CHALMERS

Rheos
A Domain-specific language for high-level sampling tasks in high-performance computing

Master of Science Thesis in the Programme
Computer Science: Algorithms, Langugages & Logic

MARTIN HARDSELIUS

VIKTOR ALMQVIST

Chalmers University of Technology
Department of Computer Science & Engineering
Göteborg, Sweden 2012

Rheos
A Domain-specific language for high-level sampling tasks in high-performance computing

MARTIN C. HARDSELIUS
VIKTOR A. ALMQVIST

Examiner: AARNE RANTA

Chalmers University of Technology
Department of Computer Science & Engineering
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Deparment of Computer Science & Engineering
Göteborg, Sweden, June 22, 2012

ABSTRACT

Many computations running on high-performing systems do not make use of the performance available. To solve this problem, software written to achieve strong scaling is needed.

Copernicus is a system for execution of large-scale sampling tasks in high-performance environments. It aims to achieve strong scaling, regardless of underlying architecture. The system was originally developed to run large scale bio-molecular simulations. However, lacking an intuitive way of describing computational projects, the developers felt a need for an user-friendly text-based input for Copernicus.

This master's thesis describes a design and implementation of a domain-specific language to meet the need of a suitable input description for Copernicus. The language design is simple yet manages to capture the abstract model of how a computational project is executed. The language is strongly typed and inspired by elements from both functional programming and data-flow languages, making Rheos a powerful descriptive domain-specific language.

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [12]

ACKNOWLEDGMENTS

We want to thank our supervisors at KTH Royal Institute of Technology; Erik Lindahl, Iman Pouya and Sander Pronk, for the opportunity to carry out this thesis and to take part of the Copernicus project. We also would like to thank prof. Aarne Ranta, at Chalmers University of Technology, for agreeing to be our examiner.

CONTENTS

Part I

DISTRIBUTED COMPUTING

1

INTRODUCTION

With the ever increasing need for storage and computational power, governments, research institutes and industry are rushing to adopt cloud computing, moving away from a model where computational projects are executed on local computers.

The communities of researchers that need access to the computational power required to carry out non-trivial simulations and analysis of data are often distributed geographically, as are the computing resources they rely on.

Both high-performance computing and embedded systems are moving towards many-core systems, and in the next few years we can expect to see platforms with 100's or more processor cores [13], maybe even 1000's or tens of 1000's cores. The K Computer, built by Fujitsu, contains more than 80,000 Nodes with eight cores each. Computational clusters and clouds built out of many-core systems will offer unprecedented quantities of computational resources. Scaling software and managing these resources will offer a tremendous challenge [14].

1.1 BACKGROUND

To run computations effectively on modern supercomputers and computer clusters the applications need *strong scaling*. In high-performance computing, strong scaling is defined as how the time-to-solution varies with the number of processor cores available for a fixed *total* problem size. When this is a limitation for applications, the available resources are not used to reach highest possible performance.

Many interesting real-world applications (all that are not embarrassingly parallel) require some inter-process communication for scaling and are therefore limited both by the availability of this bandwidth as well as the total amount of resources for high absolute performance [18].

Molecular dynamics simulations are computations which have inherent parallelization limits due to the finite number of particles simulated, but there is a possibility to achieve strong scaling since many of these computations are of statistical nature. Relying on sampling of many individual simulations makes it possible to distribute the

workload on supercomputers and compute clusters. Parallelization of such simulations gives a significant performance boost when a high number of cores are at disposal.

The high-level way of parallelization can generally be described by a very simple workflow. The workflow contains a simulation and an analysis stage with a feedback.

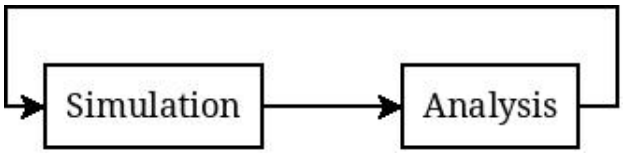

Figure 1: Parallelization workflow

The entire computation is initialized by generating a large number of small simulations. Each simulation will send the result data to the analysis stage. The analysis stage will analyze the data and create some result of the computation so far. In this workflow the analysis stage has a feedback to the simulation stage. This means the analysis stage will generate new directed simulation depending on the current result, i.e. what parts still needs more data. A computation like this is highly parallel and modular which gives a possibility of using more resources to speed up the entire computation.

An example of such a workflow would be a Markov state modeling. Grouping molecular simulations together depending on characteristics of the result would be the states in such a Markov state. The characteristics can be any property of molecules, including the its shape. A large number, in the order of thousands, of simulation states would start from different states and gather data for the analysis stage. The Markov state model gives a course-grained description of a simulated system, resulting in a transition matrix between states, and their static statistical weights. This can be used in an iterative way by identifying under-sampled regions and adaptively sample those, resulting in a feedback loop such as in Figure 1.

