chooses the correct adaptor for the requested action and the related preferences. First JavaGAT looks at the preferences specified by the user and all the adaptors.

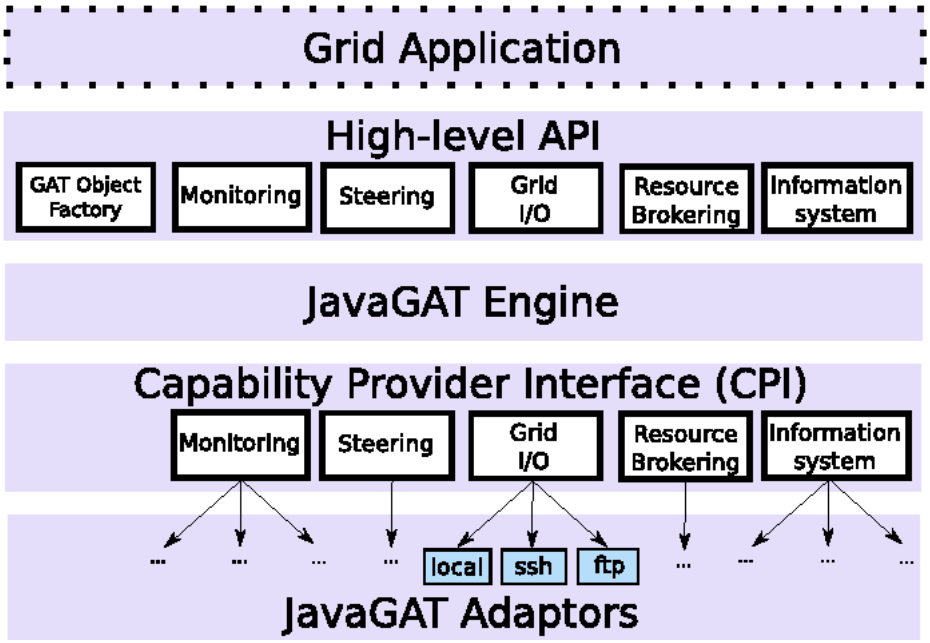

Figure 1: The structure of the JavaGAT implementation, taken from [29]

Adaptors that do not match or are unsuitable with these preferences are filtered out of the list to execute this specific method call. The adaptors that are not filtered out are instantiated by calling their constructors. Adaptors which threw an exception are then filtered out and the exception is added to the nested exception (see below). The list with instantiated adaptors is then sorted by using a *adaptor ordering policy* to specify which adaptor should be tried first to execute the method call. Each adaptor is then tried until one of them succeeds. If none succeeds, the nested exception is thrown.

Nested exceptions are special aggregations of exceptions which come from the different adaptors. Adaptors might raise exceptions when they fail at a certain action or do not implement the requested action. All these exceptions are stored in a nested exception which is only thrown to the grid application after every adaptor failed at fulfilling the request.

The adaptor writing framework makes it easy for users to write or change adaptors with little effort. This is an advantage because there are many middleware systems available and they are changing often.

The Java reference implementation implements SAGA completely since this is a requirement in the SAGA specification to be called fully compliant. Implementation which do not implement all specified methods are called partially compliant. There are still some deviations from GFD.90, such as symbolic links and permissions. Java does not have a notion of symbolic links and permissions so it is not possible to include them. If methods relating to them are called, the implementation will throw a `NotImplementedException`. The implemented Java language binding also contains small extensions like file streams and using RPC with objects instead of using byte arrays.

Adaptors currently included with the Java reference implementation are XMLRPC for RPC, Socket for streams, Gridsam for jobs, and adaptors from JavaGAT. There are 39 adaptors available divided in different file, logical file, resource broker, monitorable and endpoint adaptors. More information can be found at [2].

3.3.2 C++ SAGA Reference Implementation

Before the GFD.90 specification for SAGA was released, a group at the Louisiana State University [28] started to build a C++ reference implementation [17]. People from that group were

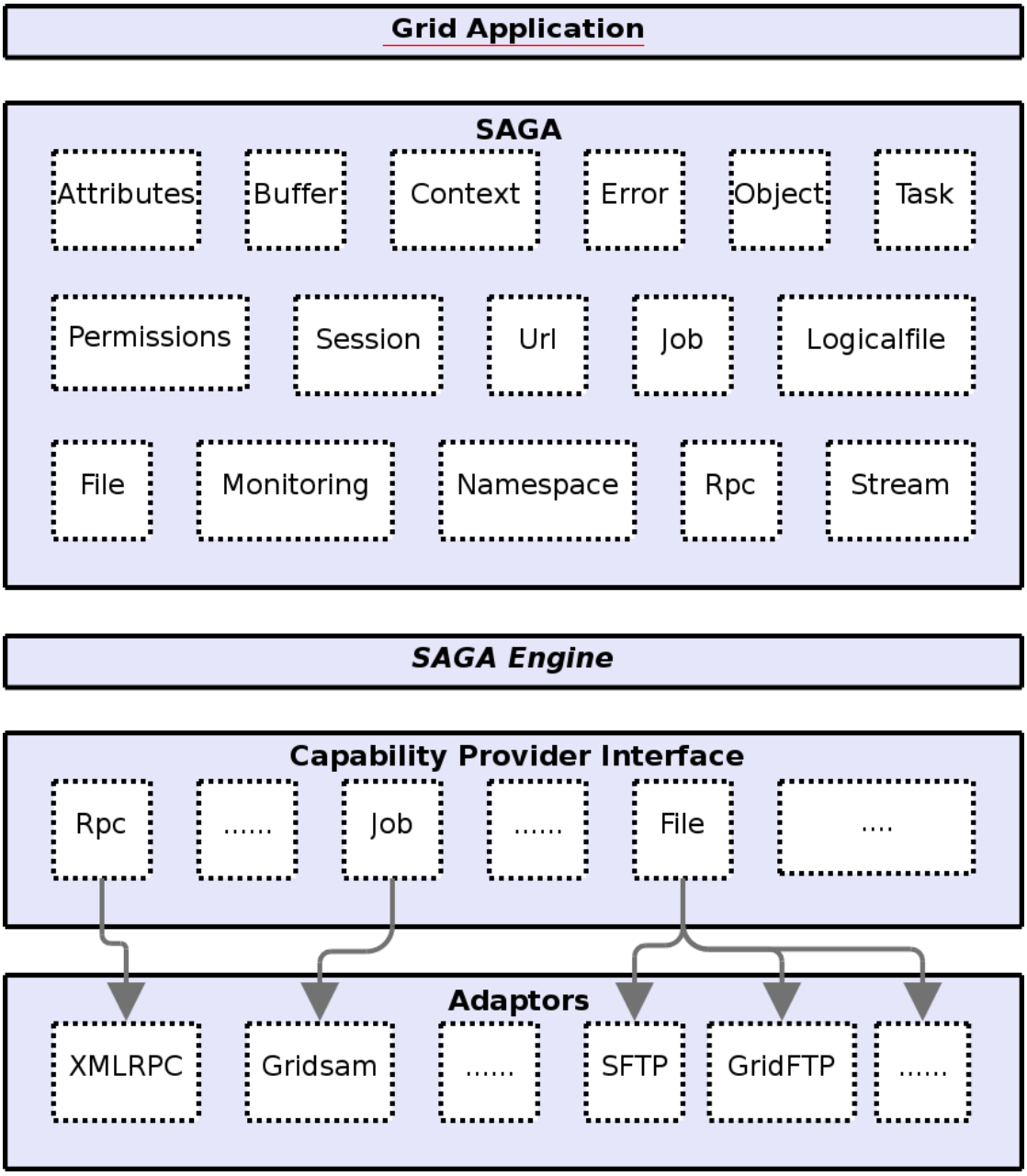

Figure 2: The structure of the Java SAGA reference implementation

also involved in the SAGA design process and the development of GAT, and pioneered in implementing a SAGA C++ implementation. This implementation works in a similar way as the Java implementation. It also includes call routing through to the adaptors, the probing of multiple adaptors and the variable adaptor strategies. In addition, it tries latency hiding strategies such as bulk optimization and automatic load distribution over multiple adaptors.

The C++ reference implementation offers wrappers for C and Python, but they are in alpha stage and therefore not fully functional. A Python language binding is not available and the wrapper closely follows the syntax of the underlying C++ layer. The Python language binding described in this thesis is specified top-down from the SAGA specification, where as the wrapper to C++ is designed bottom-up from the implementation. For a discussion, see Section 7.4.

4 Python

The Python language binding for SAGA is a specification of the classes and methods defined by SAGA in the programming language Python. In this section we will explain fundamentals of Python and how it influenced the specification.

4.1 Introduction to Python

Python was created in 1989 by Guido van Rossum who was working at the Centrum voor Wiskunde en Informatica (CWI) in the Netherlands. He worked on the programming language ABC, but needed some extra features. Not wanting to fall back to C, he created his own language called Python.

Python is a high-level, interpreted scripting language. It is object-oriented, but also borrows some features from functional languages. Python uses whitespace to define the blocks of code where other languages would use brackets. Python uses whitespace to improve readability. The Python interpreter is written in C, sometimes called CPython, and can be extended by writing new modules or by writing and compiling special C modules for it. Because the interpreter is written in C, it is portable to platforms which support the ANSI C compiler. Scripts have to be written only once to be run on many platforms. Some projects have rewritten the interpreter in different programming languages to take advantage of the distinct features of the language. These projects include Jython (an interpreter written in Java) and IronPython (an interpreter written in C#). According to estimations [26], Python is the sixth most popular programming language after Java, C, C++, VB and PHP. This rating is based on information from search engines, engineers and companies world-wide. Together with Java, C++ and C, support for Python will give users four programming languages to choose from to develop applications using SAGA.

4.2 Syntax Features

In this section we will explain several Python specific features we used in the SAGA Python language binding and how these syntax features influenced the language binding.

4.2.1 Dynamic Typing

Python uses dynamic typing, which means that the type of a variable is bound to the type of the variable value, and can change regularly. In static typing, the type is set when the variable is created and cannot be changed afterwards. Dynamic typing gives new possibilities when dealing with return values and parameter types. A commonly heard term is that Python uses 'Duck Typing' ("If it looks like a duck and quacks like a duck, it must be a duck."). This means that calling a method of an object is always possible. Whether it actually succeeds or results in a runtime error depends on the object. In this context it means that if you can call `quack()` and

`walk()` on an object, the type was probably `Duck`. If it gave a runtime error, the object was definitely not a `Duck`. There are however, ways to check the type of a variable, like `type()`, `isinstance()`, `issubclass()` or `variable.__class__` .

4.2.2 Keywords and built-ins

Python has reserved keywords just like every programming language. Most of these are similar to other languages, but Python has some specific ones which are also used in the SAGA specification. In SAGA's `Permission` enum `Exec` and `None` are mentioned, but both `exec` and `None` are reserved. Defining all constants lowercase would give problems with the reserved keyword `exec` while declaring all constants in CapWords gives problems with the keyword `None`. Although Python is case-sensitive, this gives problems when defining a consequent naming scheme.