1.1.1 *Cloud computing*

Clouds are solutions to run computations on high-performing computer systems. Foster et al. [9] defines cloud computing as:

> *A large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet.*

The resources are opaque to the user who use a pre-defined API to run and use the system, an abstract layer that hides the underlying ar-

chitecture. Running molecular dynamics simulations on a cloud computing resource would need high parallelization, such as described above, to achieve a performance boost.

There are a few challenges when using clouds. Defining an API for users to discover, request and use resources provided by the cloud can be difficult. An API needs to have a good way of using the computational power to execute the users projects. The users should also be able to use all different features available in the cloud and the API needs to be simple enough so that any user can understand it without having knowledge of the cloud system behind the API.

The cloud needs to coordinate executions on the available resources when the computations are often highly parallel. Executions may even need to support different software and hardware.

Monitoring progress and resources is a challenge since the users are not in direct contact with the hardware which actually runs the application. "Essentially monitoring in clouds requires a fine balance of business application monitoring, enterprise server management, virtual machine monitoring, and hardware maintenance, and will be a significant challenge for cloud computing as it sees wider adoption and deployments." [9]

Provenance is basically a trace of the computations with all the necessary information (data sources, intermediate states). This is very important for researchers, in order to track the project and be able to recreate the results. Without this the an experiment would not be as useful to the researchers as it could be, for example to validate their findings. Users can save alot of computation hours when having access to provenance information. In some cases it is of great use to be able to change something and start from an intermediate state of a computation instead of starting from scratch. Provenance is a relatively unexplored area within cloud computing and can be challenging to provide for general applications.

One way of programming/using a cloud can be to use workflow systems. The workflow can be represented as a graph of individual executions of applications where the edges are dependencies and how data are passed between the applications. Users can submit these workflow schemes to the cloud using the API interface.

There is a cloud solution for running parallelized molecular simulations and it is called Copernicus.

1.1.2 *Copernicus*

Copernicus is a software system that is made to distribute and parallelize large molecular dynamics simulations. The system integrates elements from distributed computing, and applies them to more traditional high-performance compute clusters. By taking advantage of the fast interconnects that may be available on these compute envi-

ronments, individual simulations are parallelized as far as possible. This approach enables Copernicus to use orders-of-magnitude more cores than a traditional simulation run on a supercomputer, and it allows for larger-scale simulations than would be possible with purely distributed systems, while it reduces time-to-solution significantly.

The idea behind Copernicus is to exploit the inherent parallelism of ensemble simulation and to make use of advanced sampling algorithms, while keeping the performance advantages of massively parallel simulations. Such computations are called projects in the system.

> *A project is executed as a single job, but breaks it up into coupled individual parallel simulations over all available computational resources, with the single simulation as the individual work unit. While the software has been optimized for using multiple high-performance compute clusters, it works equally well with cloud computing instances or even individual workstations [18].*

To handle projects with many simulations as a single entity Copernicus needs to able to

- *match and distribute the individual simulations to the available computational resources,*
- *run simulations on a variety of remote platforms simultaneously: HPC clusters, workstations, cloud computing instances, etcetera,*
- *parallelize tasks to the maximum extent possible on each resource, and use adaptive coupling beyond this,*
- *allow flexibility in the types of projects tat can be run,*
- *perform real-time analysis of the running project,*
- *enable monitoring of running projects [18].*

Copernicus network structure contains three components: clients, servers and workers as seen in Figure 2.