Python also has some standard functions which are available. These include functions like `list()`, `tuple()`, `dir()` and `type()`. SAGA has some parameters names which interfere with the built in methods, such as `type` in the `Metric` constructor.

4.2.3 Methods

Methods are defined in Python by using the `def` keyword followed by the method name and the parameters in brackets. `def` defines a method, but can also be used to overwrite a previously defined method with the same name. This makes it impossible to do overloading. Overloading is defining multiple methods with the same name but only differing in parameter types. The same effect can still be achieved in Python since the parameter type is not defined in the method declaration. It does give difficulties where the SAGA specification has overloaded methods with completely different sets of parameter types. An example of an overloaded method is the `copy()` method from the `NSEntry` and `NSDirectory` classes in SAGA's `namespace` package. The method has a slightly different meaning in both classes.

Multiple Return Values

Python also supports returning multiple variables from one method call opposed to many other languages. This removes the need for using special objects that hold multiple variables, or special global variables. The actual returned type is a special read-only list called a tuple, but users can use multiple variables to automatically unpack the tuple among the variables. See Algorithm 1 for an example.

Algorithm 1 Example of multiple return values in Python

```
def method():
    variable1 = 'a String'
    variable2 = 2
    return variable1, variable2

var1, var2 = method()
```

Since the variables are dynamically typed, the same method can return multiple variables of different types. This feature is for example used in the `JobService.run_job()` method, which returns the job and three handles to the standard input, output and error.

Default Parameter Values

In Python it is possible to specify default values for parameters in the method definition. Default values for parameters are optional. If a method definition has parameters with default values, those parameters need to be placed behind the parameters without default values. Python needs

this order to determine which given value in the method call belongs to which parameter. See Algorithm 2 for an example. In the example, the `pass` statement does nothing, but leaving the method empty will result in a syntax error.

Algorithm 2 Example of default parameters in Python. Each `write()` call has exactly the same parameter values.

```
def write( buffer, size=3, offset=5):
    pass

write( buf )
write( buf, 3 )
write( buf, 3, 5 )
write( buf, offset=5 )
write( offset=5, size=3, buffer=buf)
```

Named Parameters

In Algorithm 2, the last two `write()` calls explicitly define the parameter names. This explicit naming of the parameters allows that, when dealing with a large amount of defaulted parameters, programmers can specify which parameters they want to change in the method call instead of specifying all the parameters. An example is shown in Algorithm 3.

Algorithm 3 Example of explicitly naming the parameters

```
def method( first=1, second=2, third=3, last = '4'):
    pass

method ( last = 'final')
```

To improve usability in the Python language binding, deciding about the order of the defaulted parameters is an interesting issue. A `File.read()` call from the SAGA specification defined as `read(size='-1', buffer=None)` has a different parameter order than a `File.write()` call which is defined as `write( buffer, size='-1')`. This looks inconsistent, but is done because a data buffer is needed for writing, but is optional for reading.

Data Types

Although Python supports many data types, it misses some types which are standard in other languages and are mentioned in GFD.90. These include enum, byte and char. Characters are just strings with a length of one. The array package does have the notion of a character to store strings of length one, but there is no separate data type.

Python also has no notion of an enum, but this can be solved in different ways. First of all, a class can be constructed containing all the values from the enum, such as the Flag enums in the SAGA specification. Another solution is to use dictionaries, which have similarities with hashtables in other languages. Dictionaries are commonly used in Python, but their syntax is less straightforward than using simple objects filled with variables. In addition to that, the dictionary has to be put in the class and gives an unnecessary long syntax, like `Flags.values['READ']` instead of `Flags.READ`. The values dictionary could be placed in the package, like `saga.file.values`, but would then conflict with the package or module naming scheme.

The byte data type is missing from Python. This causes some confusion how to specify and implement the mutable Buffer defined in SAGA. The Buffer encapsulates a sequence of bytes to use in read and write operations and is an essential part of SAGA. A possibility is to deny

users to use application managed buffers and use implementation managed buffers only, but this would limit the use of Buffers in SAGA and exclude a part of SAGA from the language binding. There is no standard to use buffers and they are not available in Python. Users could write their own buffer library, but it defeats the platform independent principle.

A solution was found in using arrays of chars. Arrays are guaranteed to contain only the specified data type, and if characters are read or written they are immediately visible to users. Problems may arise if underlying implementations use different encodings for the characters, like Unicode, but that should be solved the programmers implementing the language binding. Suggestions were made by people on the saga-rg mailing list to completely remove the buffers since they would be not *pythonesque* or *pythonic*[1]. We choose not to remove the buffers because of two reasons. The first reason is that the SAGA specification demands buffers in the SAGA Python language binding to be fully compliant. Removing them would result in a partial compliant language binding, which was not our goal. The second reason is that buffers like rpc's Parameter and file's IOVec are needed for specific functionality. Removing them would cause problems while solving no problems.

Like any other programming language, Python has support for numbers. Normally programmers will only need an integer to represent whole numbers. If the integer overflows because two very large numbers are multiplied, Python will automatically switch to a long. Longs can be arbitrarily long in Python. Internally, the integer in CPython is implemented as a C long and implemented as a BigInteger in Jython. Python has no double but only a double-precision float.

4.2.4 Extending Python

As mentioned in Section 4.1, it is possible to write libraries for Python and import them dynamically. These libraries can be written in Python but it is also possible to write them in C or C++. It is possible to program a library which is usable by the CPython interpreter by using predefined data types and structs. The libraries can be used to speedup performance in certain loops, or to add functionality to applications which is not available in standard Python. The downside of this mechanism is that the switching between the library and the interpreter takes some time, which creates a performance hit for simple calls, like adding two numbers, that could also be done in standard Python. The writing of these libraries is not trivial since it uses and converts many Python specific data types and specific syntax. Dealing with pointers needs to be precise or the library causes segmentation faults. Fortunately there are tools available which help in this task, such as SWIG [31] and Boost [18]. The Boost tool is used in the SAGA C++ reference implementation to create their Python wrapper, which makes the SAGA functionality available to Python users.

During the specification of the Python language binding we have used the features which are already available in the language and have avoided using features which would mean that extra libraries would have to be added. The specification can therefore be implemented by using the standard Python distribution or derivatives like Jython (see Section 4.3).

4.2.5 Versions of Python

Our Python language binding conforms to the 2.5 version of Python which was released in February 2008. This does not say whether the underlying implementations of the language binding and Python interpreters like Jython accept the full 2.5 syntax. The language binding not expected to change or become invalid in upcoming versions of Python.

Version 3.0 of Python was released in December 2008. This version is incompatible with the previous versions. There are some changes which effect our language binding. More information can be found in Section 8.

[1] Although *pythonic* is an extremely vague term, Python programmers generally assume there is only one correct way to do something. Other solutions are not *pythonic*.

4.3 Jython

The Python reference implementation is not the only interpreter of Python source code as there are multiple implementations available. Stackless [23] has all the functions of the reference implementation but is internally different to let multi-threaded Python applications perform much better. The implementation in .Net is called IronPython [13] and the implementation in Java is called Jython [16]. These implementations allow users to make use of Python and Java or Python and .Net, so they can take advantage of features from these languages and all tools and libraries which are written in them.

Writing a SAGA Python reference implementation would take a significant amount of time and resources, so it is better to use an existing SAGA implementation. To let a Python application use the SAGA C++ reference implementation, it can access it through the wrapper. To let a Python application access the Java implementation, we have to use something different than the Python reference implementation. The solution for running Python application on top of the SAGA Java reference implementation is to run Python applications on the Jython interpreter which can use the underlying SAGA Java reference implementation.

Jython was first called JPython when it was created in late 1997 by Jim Hugunin. In 2000 it was moved to SourceForge and renamed to Jython. Its current version is 2.2.1. The 2.5 release is scheduled around the beginning of 2009, and will probably have the same features as the 2.5 release of the Python reference implementation.

4.4 Differences between Jython and Python

Jython and Python interpret the same language, but there are some differences between them. An out-dated list of them can be found at [4], which shows that differences are mostly in behavior between the two interpreters. Jython also lacks some built-in modules and modules like Numeric for scientific calculations. It will always lag behind the reference implementation since new features and changes to the language are designed and implemented there first. This could become a problem if programmers write programs using the most recent syntax. In that case, it is possible that the application will not run on Jython.

Jython is also different from CPython because Jython is bound to the restrictions of the Java language. These restrictions include not being able to with operating system specific issues such as links and permissions.

When taking all the differences into account, it should be possible to create applications which run on both the CPython and the Jython interpreter as long the programmer remembers to keep the differences into account. Python applications developed using Jython should certainly run on a recent Python reference implementation.

5 Specification of the SAGA Python Language Binding

This section describes how the specification of the SAGA Python language binding was designed and which choices were made in the process. Although the complete specification, including all the notes of the 16 modules, 55 classes and more than 200 methods, would be too much to be included within this thesis, we will describe most design decisions. The full specification can be found in a subversion repository accessible through the PySaga project website [1].

5.1 Global Design Decisions

The specification for the language binding was inspired by three things; The GFD.90 [9] document, the Python style guide [8] and suggestions given on the saga-rg mailinglist. Since we did not have much experience with Python the people on the mailinglist gave us quite some insight about proper syntax and programming constructs. The style guide was a good start to set the

Algorithm 4 Layout of the PySaga source code

```
- saga
    - __init__.py
    - attributes.py
    - buffer.py
    - context.py
    - error.py
    - file.py
    - job.py
    - logicalfile.py
    - monitoring.py
    - namespace.py
    - object.py
    - permissions.py
    - rpc.py
    - session.py
    - stream.py
    - task.py
    - url.py
```

rules on how the specification should look like. GFD.90 contained all specifics on SAGA itself and is the primary source of information related to SAGA. Most naming was directly taken from the document, just as all the documentation.

The layout of the packages is simple is can be seen in Algorithm 4. The root of the package structure is `saga`. This is the name of a directory containing all the modules. The modules are files with the `.py` extension and contain the classes which are related to each other and are defined as a package in the GFD.90 document. Python recognizes this structure of a directory, module files and a special `__init__.py` file as a complete package is able to import them. For example the URL class can be imported with the statement `from saga.url import URL`.

All the document strings in the specification are complemented with special statements used by an application called Epydoc [7]. Epydoc can generate a browsable version of the language binding which helps application programmers designing Python programs using SAGA. Almost all documentation in the document strings comes from GFD.90 and is adapted to the language binding.

There are different naming conventions available. The CapitalizedWords convention is the convention to start each word in a name with a capital letter and to use no space, dash or underscore between words. The name of the convention is an example of this. The all-lowercase convention consists of creating names without capitals and without characters between words. The all-lowercase with underscores convention uses names in lowercase and uses underscores to separate words in a name, like in `saga_object`. Names in the UPPERCASE convention consist of words written in capital letters and using no characters between the words. We have used the CapitalizedWords convention for the naming of classes, the all-lowercase convention for package and module names, the all-lowercase with underscores for the method names, the parameter names and variable names, and the UPPERCASE convention for constants. These conventions come from the style guide with the exception of convention for constants. Using all-lowercase or CapWords would create clashes between some constants and reserved keywords as mentioned in Section 4.2.2.

Interfaces do not exist in Python, but GFD.90 does define them. Interfaces are specified as normal classes, with the specified methods in them. Implementations of the language binding should make sure that instantiating an 'interface' class does not work since it makes no sense to do so.

None	`__init__(self, message, saga_object=None)` Initialize an Exception object.
string	`get_message()` Returns the message associated with the exception.
Object	`get_object()` Returns the SAGA Object associated with the exception.

Table 1: Specified methods in Error module

Since enumerations do not exist in Python but GFD.90 does define them, we have specified the enums as normal classes in the specification. These classes contain the constants from the enums.

We have mapped the possible return types to their Python counterpart, such as strings to strings, numbers to ints and arrays to lists.

To deal with asynchronous methods and the task creation we have added a tasktype parameter to all methods in classes which are subclasses of the `Async` class. This parameter is the last one of all the defined parameters and defaulted to `TaskType.NORMAL`. This means that by default the method is executed synchronous and returns its specified return value and not a `Task` object. By placing the parameter last and by defaulting it, it the syntax of the synchronous and asynchronous calls stay consistent and should not interfere with each other.

Get and set methods are not regarded as *pythonic,* but GFD.90 requires them for the language binding to be fully compliant. Therefore many class variables on which the getters and setters operate can also be accessed through so-called properties. In the description of the packages we shall mention the properties where they are defined. Another reason why getters and setters are included is that some classes are subclasses of the `async` interface. This means that the methods defined in the class can be executed synchronously and asynchronously. By only having the properties, users cannot easily switch between asynchronous and synchronous since the specific method is set when defining the properties. Therefore, the get and set methods are still needed. In the Python language binding specification, only synchronous versions of the methods are used to define the properties.

The complete class and interface hierarchy is shown in Figure 3. The purple blocks are the packages defined in PySaga. In the package blocks, the classes are shown. Although Python has no notion of interfaces, the interfaces in the figure are the classes that GFD.90 defines as interfaces. These classes are not supposed to be instantiated and are therefore different from the other classes. The arrows mean that a certain class extends or inherits another class. An example is that `IOVec` inherits from `Buffer`. The small numbers in the classes mean that the specific class implements the interfaces labeled by those numbers. An example is that `Session` implements `Object`, which is labeled by big 1. `Async`, labeled by a big 4, implements no interface but `Async` itself is implemented by many classes, such as `Permissions`. Although `SagaException` is the only class shown in the `error` package block, all subclasses of `SagaException` are also in the `error` package, but are not shown in the figure.

5.2 Error

The `Error` module consists of a `SagaException` class which features the methods from Table 1 and 11 subclasses of the `SagaException` which are shown in Table 2. We have placed the `sagaObject` parameter behind the message parameter because not all exceptions contain an object and thus the `saga_object` is optional. `__init__` contains both constructors mentioned in GFD.90. `SagaException` exposes the properties `message` and `saga_object` to directly get the message and the associated base `Object`. Python has exception-handling capabilities and does not need the `error_handler` interface.

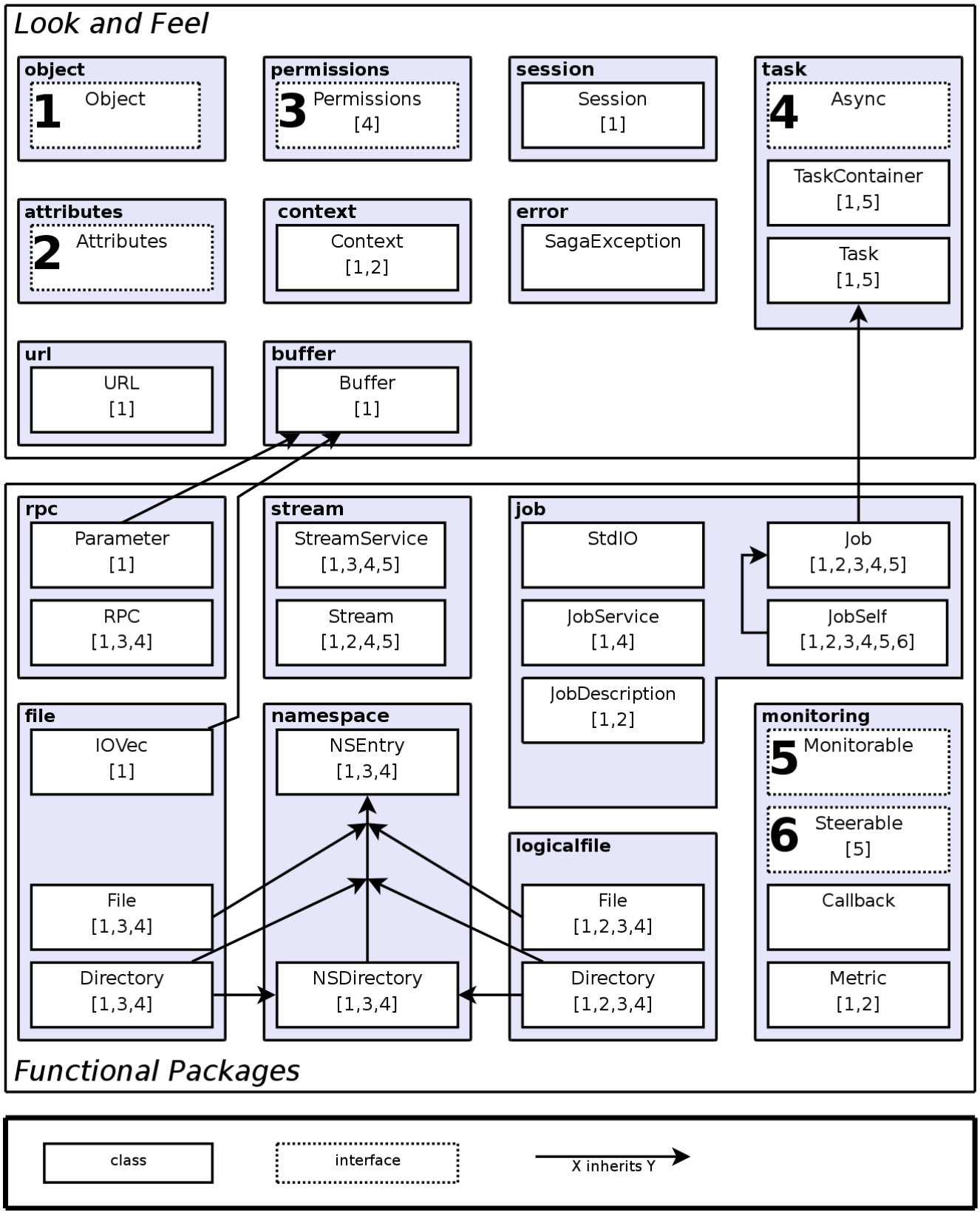

Figure 3: The PySaga hierarchy

NotImplemented	IncorrectURL	BadParameter	AlreadyExists
DoesNotExist	IncorrectState	PermissionDenied	AuthorizationFailed
AuthenticationFailed	Timeout	NoSuccess	

Table 2: Subclasses of SagaException

string	get_id(self) Query the object ID.
int	get_type(self) Query the object type.
Session	get_session(self) Query the objects session.
Object	clone(self) Deep copy the object.

Table 3: Methods from the Object class

5.3 Object

The Object module contains the `ObjectType` class and the `Object` class. `ObjectType` was originally designed as an enum but is now a class with constants that are returned by objects to describe their type. `Object` was an interface, but is now a class with the same methods. The methods are shown in Table 3, and are not different from the GFD.90 specification. `get_type()` returns a value from `ObjectType` which happen to be integers and `clone()` returns a deep copy of an object which is an object itself.

`Object` exposes the properties `id` to get the object ID, `type` to get the object type and `session` to get the `Session` of the object.

5.4 URL

The `url` package contains only the `URL` class. The methods defined in `URL`, shown in Table 4, are practically the same as defined in GFD.90. `URL` is a subclass of `Object` and thus inherits all its methods and properties. Additional properties exposed by `URL` are `string`, `scheme`, `host`, `port`, `fragment`, `path`, `query` and `userinfo`.

5.5 Buffer

The buffer package contains only the Buffer class. Buffer contains the methods shown in Table 5. Both constructors are merged into the `__init__` method. This caused a switch of the parameters. With a `Buffer(size, data)` call the buffer should become application managed and with a `Buffer(size)` call the buffer should become implementation managed. `get_size()` and `set_size()` are like in GFD.90, but `get_data()` and `set_data()` are different. `get_data()` returns a char array or list of chars to represent the data if the buffer is application managed. It returns a string containing the data if the buffer is implementation managed. Using `set_data()` switches the buffer to application managed, and accepts a char array or list of chars as the data parameter. Python does not have a byte data type in version 2.0 and that is why chars are used instead of the bytes specified in GFD.90. The properties specified in Buffer are data and size.

5.6 Session

The session package contains the `Session` class. `Session` is specified completely like GFD.90 and contains the methods shown in Table 6. `list_contexts()` returns a list of contexts instead of an array. The contexts are also available through the property `contexts`.

	`__init__(self, url='')` Initialize an URL instance.
	`set_string(self, url='')` Set a new url.
`string`	`get_string(self)` Retrieve the url as string.
	`set_scheme(self, scheme='')` Set the scheme of the url.
`string`	`get_scheme(self)` Get the scheme of the url.
	`set_host(self, host='')` Set the host of the url.
`string`	`get_host(self)` Get the host of the url.
	`set_port(self, port=-1)` Set the port of the url.
`int`	`get_port(self)` Get the port of the url.
	`set_fragment(self, fragment='')` Set the fragment of the url.
`string`	`get_fragment(self)` Get the fragment of the url.
	set_path(self, path='') Set the path of the url.
`string`	`get_path(self)` Get the path of the url.
	`set_query(self, query='')` Set the query of the url.
`string`	`get_query(self)` Get the query of the url.
	`set_userinfo(self, userinfo='')` Set the userinfo of the url.
`string`	`get_userinfo(self)` Get the userinfo of the url.
`URL`	`translate(self, scheme)` Translate an URL to a new scheme.

Table 4: Methods from the URL class

	`__init__(self, size=-1, data=None)` Initialize an I/O buffer.
	`__del__(self)` Destroy a buffer.
	`set_size(self, size=-1)` Set size of buffer.
`int`	`get_size(self)` Retrieve the current value for size.
	`set_data(self, data, size=-1)` Set new buffer data.
`char array, list or string`	`get_data(self)` Retrieve the buffer data.
	`close(self, timeout=-0.0)` Closes the object.

Table 5: Methods from the Buffer class

	`__init__(self, default=True)` Initialize the object.
	`add_context(self, context)` Attach a security context to a session.
	`remove_context(self, context)` Detach a security context from a session.
`list`	`list_contexts(self)` Retrieve all contexts attached to a session.