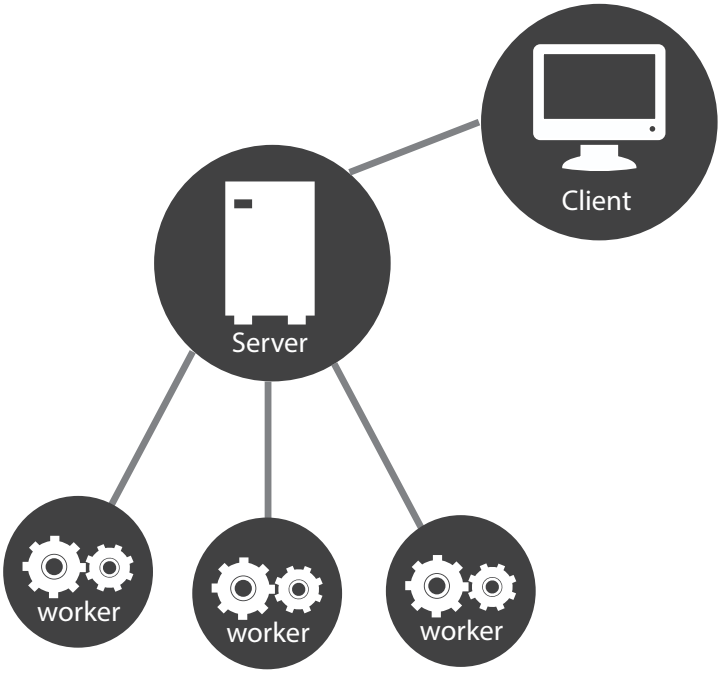

Figure 2: A client will issue a project request to the server. Once the server has received the project, it will divide it into jobs. Idle workers will ask the server for jobs to execute. The workers connected to the server may reside on different types of platforms; the first worker can be run on a supercomputer, the second on a cloud computing instance, while the third is run on a regular workstation.

The clients are user interfaces to interact with the system. Users will send and start their computational project to a server using the client. The server handles projects and controls the work distribution. Jobs will be sent to available workers, depending on which worker is best suited for the job. A worker will calculate the jobs assigned to it and send the result back to the server. It will also announce to servers when it is available. Multiple worker processes can be run on the same system, e.g. supercomputers would run a great number of workers to use all the available cores.

Servers are connected together in an open, but authenticated peer-to-peer network to support deployment on almost arbitrary topologies. Any Copernicus server can both send and receive commands, either from user clients, from workers, or from other servers. There is no top-level server in the architecture [18].

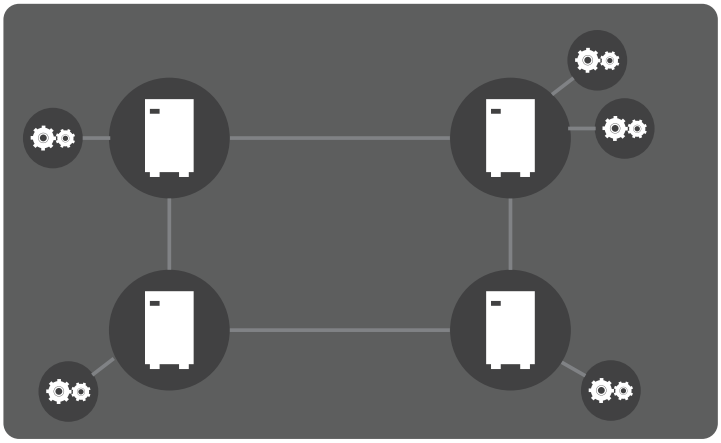

Figure 3: Servers are connected to each other in a peer-to-peer network. If a link goes down, traffic between servers will be re-routed dynamically.

Copernicus projects are described by building computational *data-flow networks*. Data-flow networks are networks which describe how streams of data is sent between different executions. A network is a set of connections between black boxes, where a black box can either be a function or another network. Both functions and networks have external inputs and outputs which are used to connect the networks between scopes. A function has a sub-network and a controller which both can't be accessed outside the function. The sub-network is a normal network where the controller can add connections and black boxes. A controller is in itself a black box in Copernicus. It has access to the networks definition and has permission to add new black boxes and connections. A real life example of a project is shown in Figure 4.

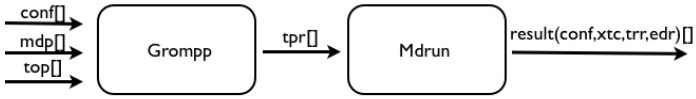

Figure 4: Copernicus project model

Both *Grompp* and *Mdrun* is a part of *Gromacs* which is a molecular dynamics package primarily designed for biomolecular system. Grompp is a pre-processor for simulations, which include checks of validity of the input. Mdrun is the main engine in Gromacs where the actual simulation happens. In this project Grompp takes inputs and generates topology files. The list `conf[]` is the configurations for each simulation. Mdrun then takes the topology files as inputs and runs the simulations. Even though this example does not have a feedback to generate more simulations, the Copernicus system is still very useful. When having access to limit computation hours on different systems, users could just split the list `conf[]` and split up the jobs, while just repeating the same workflow with different inputs, and still get

the same data. If this was not possible the user would need save the state of the computation and send potentially very large amount of data between the systems.

A problem with Copernicus has been the lack of an intuitive way of feeding input to the system, and that is why this project was formed.

1.2 PROBLEM STATEMENT

The objective is to find and implement a solution for the need of a new way of giving Copernicus information of the users projects. The developers specifically stated that the wanted a domain-specific language (DSL) for this solution, and that they later on want to add a graphical solution using this DSL.

The DSL should allow users of Copernicus to define their computational projects. The projects should be able to be defined as piping computations in a data-flow network, which means that the DSL needs to be able to describe data-flow networks in plain text.