Table 6: Methods from the Session class

5.7 Context

The `context` package contains only the `Context` class. `Context` inherits all methods and properties from `Object` and `Attribute` and has two additional methods as shown in Table 7. The constructor uses `name` as parameter name since `type` is already used in Python. The known attributes of `Context` are also defined as properties. They are `Type`, `Server`, `CertRepository`, `UserProxy`, `UserCert`, `UserKey`, `UserID`, `UserPass`, `UserVO`, `LifeTime`, `RemoteID`, `RemoteHost` and `RemotePort`.

5.8 Permissions

The Permissions module contains the Permission and the Permissions classes. Permission is a class with constants from the permission enum defined in GFD.90. The values are in capitals since the CapitalizedWord and the lowercase conventions cannot apply to none and exec since None and exec are both keywords.

The Permissions class is defined as an interface in GFD.90, but is now a class. It contains the methods shown in Table 8. The methods are much like the methods defined in GFD.90, except for the tasktype parameters. Permissions implements the async interface and needs to

`__init__(self, name='')` Initialize a security context.
`set_defaults(self)` Set default values for specified context type.

Table 7: Methods from the Context class

	`permissions_allow(self, id, perm, tasktype=TaskType.NORMAL)` Enable permission flags.
	`permissions_deny(self, id, perm, tasktype=TaskType.NORMAL)` Disable permission flags.
`bool`	`permissions_check(self, id, perm, tasktype=TaskType.NORMAL)` Check permission flags.
`string`	`get_owner(self, tasktype=TaskType.NORMAL)` Get the owner of the entity.
`string`	`get_group(self, tasktype=TaskType.NORMAL)` Get the group owning the entity.

Table 8: Methods from the Permissions class

implement synchronous and asynchronous versions of all methods. The `tasktype` parameter allows to distinguish between those versions. Permissions defines the properties `group` and to access the directly get the owner and group of the object implementing Permissions.

5.9 Attributes

`Attributes` is the only class in the `attributes` package. It contains the the methods shown in Table 9. `Attributes` was designed as an interface in GFD.90 and is class in the language binding. The arrays of strings have changed into lists, but all other parameters and data types stay the same. `Attributes` had a property called `attributes`, which is a dictionary of all the attributes of the entity that is a subclass of `Attributes`.

	set_attribute(self, key, value) Set an attribute to a value.
string	get_attribute(self, key) Get an attribute value.
	set_vector_attribute(self, key, values) Set an attribute to an array of values.
list	get_vector_attribute(self, key) Get the tuple of values associated with an attribute.
	remove_attribute(self, key) Removes an attribute.
list	list_attributes(self) Get the list of attribute keys.
list	find_attributes(self, pattern) Find matching attributes.
bool	attribute_exists(self, key) Check the attribute's existence.
bool	attribute_is_readonly(self, key) Check if the attribute is read only.
bool	attribute_is_writable(self, key) Check if the attribute is writable.
bool	attribute_is_removable(self, key) Check if the attribute is removable.
bool	attribute_is_vector(self, key) Check whether the attribute is a vector or a scalar.

Table 9: Methods from the Attributes class

5.10 Monitoring

The `monitoring` package contains four classes: `Callback`, `Metric`, `Monitorable` and `Steerable`. `Callback` is a class that users must subclass to have any meaning for their application. They also have to implement the `cb()` method to specify the correct behavior. A `Callback` object, together with a name of a metric, can be added to various objects through the `add_callback()` method. If the metric changes, the SAGA implementation will call the `cb()` method of `Callback` object. The `cb()` method, see Table 10, uses different names than GFD.90, but has the same order and meaning. The names are different to make them clearer and consistent. For example, the `context` parameter is also used in `Session.add_context()`.

`bool`	`cb(self, monitorable, metric, context)` Asynchronous handles for metric changes

Table 10: Method from the Callback class

The `Metric` class specifies the methods, shown in Table 11, almost exactly like GFD.90. The differences are the renaming of the `type` parameter from the constructor to `mtype` and the addition of properties. These properties are `Name`, `Description`, `Mode`, `Unit`, `Type` and `Value.`

The `Monitorable` class is contains the methods shown in Table 12. The string array has been replaced by a list of strings and a `metrics` property has been added. Other than that, the methods are the same as in GFD.90.

The `Steerable` class contains the methods defined in Table 13. The methods do not differ from GFD.90.

5.11 Task

The `task` package consists of six classes: `State`, `WaitMode`, `Async`, `TaskType`, `Task`, `TaskContainer`. `State` and `WaitMode` are defined as enums in GFD.90 and are now classes with constants. `Async` is an empty class since it is only used for tagging classes that implement the task model. `TaskType` defines the possible values used in combination with asynchronous methods. `NORMAL` means that the method returns its specified return values like a normal synchronous method, `SYNC` means that the method returns a `Task` object in a final state, `ASYNC` means that the method returns a `Task` object in the `RUNNING` state and `TASK` means that the method returns a `Task` object in the `NEW` state.

The `Task` class contains the methods shown in Table 14. `Task` does not specify a constructor because `Tasks` can only be created through asynchronous method calls. `get_result()` returns the result of the asynchronous call and can return any data type. `get_object()` is used to get the originating object of the `Task` and can also return any SAGA object that subclasses the `Async` class. `get_state()` returns a constant from the `State` class. Those constants are defined as ints. Task defines three properties; `state`, `result` and `object`.

	`__init__(self, name, desc, mode, unit, mtype, value)` Initializes the Metric object.
`int`	`add_callback(self, cb)` Add asynchronous notifier callback to watch metric changes.
	`remove_callback(self, cookie)` Remove a callback from a metric.
	`fire(self)` Push a new metric value to the backend.

Table 11: Methods from the Metric class

`list`	`list_metrics(self)` List all metrics associated with the object.
`Metric`	`get_metric(self, name)` Returns a metric instance, identified by name.
`int`	`add_callback(self, name, cb)` Add a callback to the specified metric.
	`remove_callback(self, name, cookie)` Remove a callback from the specified metric.

Table 12: Methods from the Monitorable class

`bool`	`add_metric(self, metric)` Add a metric instance to the application instance.
	`remove_metric(self, name)` Remove a metric instance.
	`fire_metric(self, name)` Push a new metric value to the backend.

Table 13: Methods from the Steerable class

	`__del__(self)` Destroy the object.
	`run(self)` Start the asynchronous operation.
	`cancel(self, timeout=0.0)` Cancel the asynchronous operation.
`bool`	`wait(self, timeout=-1.0)` Wait for the Task to finish.
`int`	`get_state(self)` Get the state of the Task.
`<return value>`	`get_result(self)` Get the result of the async operation.
`<object>`	`get_object(self)` Get the object from which this Task was created.
	`rethrow(self)` Re-raise any exception a failed Task caught.

Table 14: Methods from the Task class

	`__init__(self)` Initialize the TaskContainer.
	`__del__(self)` Destroy the TaskContainer.
`int`	`add(self, task)` Add a Task to a TaskContainer.
`Task`	`remove(self, cookie)` Remove a Task from a TaskContainer.
	`run(self)` Start all asynchronous operations in the TaskContainer.
`Task`	`wait(self, mode=0, timeout=-1.0)` Wait for one or more of the Tasks to finish.
	`cancel(self, timeout=0.0)` Cancel all the asynchronous operations in the container.
`int`	`size(self)` Return the number of Tasks in the TaskContainer.
`list`	`list_tasks(self)` List the cookies of the Tasks in the TaskContainer.
`Task`	`get_task(self, cookie)` Get a single Task from the TaskContainer.
`list`	`get_tasks(self)` Get the Tasks in the TaskContainer.
`list`	`get_states(self)` Get the states of all Tasks in the TaskContainer.

Table 15: Methods from the TaskContainer class

TaskContainer has the methods shown in Table 15 and has the properties `cookies` and `tasks`. The parameter `mode` in `wait()` must be a value from `WaitMode`. Overall, `TaskContainer` looks look the `TaskContainer` defined in GFD.90.

5.12 Job

The `job` package consists of six classes; `State`, `JobDescription`, `JobService`, `StdIO`, `Job` and `JobSelf`. `State` is defined like the `State` class in the `task` package, but it adds the `SUSPENDED` state for job.

`JobDescription` has no methods of its own, but has a lot of attributes exposed as properties. These properties are `Executable`, `Arguments`, `SPMDVariation`, `TotalCPUCount`, `NumberOfProcesses`, `ProcessesPerHost`, `ThreadsPerProcess`, `Environment`, `WorkingDirectory`, `Interactive`, `Input`, `Output`, `Error`, `FileTransfer`, `Cleanup`, `JobStartTime`, `TotalCPUTime`, `TotalPhysicalMemory`, `CPUArchitecture`, `OperatingSystemType`, `CandidateHosts`, `Queue` and `JobContact`.

`JobService` implements `Async`, therefore all methods have the `tasktype` parameter. The `url` and `session` parameters in the `__init__()` methods are switched because this allows `session` to be defaulted to the default session when only the `url` parameter is given when creating the `JobService`. `run_job()` now returns `StdIO` objects to represent the standard input, standard output and standard error. `list()` returns a list of strings instead of an array of strings. The list of methods can be seen in Table 16.

The `StdIO` class is a class that represents the standard input, standard output and standard error of a `Job`. It is modeled after the stdio objects in Python, which are actually special files. The methods of `StdIO` are shown in Table 17. `StdIO` has the properties name and mode.

`Job` objects can only be created through the `Jobservice` and have the methods shown in

	`__init__(self, url='', session=Session(), tasktype=Tasktype.NORMAL)` Initialize the object
`Job`	`create_job(self, jd, tasktype=Tasktype.NORMAL)` Create a job instance
`Job, StdIO, StdIO and StdIO`	`run_job(self, commandline, host='',Tasktype.NORMAL)` Run a command synchronously.
`list`	`list(self,Tasktype.NORMAL)` Get a list of jobs which are currently known by the resource manager.
`Job`	`get_job(self, job_id,Tasktype.NORMAL)` Given a job identifier, this method returns a Job object representing this job.
`JobSelf`	`get_self(self, Tasktype.NORMAL)` This method returns a Job object representing this job, i.e. the calling application.

Table 16: Methods from JobService

	`__init__(self)` Initializes the StdIO object.
	`close(self)` Closes the stream from reading or writing (if applicable)
	`flush(self)` Forces the data stream to flush the data
`string`	`get_name(self)` Returns the name of the StdIO object.
`string`	`get_mode(self)` Returns the mode of the StdIO object
	`write(self, data)` Write data to the stdin of the job.
	`writelines(self, data)` Write a sequence of strings to the stdin of the job.
`string`	`read(self, size=-1, blocking=True)` Read at most size bytes from stdout/stderr.
`string`	`readline(self, size=-1, blocking=True)` Read the next line from the stdout or stderr of the job.
`list`	`readlines(self, size=-1, blocking=True)` Read multiple lines from the stdout or stderr.