The intended users are assumed to possess some knowledge of programming, but are not necessarily adept programmers. The design of the DSL should therefore be simple and intuitive. The DSL needs to be easy to understand so it becomes an asset instead of an hindrance.

The DSL should be fully functional in Copernicus. The users needs to be able to use all the features and properties available in Copernicus.

Copernicus has function libraries which needs to be usable in the language. This implies a certain amount of flexibility since there are not a static amount of libraries, as new ones can be added. The DSL should be able to cope with any new plug-ins.

The implementation should have an output of a form so that it can easily be integrated in the Copernicus system. The implementation also needs to be easy to install on any system, supercomputer or other.

1.2.1 *Delimitations*

The most important part of the project is to have a working implementation. However, since a language that just replaces the current XML descriptions of projects is quite limited, more effort was spent on adding features making the DSL more powerful. There are features which can be added to the DSL for describing even more advanced projects with better syntax, e.g. simple arithmetic expressions.

There implementation should allow the user to compile the code into XML, since Copernicus already has support for reading XML files describing computational projects. This will most likely be replaced by building the projects directly from the abstract syntax trees, rendering the XML generation redundant. This step would require

much more understanding of how Copernicus work, which is not within the time limit of this project.

This project has not considered a graphical solution at all, besides that it was designed with an easy translation between graphical model and code in mind. Graphical implementations of related work has been used get inspiration for the DSL. A graphical interface would be a good addition, and such a solution can be implemented as a front-end to the domain-specific language later on.

The language we chose to write the implementation in is Python. This choice was based on foremost an easy implementation and maintenance, since Copernicus is written in Python. It would have been possible to use effective tools and another language, but instead tools specifically for python were used.

1.3 RELATED WORK

1.3.1 *MapReduce*

MapReduce [8] is a programming model for distributed processing and generating large data sets. The idea is to have a "map" step and a "reduce" step. The map step means that a master node divides its problem and distributes it to worker nodes. The worker nodes can act as a master node to other worker nodes and distribute problems to them. This makes a tree-like process of dealing with the problems. The reduce step means that a master node collects the answers from its workers and combines them into one answer.

1.3.2 *Hadoop*

Hadoop [10] is a software framework written in Java to support distributed applications. Hadoop was derived from MapReduce and Google's File System. It is designed to scale from a single server to a cluster of computers to make use of clusters computational power. This is a big difference from Copernicus peer-to-peer styled networking. It handles failures at the application level.

To link Hadoop with applications, a C++ API and library is provided. Using an API like this defeats the purpose of having an easy and intuitive way of describing computational projects, but for large scale commercial implementation used by professional programmers, as Hadoop seems to be aimed towards, this becomes a viable option.

1.4 REMAINING CHAPTERS

The chapters in the next part will cover the different parts of the project in a fashion fairly close to the different stages the project went through.

The research conducted to gather domain knowledge is presented in Chapter 2. In Chapter 3, the language, its features and various design choices will be covered. The actual implementation details are described in Chapter 4.

Part II

CREATING A DOMAIN-SPECIFIC LANGAUGE

2

RESEARCH

This chapter presents some of the research that was done on Copernicus, different programming paradigms suitable for the problem domain, and existing programming languages belonging to those paradigms.

2.1 PROGRAMMING PARADIGMS

Different problem domains call for different programming paradigms. The execution model of *Copernicus* can be thought of as a flow network, which makes some paradigms more interesting, and the domain-specific language should be made to reflect that fact. The data-flow programming model contrasts the classical control flow model implemented in languages such as C.

Applications written in e. g. C have inherent limitations when run in parallel environments, because of the top-down sequential programming approach. The data-flow model consists of nodes connected to each-other to express the logical execution flow, and it can easily be used to express parallelism.

2.1.1 *Data-flow Programming*

The origin of data-flow languages is related the ever increasing need for parallelism in today's applications. Data-flow programming is a paradigm which has an execution model where a program is represented as directed graph. The data flows between operations along the arcs. Directed arcs represent dependencies between instructions. Arcs that flow toward a node are called *inputs*, while arcs flowing away from a node are called *outputs* [11]. The model focuses on how components of the program *connects* in contrast to the classical Von Neuman model, which focuses on *how they happen*.

2.1.2 *Flow-Based Programming*

In flow-based programming (FBP), applications are defined as networks of "black box" processes. Data is exchanged across predefined

connections via message passing. The black box processes can be reconnected in different ways to form new applications while their internals remain unchanged, thus making FBP a *component-oriented* approach [15, 16].

2.1.3 *Reactive Programming*

Reactive programming is oriented around data flows and the propagation of change. The key ideas are notions of *behaviors* and *events*, where behaviors are reactive values that varies over time, while events are time-ordered sequences of discrete-time event occurrences [19]. The underlying execution model will automatically propagate changes through the data flow.