Table 17: Methods from StdIO

JobDescription	get_job_description(self) Retrieve the job_description which was used to submit this job instance.
StdIO	get_stdin(self) Retrieve input stream for a job.
StdIO	get_stdout(self) Retrieve output stream of job
StdIO	get_stderr(self) Retrieve error stream of job
	suspend(self) Ask the resource manager to perform a suspend operation on the running job.
	resume(self) Ask the resource manager to perform a resume operation on a suspended job.
	checkpoint(self) Ask the resource manager to initiate a checkpoint operation on a running job.
	migrate(self, jd) Ask the resource manager to migrate a job.
	signal(self, signum) Ask the resource manager to deliver an arbitrary signal to a dispatched job.

Table 18: Methods from Job class

Table 18. These methods do not differ from GFD.90 except for the discussed StdIO objects. Job has attributes that are also exposed as properties. These properties are JobID, ExecutionHosts, Created, Started, WorkingDirectory, ExitCode and Termsig.

The JobSelf class has the same methods as defined in Job.

5.13 Namespace

The namespace package consists out of the Flags, NSEntry and NSDirectory classes. Flags is now defined as a class of integer constants. NSEntry, with the methods shown in Table 19, is a class with some changes. The constructor has session and name parameters switched, each method has a tasktype parameter because NSEntry and NSDirectory subclass Async and a number of methods have a *_self* suffix added to the method name. The reason for this is that the same methods exist in NSDirectory, but with slightly different meaning. Since overloading is not available and keeping the same name would result in complicated and user unfriendly implementations, the method names were changed.

The differences in NSDirectory, shown in Table 20, are the tasktype parameter, switched name and session parameters and merging of the normal management methods with their wildcard versions. The merging of the methods copy, link, move and remove means that the same method accepts both URL objects as strings for the source parameter. This merging also happened for the permission_allow and permission_deny methods.

5.14 File

The file package contains the classes Flags, SeekMode, IOVec, File and Directory. Flags and SeekMode are classes with integer constants. We decided to keep the SeekMode constants like they are in GFD.90 (1, 2 and 3) and not change them to the variables Python uses to indicate the seek mode (0, 1 and 2). This should not create confusion between different SAGA implementations and encourages users to use the defined constants, like SeekMode.START.

IOVec is a subclass of Buffer and uses similar constructor using a char array or a list of chars. IOVec contains the methods shown in Table 21.

	`__init__(self, name, session=Session(), flags=0, tasktype=TaskType.NORMAL)` initialize the the object
	`__del__(self)` destroy the object
`URL`	`get_url(self, tasktype=TaskType.NORMAL)` obtain the complete url pointing to the entry
`URL`	`get_cwd(self, tasktype=TaskType.NORMAL)` obtain the current working directory for the entry
`URL`	`get_name(self, tasktype=TaskType.NORMAL)` obtain the name part of the url path element
`bool`	`is_dir_self(self, tasktype=TaskType.NORMAL)` tests the entry for being a directory
`bool`	`is_entry_self(self, tasktype=TaskType.NORMAL)` tests the entry for being an NSEntry
`bool`	`is_link_self(self, tasktype=TaskType.NORMAL)` tests the entry for being a link
`URL`	`read_link_self(self, tasktype=TaskType.NORMAL)` get the name of the link target
	`copy_self(self, target, flags=0, tasktype=TaskType.NORMAL)` copy the entry to another part of the name space
	`link_self(self, target, flags=0, tasktype=TaskType.NORMAL)` create a symbolic link
	`move_self(self, target, flags=0, tasktype=TaskType.NORMAL)` rename or move target
	`remove_self(self, flags=0, tasktype=TaskType.NORMAL)` removes this entry, and closes it
	`close(self, timeout=0.0, tasktype=TaskType.NORMAL)` closes the NSEntry
	`permissions_allow_self(self, id, perm, flags=0, tasktype=TaskType.NORMAL)` enable a permission
	`permissions_deny_self(self, id, perm, flags=0, tasktype=TaskType.NORMAL)` disable a permission flag

Table 19: Methods from NSEntry

	`__init__(self, name, session=Session(), flags=0, tasktype=TaskType.NORMAL)` initialize the object
	`change_dir(self, url, tasktype=TaskType.NORMAL)` change the working directory
	`list(self, name_pattern='.', flags=0, tasktype=TaskType.NORMAL)` list entries in this directory
	`find(self, name_pattern, flags=2, tasktype=TaskType.NORMAL)` find entries in the current directory and below
`bool`	`exists(self, name, tasktype=TaskType.NORMAL)` checks if entry exists
`bool`	`is_dir(self, name, tasktype=TaskType.NORMAL)` tests name for being a directory
`bool`	`is_entry(self, name, tasktype=TaskType.NORMAL)` tests name for being an NSEntry
`bool`	`is_link(self, name, tasktype=TaskType.NORMAL)` tests name for being a symbolic link
`URL`	`read_link(self, name, tasktype=TaskType.NORMAL)` returns the name of the link target
`int`	`get_num_entries(self, tasktype=TaskType.NORMAL)` gives the number of entries in the directory
`URL`	`get_entry(self, entry, tasktype=TaskType.NORMAL)` gives the name of an entry in the directory
	`copy(self, source, target, flags=0, tasktype=TaskType.NORMAL)` copy the entry to another part of the name space
	`link(self, source, target, flags=0, tasktype=TaskType.NORMAL)` create a symbolic link from the target entry.
	`move(self, source, target, flags=0, tasktype=TaskType.NORMAL)` rename or move target
	`remove(self, target, flags=0, tasktype=TaskType.NORMAL)` removes the entry
	`make_dir(self, target, flags=0, tasktype=TaskType.NORMAL)` creates a new directory
`NSEntry`	`open(self, name, flags=0, tasktype=TaskType.NORMAL)` creates a new NSEntry instance
`NSDirectory`	`open_dir(self, name, flags=0, tasktype=TaskType.NORMAL)` creates a new NSDirectory instance
	`permissions_allow(self, target, id, perm, flags=0, tasktype=TaskType.NORMAL)` enable a permission
	`permissions_deny(self, target, id, perm, flags=0, tasktype=TaskType.NORMAL)` disable a permission flag

Table 20: Methods from NSDirectory

	`__init__(self, size=-1, data=None, len_in=-1, offset=0)` initialize an IOVec instance
	`set_offset(self, offset)` set the offset
`int`	`get_offset(self)` retrieve the current value for offset
	`set_len_in(self, len_in)` Set len_in
`int`	`get_len_in(self)` retrieve the current value for len_in
`int`	`get_len_out(self)` Retrieve the value for len_out

Table 21: Methods from IOVec

`File`, methods shown in Table 22, is a subclass of `NSEntry` and also has the `session` and `name` parameters switched in the constructor. The `read()` method allows the omission of the `data` parameter to return a string and the `write()` method allows a string to be used as `data`. This lies close to the way `read()` and `write()` for files is implemented in Python. Any array defined in GFD.90 for the File class is replaced by a list. The `tasktype` parameter is added because `File` and `Directory` are both subclasses of `Async`.

`Directory`, methods shown in Table 23, also has the switched parameters in the `__init__()` method and has added the `tasktype` parameters.

5.15 Logicalfile

The `logicalfile` package contains the `Flags`, `LogicalFile` and `LogicalDirectory` classes. `LogicalFile` has a switched session parameter in the __init__() method and had the array replaced by a list. The methods for `LogicalFile` are shown in Table 24. `LogicalDirectory` has the same changes as `LogicalFile`. The methods of `LogicalDirectory` are shown in Table 25.

5.16 Stream

The `stream` package contains the `State`, `Activity`, `StreamService` and `Stream` classes. `State` and `Activity` are two classes with integer constants. `StreamService` is a subclass of `Async` and has the tasktype parameter in all methods, shown in Table 26. Stream also has a switched session parameter and tasktype parameters. The methods of `Stream` are shown in Table 27. Stream exposes the properties `Bufsize`, `Timeout`, `Blocking`, `Compression`, `Nodelay` and `Reliable`.

5.17 RPC

The rpc package contains the IOMode, Parameter and RPC classes. IOMode is a class with integer constants. Parameter, see Table 28, is a subclass of Buffer but also allows normal data types as data parameter. These normal data types can be integers, floats, booleans or strings or lists of them. This lies close to using these data types for the XMLRPC middleware that also accepts normal data types in addition to plain byte arrays. Parameter exposes the property `mode`.

RPC also has the switched session parameter, tasktype parameters and arrays replaced by lists. See Table 29 for the methods.

	`__init__(self, name, session=Session(), flags=512, tasktype=TaskType.NORMAL)` initialize the File object.
	`get_size(self, tasktype=TaskType.NORMAL)` returns the number of bytes in the file.
`int or string`	`read(self, size=-1, buf=None, tasktype=TaskType.NORMAL)` reads up to size bytes from the file into a buffer.
`int`	`write(self, buf, size=-1, tasktype=TaskType.NORMAL)` writes up to size from buffer into the file at the current file position.
`int`	`seek(self, offset, whence=0, tasktype=TaskType.NORMAL)` reposition the file pointer.
	`read_v(self, iovecs, tasktype=TaskType.NORMAL)` gather/scatter read.
	`write_v(self, iovecs, tasktype=TaskType.NORMAL)` gather/scatter write.
`int`	`size_p(self, pattern, tasktype=TaskType.NORMAL)` determine the storage size required for a pattern I/O operation.
`int`	`read_p(self, pattern, buf, tasktype=TaskType.NORMAL)` pattern-based read.
`int`	`write_p(self, pattern, buf, tasktype=TaskType.NORMAL)` pattern-based write.
`list`	`modes_e(self), tasktype=TaskType.NORMAL` list the extended modes available in this implementation, and/or on server side.
`int`	`size_e(self, emode, spec, tasktype=TaskType.NORMAL)` determine the storage size required for an extended I/O operation.
`int`	`read_e(self, emode, spec, buf, tasktype=TaskType.NORMAL)` extended read.
`int`	`write_e(self, emode, spec, buf, tasktype=TaskType.NORMAL)` extended write.

Table 22: Methods from File

	`__init__(self, name, session='default', flags=512)` initialize the Directory object.
`int`	`get_size(self, name, flags=None)` Returns the size of the file.
	`is_file(self, name)` alias for `NSDirectory.is_entry()`
`Directory`	`open_dir(self, name, flags=512)` Creates a directory object.
`File`	`open(self, name, flags=512)` Creates a new file instance.

Table 23: Methods from Directory

	`__init__(self, name, session=Session(), flags=512, tasktype=TaskType.NORMAL)` Initialize the LogicalFile.
	`add_location(self, name, tasktype=TaskType.NORMAL)` Add a replica location to the replica set
	`remove_location(self, name, tasktype=TaskType.NORMAL)` Remove a replica location from the replica set
	`update_location(self, old, new, tasktype=TaskType.NORMAL)` Change a replica location in replica set.
`list`	`list_locations(self, tasktype=TaskType.NORMAL)` List the locations in the location set
	`replicate(self, name, flags=0, tasktype=TaskType.NORMAL)` Replicate a file from a location to a new location.