2.2 PROGRAMMING LANGUAGES

There are some existing implementations of FBP out there. However, these implementations mainly consist of language extensions or libraries for general-purpose languages. Some of these language extensions includes *THREADS, JavaFBP, C#FBP* and *DrawFBP* [15].

3

LANGUAGE

This chapter describes the domain-specific language, as well as some rationale behind the design and future work.

For a complete description of the grammar in Backus-Naur Form (BNF), see Appendix A. The implementation details will be described further in Chapter 4.

3.1 DESIGN

Designing the DSL was a process which continued through the entire project. As there are no comparable packages and no text-based solution to similar problems, the DSL had no real starting base. The initial inspiration came from the research on programming paradigm's and graphical implementations related to network based programming.

Inspiration from well known programming languages was included for the DSL to be simple and intuitive to the common user. Both functional and imperative languages were considered when developing the design.

The most important steps in this process was a continuing discussion with the developers of Copernicus. It was important to have a DSL which they were satisfied with, but also to get input on what design choices to make. The developers perspective was important for the DSL to reflect realistic scenarios and to get a better collective view of the different solutions. At each meeting the developers was presented with a draft of the latest version of the DSL.

3.2 GENERAL STYLE & FEATURES

The DSL is a descriptive language which has four types of top-level syntactic definitions: atoms, networks, imports and new types. Atoms and networks are *components* which are connected to build project networks in Copernicus. The four top-level definitions are described more in detail in the following sections. All the top-level definitions needs to be defined before used in the current version of the DSL.

Atoms and networks both have a set of external inputs and outputs. An input/output does not need to have a connection since atoms/networks only evaluates once it has all non-optional inputs has received

a new value. It is possible to add new atoms/networks and connections in the Copernicus system, but as there are currently no interactive implementation of the DSL such actions cannot be done with this version of the solution. An input/output can be assigned constant values if they are of primitive types. These features are explained in more detail in this chapter.

3.3 MODULES

Modules have no intrinsic function in the current implementation of the language. A module is essentially a code file. Importing a file is importing a module. This is how to import the file "path/to/file.cod" from sub-directories of the current files path:

```
import path.to.file
```

All code from the imported module is simply added where the import statement is located in the code. This way the code from the imported file will be used as it would have been written in the current file. This system is not designed or developed to have modules and packages as wrappers for code, but rather to sort code in different files.

Currently it is only possible to import files from sub-directories. There are no way to import from other paths or any form of a standard library. The discussion of a more advanced and useful import system revealed that the developers had some different suggestions but had not decided how such a system should work with client server setup in Copernicus.

3.4 TYPING

In the language, all connections between executables are typed. Their types are set when inputs and outputs are defined. Atoms and even networks have types according to their external inputs and outputs.

Types are matched, without regard to type hierarchy, but to what set of primitive types can be inferred from them, and what dimension.

The type-checker does not type-check executables. Instead the system relies on the correct definition when wrapping them in the language.

3.4.1 *Primitive types*

The first primitive, integer, are represented by `int`. Integers cannot be assigned floating point values, i. e.no type-casting, but has to be assigned integers. Floating point types are represented by `float`, and as integers floating points cannot be assigned anything other than

floating point values. This means that `1.0` is not the same as `1` in the language and will cause a type error if assigned to a connection with the wrong type.

Strings are represented by `string`. A constant strings is written between the two quotation symbols. The last primitive type is files, which are described as `file`. Files are paths written like strings, but are not checked by the language. Instead, they are maintained by Copernicus.

3.4.2 *Compound types*

There are two compound types in the language. The first one are arrays, which are ordered lists of a certain type. An integer array is written like `int[]`, where each pair of brackets represent one dimension. It is possible to have any number of dimensions of arrays. Accessing the fourth element of a variable `x` which is typed as the example integer array, is written `x[4]`. The resulting type of accessing elements is the same type with one less dimension. In the example that would be an integer (`int`).

The second compound type is records. They are ordered sets of elements which are assigned names. This make it possible to access elements in a record by both its index and name. Accessing the fourth variable `e` of a record `x` using its name would be written as `x.e`, and to access `x` by its index would be written as `x(4)`. The elements can have any kind of type and does not have to have the same type as the other elements in the record. A new type has to be defined to represent a record.

3.4.3 *New record types*

Defining new types are the way records are described in the language. A new type needs a name and which types of elements it contains. Each element needs a type and a unique name in the record. New types can only be defined outside networks and atoms.

The syntax for defining a new type begins with the keyword `type`. Consider a record called `setting` containing a file called `f` and a string called `name`. The following code represents this record `setting`.

```
type setting
    ( file  : f
    , float : name )
```

The file has in this case index zero and the file has index one. They can be access with both name and index as described above.

3.4.4 *Atom & Network type*

Atoms and networks are components used to build networks with. They work like black boxes in data-flow networks. As components are able to have any number of inputs and outputs their types are a special case of a record. Such records contain three elements: inputs, outputs, meta parameters. The inputs and outputs are records as well, in which every input and output parameter are elements. The meta parameter is a record which should contain types and is more explained in Section 3.4.6.

A component with an array of type `setting` called `settings` and an integer called `length` as inputs and a file called `output` as outputs, is described by the following syntax.

```
in ( setting[] settings
   , int       length  )
out ( file output )
```

The full header syntax for networks and atoms is described in Section 3.6 and Section 3.7.

Since both the type of `in`, `out` and the type of functions are records, there are numerous ways of referring to an input parameter. Referring to the a parameter `length` when a function of the same type as the above has been instantiated as `func` may look like `func.in.length` or `func(0)(1)` which both are the same thing.

3.4.5 *The 'network' type*

Besides the syntactic objects `network`, which have types described in the previous section, there is a type `network`. The type `network` is mainly used with controllers, as seen in Section 3.8.3. The type `network` refers to a record containing a set of instantiated components and a set of connections.

3.4.6 *Meta types*

Components can take types as inputs, called meta types, to be able to define generic components in networks. The parameters are defined with a variable name and a type group. There are currently three different type groups: `func`, `list`, and `type`. The group is a constraint on the type parameter which accepts only certain types. The group `type` means the parameter only accepts primitive types, `list` accepts compound type (records and arrays), and `func` accepts record types which at least have an element `in` and an element `out`. This way one can define new types which can be used to describe types of components.

The type parameters are defined and as the following example:

```
< func f , list l , type t >

in ( f.in i
   , l*   inputs
out ( t[] outputs )
```

`i` will be of a record type since `in` is a record. The symbol * removes one dimension from the array `l`. The output `outputs` will be a one dimensional array where the type is given by the parameter `t`.

3.5 INSTANCE NAMES

Instance names refer to instantiated Components. They are instantiated inside networks, where they can be connected to other instance names or external inputs and outputs. How to instantiate an instance name is described in Section 3.8.

Instance names contains letters, numbers, and underscores, but they have to start with a lower case letter.

3.6 ATOMS

An atom is a wrapper for executable code, python scripts and functions. In the language atoms are components just like networks and the information of what to execute and how is hidden inside the language for a more intuitive way of building project networks. Executables should be wrapped and added to an appropriate library so users do not have to concern themselves with external (outside the language) project design.

An atom has a header and an option part. The header contains the name of the atom, the type parameter definition, the type signature of the outputs and inputs, and what type of executable the atom uses. Execution is implementation-specific. Currently, there are three different types of executables in the current version: `python`, `python-extended`, and `external`. `python` means that the atom calls built-in functions of python, `python-extended` means the atom calls python scripts, and `external` calls binary executables.

The following code is the header for an atom `someatom` which calls a binary executable and has the type definition used in previous sections (note that in this case the type parameters are not used but are there to give a full description of a header).

```
atom external someatom < func f , list l , type t >
  in  ( setting[] settings
      , int       length  )
  out ( file output )
```

The option part is a list of options and values. The options are the information on what and how to execute for Copernicus. The values

are relative to Copernicus and are not a part of the language, which is why users should not have to write atoms themselves but import and use pre-defined atoms.

The following code is the definition of an atom `add` which uses a built-in python function to add two floating points together.

```
atom python add
  in  ( float a
      , float b )
  out ( float o )
  options ( fuction : builtin.float.add
          , import  : builtin.float )
```

3.7 NETWORKS

A network is a description of what to instantiate and how the components and external input/outputs are connected inside the network. Networks can be components the same way as atoms, which makes them sub-networks when instantiated in other networks. Each network has its own scope of type variables and instance names so the external inputs/outputs is needed to make connections to an external component.

The header of a network differs from atoms headers on two points. The key word `atom` is replaced with `network` and a network does not have an executable type. Writing a network `somenet` with the same parameters and type signature as `someatom` looks like this:

```
network somenet < func f , list l , type t >
  in  ( setting[] settings
      , int       length  )
  out ( file output )
  {
    var = someatom (in.settings)
    out.output <- var.out.o
  }
```

The second part of a network is its network body which is a list of statements separated by new lines. These statements are the description of the network, and they are explained in detail in the next section.

3.8 NETWORK DESCRIPTION STATEMENTS

There are three types of statements: assignment, connections, and a controller statement. With these statements a network can be described inside a network body. It is not possible to build networks outside a network definition.