Table 24: Methods from LogicalFile

	`__init__(self, name, session=Session(), flags=512,`
	`tasktype=TaskType.NORMAL)`
	Initialize the object.

	`is_file(self, name, flags, tasktype=TaskType.NORMAL)` alias for `NSDirectory.is_entry()`
`LogicalDirectory`	`open_dir(self, name, flags=512, tasktype=TaskType.NORMAL)` Creates a new `LogicalDirectory` instance
`LogicalFile`	`open(self, name, flags=512, tasktype=TaskType.NORMAL)` Create a new `LogicalFile` instance.

`list`	`find(self, name_pattern, attr_pattern, flags=2,`
	`tasktype=TaskType.NORMAL)`
	Find entries in directory and below.

Table 25: Methods from LogicalDirectory

	`__init__(self, url='', session=Session(), tasktype=TaskType.NORMAL)` Initializes a new `StreamService` object.
`URL`	`get_url(self, tasktype=TaskType.NORMAL)` Get the URL to be used to connect to this server.
`Stream`	`serve(self, timeout=-1.0, tasktype=TaskType.NORMAL)` Wait for incoming client connections.
	`close(self, timeout=0.0, tasktype=TaskType.NORMAL)` closes a stream service

Table 26: Methods from StreamService

	`__init__(self, url='', session=Session(), tasktype=TaskType.NORMAL)` Initializes a client stream for later connection to a server.
	`__def__(self, tasktype=TaskType.NORMAL)` Destroy a Stream object.
`URL`	`get_url(self, tasktype=TaskType.NORMAL)` Get the URL used for creating the stream.
`Context`	`get_context(self, tasktype=TaskType.NORMAL)` Return remote authorization info.
	`connect(self, tasktype=TaskType.NORMAL)` Establishes a connection to the target defined during the construction of the stream.
`int`	`wait(self, what, timeout=-1.0, tasktype=TaskType.NORMAL)` Check if stream is ready for reading/writing, or if it has entered an error state.
	`close(self, timeout=0.0, tasktype=TaskType.NORMAL)` Closes an active connection.
`int or string`	`read(self, size=-1, buf=None, tasktype=TaskType.NORMAL)` Read a data from a stream.
	`write(self, buf, size=-1, tasktype=TaskType.NORMAL)` Write a data buffer to stream.

Table 27: Methods from Stream

	`__init__(self, size, mode, data=None)` Initialize an parameter instance.
	`set_io_mode(self, mode)` set io mode
`Value from IOMode`	`get_io_mode(self)` Retrieve the current value for io mode

Table 28: Methods from Parameter

`__init__(self, session, funcname, tasktype=TaskType.NORMAL)` Initializes a remote function handle
`__del__(self)` Destroys the RPC object.
`call(self, parameters, tasktype=TaskType.NORMAL)` Call the remote procedure.
`close(self, timeout=0.0, tasktype=TaskType.NORMAL)` Closes the rpc handle instance.

Table 29: Methods from RPC

6 Implementation of the SAGA Python Language Binding

This section discusses the details of the implementation of the SAGA Python language binding. It gives some insight how JySaga, the implementation on top of the SAGA Java reference implementation, is implemented and how it works.

6.1 Previous Solution

The first idea of implementing the language binding involved creating a library that could switch between both SAGA reference implementations. The first outlines of the library could import the modules that targeted a specific SAGA reference implementation. The actual switching was done based on a environment variable. This structure was rejected because of multiple reasons. The first reason was the realization that Python applications were never able to switch between SAGA reference implementations, because they both run on different Python interpreters. The advantage of the switching between both SAGA implementations and Python interpreters would be minimal. The second reason is that the importing would clutter the namespace in the library module. That would not be a simple, clean and Python-like solution.

Another issue was that there is no way to enforce that implementations exactly follow the specification. Abstract methods are not available and Python does not have interfaces. We then decided that the structure and the source code of the Python language binding specification should be as simple as possible. Implementations of the SAGA Python language binding can just take the source code and fill in all the blanks. Programmers should make sure not to change any method declaration since there is no way to enforce the correctness of the implementation of the SAGA Python language binding other than passing the unit tests.

6.2 Structure of the SAGA Python Language Binding Implementation

The goal of our SAGA Python language binding implementation is to offer a Python look and feel while hiding the fact that the user is working on a non-Python SAGA implementation. This can be achieved by wrapping all the objects presented to the user by the SAGA implementation in a Python object. We name the object inside the Python object a delegate object. Almost all method calls on the Python object are delegated to the delegate object. An overview of this structure is given in Figure 4. In the figure a Python application calls a method in our implementation of the SAGA Python language binding, called JySaga. The method in JySaga checks the parameters and raises an exception if the parameters are not correct. If the parameters were correct, the method converts them to parameters the SAGA implementation understands, and does a method call on the delegate object with the converted parameters. The SAGA implementation then acts on this method call and raises an exception or returns a return value. A raised exception is catched in PySaga, converted to a Python exception and re-raised. If the SAGA implementation returns a return value or object, they are converted to Python values or wrapped in a Python object and then returned to the Python application.

The converting of the return value is often straight forward. Booleans, numbers, voids, chars and strings are often converted by the Jython interpreter, so we did not have to help the conversion process. This is different for arrays. Lists coming from the Python application have to be converted to Java arrays and arrays coming from the SAGA Java implementation have to be converted to lists. If the arrays contain Java objects, these objects have to be wrapped by a Python object before the list can be returned to the user. This principle of converting is used through-out JySaga.

The converting of Java exception to Python exceptions is done in the `convertException()` method that each SAGA Python object inherits from the `Object` class. This method inspects the type of the given exception and creates the Python equivalent of that exception. The message is then copied, together with an optional SAGA object attached to the exception. For example,

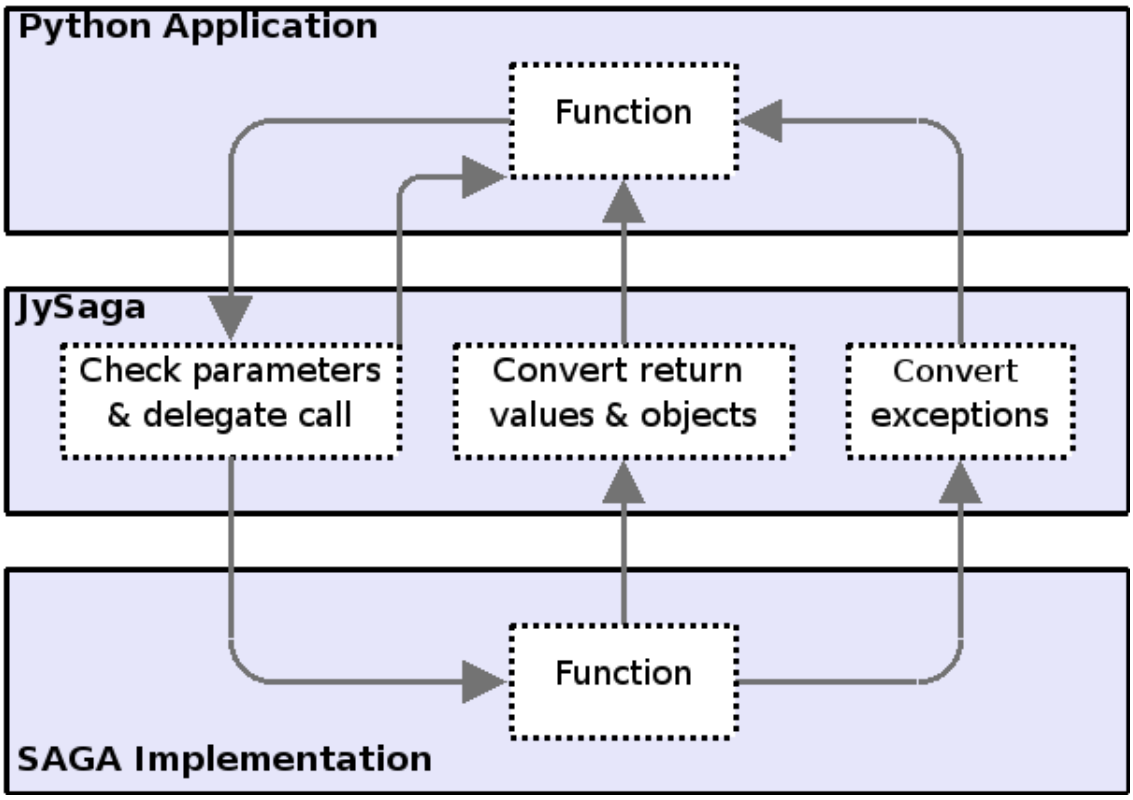

Figure 4: Overview of JySaga structure

a Java exception like `org.ogf.saga.error.DoesNotExistException` is converted to a Python `DoesNotExist` object.

Tasks object in JySaga can be created by calling the asynchronous version of a JySaga method. The Java delegate objects are stored in Python Task objects just like with other SAGA Python objects. The only implementation difficulty is returning the right type when a user calls the Task.get_result() method. JySaga cannot know beforehand which of the more then 25 different types is returned by the delegate object. JySaga has to inspect the type and create a new Python object to store the returned object in.

Cloning can be done in each method that inherits the clone() method from the Object . The cloning of a Python object also implies the cloning of the delegate object in the Python object. First the delegate object is cloned and then a new Python object is created. To prevent that in the process of the Python object creation, the Python object gets his own delegate object and not uses the cloned delegate object, a parameter is added to the `__init__()` method. The added `**impl` parameter is a dictionary of all parameters and values given but not declared in the method declaration. To create a new `Session` object and add the cloned delegate object at the same time, the call `clone = Session(delegateObject=javaClone)` is made. The `Session.__init__()` method recognizes the javaClone as a delegate object and uses it instead of calling the underlying SAGA implementation to get a new delegate object.

6.3 Modules in JySaga

This section gives some specific details about the SAGA packages in addition to the details given in the previous section. Not every possible detail is described in this thesis, since that would be to much information. For all the details, we recommend to read the source code. The source code is currently available at [25] and [1].

Error The exceptions in the `error` package are the only Python objects that have no delegate

objects. The methods in `SagaException` do not have difficult implementations as they just get and set variables.

Object `Object` defines the `convertException()` for the converting of Java exceptions to Python exceptions and the `clone()` method, both explained in the previous section. `get_type()` has a custom implementation in each subclass of `Object` and does not delegates this call. This is because the SAGA Java reference implementation does not have a `getType()` method call.

URL `URL` delegates every call without any conversion of parameter. Only the returned Java object from `translate()` is wrapped in a Python `URL` object.

Buffer `Buffer` has implemented a conversion for Python (unsigned) chars to (signed) Java bytes and back. It also makes sure that application managed buffers are updated after each change to the internal buffers. The internal buffer is a Java array of which a reference is given to the SAGA implementation.

Session `Session` mostly just delegates method calls and only has to convert the `Context` array to a list.

Context `Context` just delegates the method calls to the SAGA implementation. It uses the methods inherited from the `Attributes` class to implement the properties (see Section 5.1) defined in the SAGA Python language binding .

Permission `Permission` is a subclass of the `Async` and implements the task model. Each method of a class that is a `Async` subclass has four internal options to execute the method, and the option is chosen by the `tasktype` parameter. Unfortunately, the Java language has no notion of permissions, so each method of the Permission class returns a NotImplemented exception.

Attributes Methods of the `Attributes` class are used in combination with the properties to set the attributes of a class. In `Attributes`, conversions from lists to arrays occurs but overall, simply delegating the call is enough.

Monitoring In the monitoring package, `Callback` uses a `CallbackProxy` class be the link between the Python application and the SAGA implementation. `CallbackProxy` is a subclass of `org.ogf.saga.monitoring.Callback` and is usable for Java `addCallback()` methods. If a programmer adds a `Callback` through the `Monitorable.add_callback()` method, the `Callback` is wrapped in a `CallbackProxy` and given to the SAGA implementation. If the metric related to the `Callback` changes, the SAGA implementation calls `CallBackProxy.cb()`, which calls `Callback.cb()`.
The methods in `Metric`, `Monitorable` and `Steerable` contain no implementation details not documented in previously mentioned classes.

Task Besides the task model mentioned in the previous section, `Task` has some other specific details. If a Python application calls an asynchronous JySaga method with wrong parameters, JySaga will return a Task object in a FAILED state. Because the wrong parameters are detected by Jython and not by the SAGA implementation, no delegate object is returned and non-SAGA `TypeError` is raised. The `Task` object still functions without the delegate object and behaves like the `Task` already failed. If the Python application tries to add the failed `Task` to a `TaskContainer`, the `TaskContainer` will refuse to add the Task and will raise an exception.

Job `JobDescription` has no methods of its own, but relies on the methods from `Attributes` and properties for its functionality. `JobService` and `Job` function with just converting

objects delegating calls. `JobSelf` functions through all its inherited methods. `StdIO` is a wrapper around the `InputStream` and `OutputStream` from `java.io`. It relies on the `available()` method from the streams to check if there is data available and to check if the read on the stream will block or not. It designed to work like the standard input and output in Python, although Python's description of `readlines()` does not match with what Python's `readlines()` does.

Namespace `NSEntry` and `NSDirectory` just delegate the calls and converts the parameters and return values.

File `IOVec` uses a similar implementation as `Buffer`. `File` and `Directory` cast the long value from the SAGA implementation's `getSize()` to an integer if the value is smaller than the maximum value for an integer, as specified in PySaga. In normal Python use, users will never notice the difference between longs and integers because Jython will automatically convert them but because JySaga follows the specification of the SAGA Python language binding.

Logicalfile `LogicalFile` and `LogicalDirectory` just delegate the calls and converts the parameters and return values.

Stream `Stream` uses the properties to set the attributes of `Stream`. Using the `read()` call without specifying a buffer is possible for `Stream`. The maximum amount of data that can be read is 4096 bytes per call. Other than that, `Stream` and `StreamService` use the delegation principle.

RPC `Parameter` from the `rpc` package is a subclass of `Buffer`, but works different than `Buffer`. `Parameter` is extended to also use normal data types like integers and strings. To mimic a byte array, it uses a Python array of chars. Otherwise it uses lists of data types. Booleans are represented in Python as 1 and 0, so it hard to distinguish between a list of booleans and a list of integers. `RPC` uses a list of the `Parameters` in the `call()` method.

7 Testing

This section describes the testing of the SAGA Python language binding implementation. We will explain how it was tested and what difficulties we encountered.

7.1 Test Environment

PySaga was created using the the Eclipse IDE [11] version 3.2 using the the PyDev Python development plug-in [21] to develop the Python files. To create the HTML files, Epydoc 3.0beta1 was used. To test JySaga, Jython 2.2.1 was used on top of Java 1.5.0_09 which was running on Debian Linux. The version of the SAGA Java reference implementation was the latest version from the svn repository [24], as the released 1.0 version had some bugs which made testing impossible. To access the svn repository, users will have to create an account at the Gforge website.

7.2 Unit tests

The tests used to test the functionality are unit tests. These unit tests only test if a JySaga object implements the functionally specified in PySaga. It does this by checking if a method that is specified in PySaga exist and returns the right values and data types.It does not thoroughly test the underlying SAGA implementation, such as domain boundaries or specific exceptions for specific cases. This extensive testing would certainly expose bugs or unwanted features in the SAGA implementation, but is outside our scope.

The testing is bound to the underlying SAGA Java reference implementation, since JySaga cannot offer functionality that is not implemented in SAGA implementation. During the testing several bug reports were given to the SAGA Java reference implementation maintainer to notify him of missing functionality and bugs, which were fixed later on.

7.3 Performance

At this point, it is very hard to give a good impression of the overhead JySaga has. Comparing JySaga to the Python wrapper included with the SAGA C++ reference implementation has no meaning since they are two different software application written in different programming languages with one running on the Java virtual machine. The comparison will also use two different interpreters for the Python code and is not suitable to use in a test of performance.

The overhead of JySaga on top of the SAGA Java reference implementation consists out of two parts. The first part Jython. Jython interprets the Python code and has to transform the code to Java byte code. This part is expected to be a bigger overhead than the delegating of a method call, which takes two method call instead of one. Overhead is not a big issue if JySaga will be primarily used to steer long-running computing jobs and all work is done remotely. Overhead will be an issue if JySaga is heavily used to read and write data using the Buffers, since data is casted from chars to bytes and back again. A small test shows that reading a file of 3 Megabytes in a small Java application takes 12 milliseconds. If we add the conversion of bytes to a string, the program takes 220 milliseconds. The same application takes the same amount of time when converted to Jython. The application takes almost a second when using the current version of JySaga. This means that the JySaga can be optimized when taking full advantage of the conversion mechanisms offered in Java.

7.4 Integration of SAGA with Python

This master project has delivered a Python language binding for SAGA and an implementation for the language binding on top of the Java SAGA reference implementation. Programmers can use the specification of the language binding to use SAGA in their Python applications. Our opinion is that the specification is user friendly and conforms to how Python programmers would like to see the SAGA API. The specification consists of standard Python data types and looks like an easily understandable API.

The specification of the `Buffer`, `IOVec` and `Parameter` classes might look unorthodox but is specified this way because Python does not have a byte type. The classes were designed in GFD.90 as containers of bytes and therefore a translation had to be made. Suggestions were made on the mailing list to completely remove the buffer classes from the specifications and to use strings as representations of the data. We decided to have both principles of (application-managed and implementation-managed) buffers and strings. Strings are mostly used for simple read and write operations.

The specification includes one additional class that was not directly defined in GFD.90. The `job.StdIO` class represents the standard input, standard output and standard error of jobs. Because there can be multiple jobs, multiple representations of input and output streams must be available to the Python application. Standard input and output in Python are represented as a special file, and therefore the `job.StdIO` class has many methods that can also be found in the Python file class.

The language binding implementation is programmed to be as Python-like as possible, but only if that would not interfere with the principle that implementation details of the language binding should be hidden from the user. An example of this is the explicit checking of parameter types instead of using the 'duck typing' principle.

Without the checking, parameters would be directly passed to the SAGA Java implementation through Jython. Jython would try to fit the parameters to the available Java methods

of the SAGA Java implementation. If this fails for any reason, `TypeError` exceptions could be raised by Jython that are not clear to the user and could confuse him. In addition, TypeError is not an error defined by SAGA and should not be raised. The explicit type checking gives the user clear information when the wrong parameters were given. At this point, the parameters are checked in JySaga before the call to the SAGA Java implementation is made. In the future, this should be changed to first making the call, and checking the parameters only if the call has failed. Overall the implementation follows the Python programming style.

Since the SAGA Python language binding implementation is a 'glue layer' between Python applications and the Java SAGA reference implementation, the SAGA Python language binding implementation should also run on top of other SAGA Java implementations. A requirement is that other SAGA Java implementations must implement the SAGA Java language binding. Our Python language binding will not support classes or packages in addition to the SAGA Java language binding, but everything defined in the SAGA Java language binding should work. The SAGA Python language binding itself must be implemented in the 'glue layers' between the Python application and SAGA implementations in other programming languages. This is also shown in Figure 5 on page 40.

Currently, people from the Vrije Universiteit are working to create a wrapper around the Python wrapper of the C++ SAGA reference implementation. This wrapper will use our Python language binding specification. When the project is finished, it will be possible to write Python programs using the specification and run them on both the C++ and the Java reference implementations.

The Python wrapper which comes with the C++ reference implementation is designed bottom-up while our specification is specified top-down. The bottom-up approach lies close to the C++ implementation but is not explicitly designed for Python programmers or makes use of the specific features such as lists and default values for parameters. Our specification is designed without explicitly fitting it to a specific SAGA implementation.

8 Future Work

After the specification and implementation of the Python language binding for SAGA some work still has to be done. This sections explains which issues will need attention in the future.

8.1 Synchronize Specification with the Python Wrapper

At this point, there are both our specification and the specification which comes with the Python wrapper of the C++ reference implementation. They both differ at many points, such as naming of methods, variables and parameters. To reach the goal of creating a Python application which can run on both reference implementations these specifications need to be synchronized with each other.

This synchronizing has difficulties of its own such as technical and political and will probably result in sub-optimal solutions and loss of backward compatibility for the few Python applications which already use SAGA. At this point work is done to create a wrapper around the existing Python wrapper which uses the specification described in the thesis.

8.2 Extensions to the API

As more grid middleware and other technologies are developed, SAGA will probably need to be extended in the future. To access these new extension packages the Python language binding will have to be updated with the new functionality using the look and feel packages of the existing language binding.

Now that version 3.0 of Python (see Section 4.2.5) is released, the specification of language binding will have to be updated to reflect the changes and new data types, or a new specification will have to be made which is available in addition to a version which uses Python 2.x syntax.

There are some changes which effect our language binding. Version 3.0 implements a byte data type [19],[12] and a mutable byte buffer[5] as a byte array. These changes will severely simplify the specification and implementation of the `Buffer` class and the `Iovec` class. These classes now work with char arrays to emulate bytes. The Parameter class from the rpc package will also be helped with introduction of a byte data type because it simplifies the use of method calls which require byte arrays or byte buffers. These 'Python Enhancement Proposals' or PEPs would solve difficulties that were raised in the design of the SAGA Python language binding for the 2.x versions of Python.

New since Python 2.6 are abstract base classes [6]. These classes can have abstract methods without an implementation, much like interfaces in Java. These abstract base classes would be suitable to implement the interfaces defined in GFD.90.

Now that Python 3.0 is released, it will take some time before it is excepted as the mainstream version for Python programmers. In addition to that, it will take some time before the other implementations like Jython will fully support version 3.0. This is because the other implementations are based on the Python reference implementation and therefore will always lag behind in functionality. This means that even if a SAGA Python language binding is specified for Python 3.0, it can only be implemented in JySaga if Jython supports the syntax for Python 3.0.