3.8.1 *Assignment*

Assignment statements are instantiations of components which are assigned to instance names. An assignment needs an expression with information of what component and with which meta types and inputs is going to be instantiated. All the type arguments needs to be given when instantiated, but as it is not mandatory to connect something to an input the input arguments can be left empty. The arguments are connected to their respective input in the order they are defined in the component.

The type arguments are listed between the symbols `<` and `>`. They have to be defined on instantiation and are type references as types are referenced in Section 3.4.6 about meta types. The following example is an instantiation of the network `somenet` without any input arguments, hence the empty list `()`. The first type argument is the type of the atom `add` which is a record, the second type argument is the record of inputs for `add`, and the last type argument is the primitive type `float`.

```
var = somenet < add , add.in , float> ()
```

The input arguments are listed between the symbols `(` and `)`. An input argument can be a reference to a primitive, record or an array. The references are the same thing as accessing elements of compound types as in Section 3.4.2. It is also possible to assign constant values to inputs by.

The following code line instantiates an atom `add` and assigns it to an instance name `var`. The first input of `add` is assigned the constant value `1.1` and the second input is connected to the current networks input called `fp`.

```
var = add ( 1.1 , in.fp )
```

As mentioned, it is not necessary to supply all the input with values or connections so it is possible to remove `, in.fp` and the instantiation would still work (where the first input would still be assigned the constant value).

It is possible to instantiate a component as an input argument expression, where a specific output is connected to the respective input. In the following example an atom `mul` is instantiated and its output `o` is connected to the second input of the atom `add`. The atom `mul` is instantiated with its arguments inside parentheses and `.out.o` refers to the output which should be connected to the second input of `add`.

```
var = add ( 1.1 , (mul ( 2.0 , in.fp )).out.o)
```

3.8.2 *Connection*

Connection statements are another way to describe how inputs and outputs are connected. The left-hand side is the *destination* and the right-hand side is the *source*. It is not possible to refer to an output of the external network component or an input of an instantiated component in the right-hand side. Similarly, it is not possible to make a connection where the destination is an input of the external component. The inputs and outputs are references like the arguments in an assignment statement. The following code line connects the the output variable `o` of the instantiated instance name `var` to the external output `o` of the network.

```
out.o <- var.out.o
```

It is also possible to assign constant values to connections.

```
out.o <- 4.3
```

3.8.3 *Controllers*

When a component is set to be a controller, it is given permission to instantiate components and add connections within the current network. An example of when this could be useful is mapping a function (atom) over an array. The controller would build an instantiation of the function for each element in the input array and connect it the the appropriate element in an output array. For users to create their own controllers, they would need to know the internals of Copernicus and wrap an executable in an atom.

The following code instantiates `somecontroller` and supplies an array of type `setting`, and then sets the input `var.in.net` as the input network to the controller and the output `var.out.net` as the output network of the controller.

```
var = somecontroller (in.settings)
controller(var.in.net,var.out.net)
```

The output network is basically the new network setup, which means the instantiated `somecontroller` `var` can add components and connections.

3.9 DOCUMENTATION & COMMENTS

It is possible to add documentation strings to components and their inputs and outputs. The documentation string starts and ends with `'''`, and the documentation will be sent to The Copernicus system.

```
atom python add   '''Add two floating point numbers: q=a+b'''
  in  ( float a
      , float b )
  out ( float o   '''a+b'''
      )
  options ( fuction : builtin.float.add
          , import  : builtin.float )
```

The language has both line and comments block. A line comment start with the symbol # and anything after it will be skipped by the parser. Comments block starts with /# and ends with #/. Anything within the comment block will be skipped by the parser.

```
# This is a comment line

/#
This is a comment block
.
.
.
This is still in the comment block
/#
```

3.10 FUTURE WORK

3.10.1 *Modules*

The current import system in Copernicus is to leave the import mechanism to the server side. This may be discussed if it is optimal, and the developers of Copernicus had some ideas of how it could may be changed or improved. Even though changing these mechanics does not change the semantics for importing, it may affect the user of the DSL.

Importing of standard libraries needs to be well-defined before the client/server issues in Copernicus can be addressed. This may be an easy problem but it is still vital for continuing the implementation. Once this is done the developers needs to decide where the code should be parsed and type checked. The client can do all this work and send it to the server but as the rest of the client is very light and does not do much work, other than communicate with the server, a preferred solution is to have the server do all the work. The client can still add imports and build one code file for the server or send all the used code files. As there are a difference between importing from the standard library and importing user written project specific files, it needs to be decided how the client and server will work together to build the project. If the server should do most of the work it needs to have all the libraries.

Assuming the server would get all user written files, building projects, and doing all the work, a question is what the client should be able to do. Users may for example be able to type check their projects before sending them to a server, as it would make development easier. If so, the client needs to have access to all the libraries the server does.