8.3 Special Methods

Python uses a number of built-in methods to define certain behavior. Programmers can define or overload these methods to implement the custom behavior. An example of this is the `__init__()` method, defining what is initialized after the object is created. In the implementation of the language binding almost every class has a custom `__init__()` method. There are many special methods which can be defined like object (value) comparisons, operations on class attributes, operations like add and multiply or methods related to the representation of the class when printed. At this point, only the standard operations are specified and some have custom implementation like `__init__() and __new__()`. To improve usability more of these special methods have to be specified and implemented. These include `__len__()` to check the length or size, `__cmp__()` to compare objects or the special attribute methods.

Support for pickling also fall in this category. Pickling is what Java programmers may call serialization. It is the storing of Python objects to persistent media, such as a harddisk, to load them again a later time. These stored objects can be sent to other applications or used between different runs of the same application. The Python language binding implementation does not support this, but could be added if users need this functionality. Implementors then shall have to deal with storing the Python objects and the delegate objects, which is difficult if the underlying application does not support the serialization mechanism.

8.4 Updating Language Binding Implementations.

In addition to the points given in Section 8.2, the implementation of the language binding will also need to be changed if the underlying SAGA implementation is changed. This can happen if methods which previously threw a `NotImplemented` exception are now implemented or if bugs are solved or new features are implemented. Since the SAGA implementation is also bound to its language binding, change is not expected to happen often after a reliable working version is released. Still, it must be taken into account that change in the SAGA implementation means that change in the Python language binding is needed.

8.5 Consistency

The designing and implementing of the Python language binding was a learning process in which great amounts of experience were gained. An effect of this is that within the code of the specification and the implementation small things may differ and might not be consistent. This also include that not all the rules from the Python style guide are reflected within them. Making the code and declarations consistent will help the programmers when updating or changing the implementation. The current implementation also misses some comments about the working of the code in addition to the comments given in the document strings.

9 Conclusion

This thesis describes how we specified and implemented a SAGA Python language binding to give Python programmers a uniform set of classes and methods for programming grid-aware applications using SAGA. The complete picture is shown in Figure 5. JySaga is a software layer for Python on top of the SAGA Java reference implementation. It implements the classes and methods described in SAGA Python language binding. Python programmers can now create applications using SAGA, run them on Jython and use the SAGA Java reference implementation. When the project PySaga++, described in section 7.4, is finished it will be possible to run a Python application on both SAGA reference implementations.

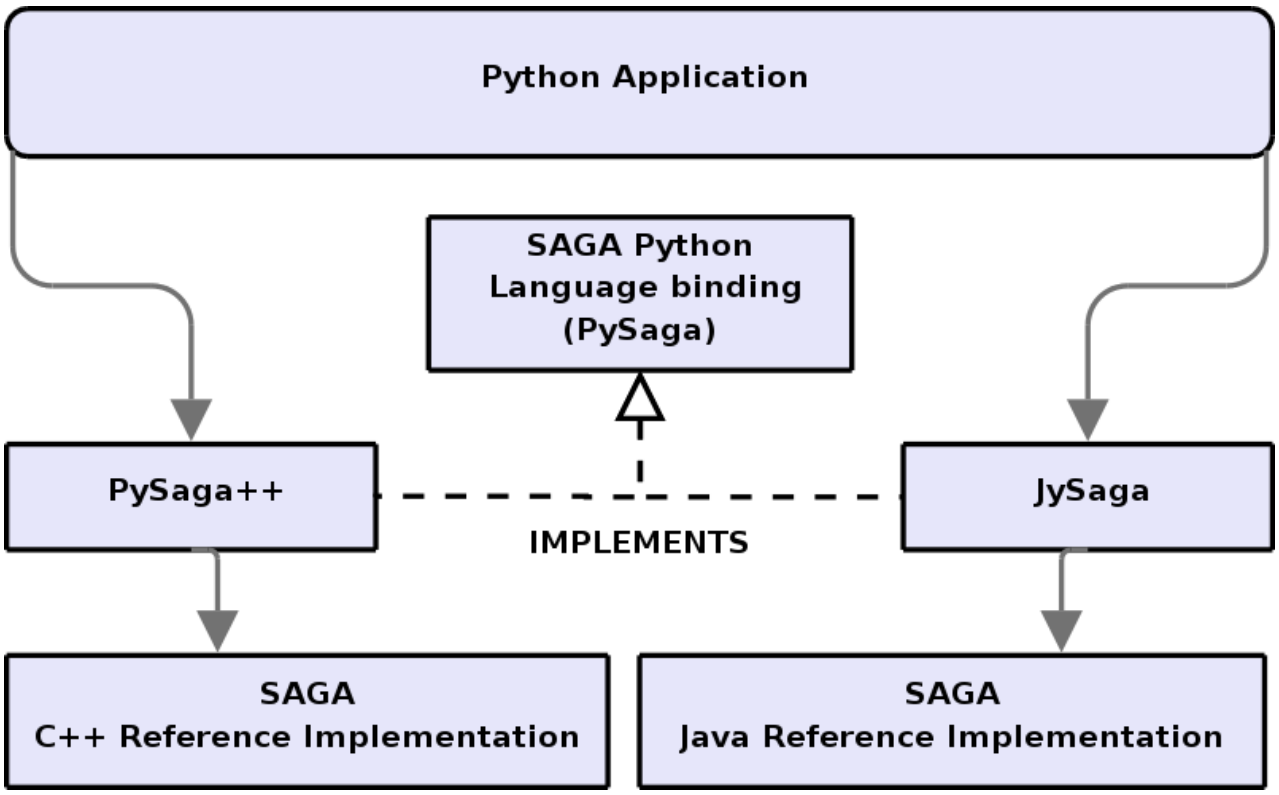

Figure 5: The position of PySaga and JySaga in the complete picture

10 Acknowledgments

I thank the following persons as they were essential in the whole process of this master thesis:

Thilo Kielmann and Mathijs den Burger for their supervision, support and invaluable inspiration. Ceriel Jacobs, Andre Merzky, Hartmut Kaiser, Steve Fisher and Manuel Franceschini for their technical support. Mom, Dad and Milou for their unlimited support.

11 Installing and Running JySaga

To run Python application on top of JySaga and the SAGA Java reference implementation, some things have to be arranged.

- You have to install the SAGA Java reference implementation and make sure it runs. To test this, you can try to run `/demo/demo2` from the installation directory of the SAGA Java reference implementation. Currently, it is best to download the trunk version from [24]. The SAGA Java reference implementation is located in the saga-implementation directory.
- You have to make sure that Jython is installed on your system. You can try to install it through the package manager of your operating system or download it from [16]. After installation, locate the `jython.jar` file as you need it later.
- Make sure that you have downloaded JySaga from [1]. The files are found the `/impl/jysaga/trunk` folder of the PySaga project.
- Make sure you have set the '`$JAVA_SAGA_LOCATION`' environment variable for the location of the SAGA Java implementation, the '`$JYTHON_JAR`' for the location of the `jython.jar` file, and optionally set the '`$JYSAGA`' for the location of the JySaga source files and the '`$JYTHON_HOME`' for Jython specific files.
- run the `./trunkStartJysagaApp` script to get the Jython interpreter running. Jython should know how to find the SAGA reference implementation and the JySaga files.

References

[1] *SAGA Python language binding API documentation.* http://gforge.cs.vu.nl/gf/project/pysaga/.

[2] Available JavaGAT adaptors. *http://www.cs.vu.nl/ibis/javagat-adaptors.html*.

[3] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schott, E. Seidel, and B. Ullmer. The grid application toolkit: Toward generic and easy application programming interfaces for the grid. In *Proceedings of the IEEE*, volume 93, pages 534–550, 2005.

[4] Differences between CPython and Jython. *http://jython.sourceforge.net/docs/differences.html*.

[5] Immutable Bytes and Mutable Buffer. *http://www.python.org/dev/peps/pep-3137*.

[6] Introducing Abstract Base Classes. *http://www.python.org/dev/peps/pep-3119*.

[7] Epydoc: Automatic API Documentation Generation for Python. *http://epydoc.sourceforge.net*.

[8] Style Guide for Python Code. *http://www.python.org/dev/peps/pep-0008*.

[9] Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Andre Merzky, John Shalf, and Christopher Smith. *A Simple API for Grid Applications (SAGA)*, 2008. http://www.ogf.org/documents/GFD.90.pdf.

[10] Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Gregor von Laszewski, Craig Lee, Andre Merzky, Hrabri Rajic, and John Shalf. Saga: A simple api for grid applications. In *Computational Methods in Science and Technology*, volume 12, pages 7–20, 2006.

[11] The Eclipse IDE. *http://www.eclipse.org*.

[12] Python Built in Functions. *http://docs.python.org/dev/3.0/library/functions.html*.

[13] IronPython. *http://www.codeplex.com/IronPython*.

[14] Shantenu Jha and Andre Merzky. *A Collection of Use Cases for a Simple API for Grid Applications*, 2006. http://www.ogf.org/documents/GFD.70.pdf.

[15] Shantenu Jha and Andre Merzky. *A Requirements Analysis for a Simple API for Grid Applications*, 2006. http://www.ogf.org/documents/GFD.71.pdf.

[16] Jython. *http://www.jython.org*.

[17] Hartmut Kaiser, Andre Merzky, Stephen Hirmer, and Gabrielle Allen. The saga c++ reference implementation. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'06) - Library-Centric Software Design (LCSD'06)*, Portland, OR, October, 22-26 2006.

[18] Boost C++ libraries. *http://www.boost.org*.

[19] Bytes literals in Python 3000. *http://www.python.org/dev/peps/pep-3112*.

[20] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. *A GridRPC Model and API for End-User Applications*, 2005. http://www.ogf.org/documents/GFD.52.pdf.

[21] The PyDev Python plugin for Eclipse. *http://pydev.sourceforge.net*.

[22] Python. *http://www,python.org*.

[23] Stackless Python. *http://www.stackless.com*.

[24] SAGA Java reference implementation source code. *http://gforge.cs.vu.nl/gf/project/ibis/scmsvn/*.

[25] The JySaga source code. *https://svn.cct.lsu.edu/repos/saga-projects/language-bindings/jysaga/*.

[26] TIOBE. Programming community index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[27] Vrije Universiteit. *SAGA Java Implementation*, 2008. http://saga.cct.lsu.edu/java.

[28] Louisiana State University. *http://www.cct.lsu.edu*.

[29] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. User-friendly and reliable grid computing based on imperfect middleware. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–11, New York, NY, USA, 2007. ACM.

[30] Gregor von Laszewski, Ian Foster, Jarek Gawor, and Peter Lane. A java commodity grid kit. In *Concurrency and Computation: Practice and Experience*, volume 13, pages 645–662, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, U.S.A, 2001. John Wiley I& Sons, Ltd.

[31] Simplified Wrapper and Interface Generator. *http://www.swig.org*.