There were a suggestion for a repository containing all libraries, where code can be added and fetched when not available locally. Users could easily add their own implementations. Such a repository needs to handle executables as well, since servers needs be able to supply them to the workers.

3.10.2 *Transpose*

There is a need for a transpose function for types. The transpose function would transform a record of any number of arrays to an array of records. It would also work the other way around. An example of when transpose could be used would be when defining a map component. One would want to send in a lists for each argument of the component in question, and transpose the argument lists to a list of records. This way each one of the records represents the inputs for a single instance of the component. Consider the following two types:

```
type a ( int[]   : a0
       , float[] : a1 )

type b ( int   : b0
       , float : b1 )
```

The code `transpose(a)` would be the type `b[]`, and the code `transpose(b[])` would be the type `a`. The transpose would work much like a zip for types which can take any kind of record or array of record.

4

IMPLEMENTATION

4.1 TOOLS

This section will describe the tools that were used to implement the Rheos language. It will serve not only as documentation for the language, but also as a reference for someone who might want to create their own language using the the same tools.

4.1.1 *Python*

The implementation language used for building Rheos was python. However, python was not the first language considered. Other languages considered were *C, C++, C#, F#, Haskell* and *Java*. However, C# and F# was never really an option, since both implementation, compilation and execution of code were tied to Unix environments. The reason behind the consideration of the other languages was that they are all supported by the *BNF Converter* (BNFC) [6]. BNFC is a compiler construction for generating a compiler front-end from a *Labeled BNF grammar*. Given this grammar, the tool produces (1) an abstract syntax implementation; (2) a case skeleton for the abstract syntax in the same language; (3) a lexer generator file; (4) a parser generator file; (5) a pretty-printer module; (6) a LaTeX file containing a specification of the language [6]. While a compiler generator certainly would have made the implementation a lot easier, the Copernicus system is written in python, and python does not exist as a target for the BNF Converter.

Copernicus is designed to run on Unix machines with as few dependencies as possible, which makes Java an unsuitable candidate, since it cannot be assumed that every candidate node has a Java runtime environment.

Doing the implementation in the C language could have been a possible solution, since it integrates well with python. Most Unix system does indeed ship with *gcc* or the *GNU Compiler Collection*. However, some older Unix distributions will not have gcc pre-installed, and others like recent versions of *Solaris* and *OpenSolaris* will have gcc under a different location.

Haskell was ruled out due to the simple fact that it is not as mainstream as the other languages. While Haskell is a very powerful language for writing compilers, maintenance of the code base might prove difficult for inexperienced users.

Virtually every Unix system ships with a python interpreter, and it is natural to write python extensions to a system already written in Python. Python is easy to learn and the code is easy to extend and maintain. In spite of python not being a classical meta-programming language, it became the implementation language of choice.

4.1.2 *PLY (Python Lex-Yacc)*

PLY is an implementation of `lex` and `yacc` parsing tools, written purely in python, by Beazley [4]. It was originally developed for an introductory class on compilers back in 2001. It provides most of the standard lex/yacc features including support for empty productions, precedence rules, error recovery, and support for ambiguous grammars. It uses LR-parsing, which is a reasonable parsing scheme for larger grammars, but slightly restricts the type of grammars that can be written [1]. PLY is straight-forward to use, and one its many advantages is the *very* extensive error checking, which certainly makes life easier.

PYTHON LEX The first step to implement the language is to write a tokenizer. This is done with the Lex module of PLY. Language tokens are recognized using regular expressions, and the steps are straightforward.

The names of all the token types are declared as a list of strings named `tokens`.

Listing 1: The token list

```
class RheosLexer(object):

...

    tokens = [
        # Literals: identifier, type, integer constant, float
        # constant, string constant
        'IDENT', 'ICONST', 'FCONST', 'SCONST', 'DOCSTRING',

        # Assignments: = :
        'EQUALS', 'COLON',

        # Connection: <-
        'CONNECTION',

        # Delimiters: ( ) { } [ ] , .
        'LPAREN', 'RPAREN',
        'LBRACE', 'RBRACE',
```

```
        'LBRACKET', 'RBRACKET',
        'COMMA', 'PERIOD',

        # Other:
        'CR', 'OPTIONAL', 'OPTIONS'
    ]
```

Tokens that require no special processing are declared using module-level variables prefixed by `t_`, where the name following `t_` has to exactly match some string in the tokens list. Each such variable contains a regular expression string that matches the respective token (Python raw strings are usually used since they are the most convenient way to write regular expression strings).

Listing 2: Token variables

```
class RheosLexer(object):

...

    t_EQUALS     = r'='
    t_COLON      = r':'
    t_CONNECTION = r'<-'
    t_LPAREN     = r'\('
    t_RPAREN     = r'\)'
    t_LBRACKET   = r'\['
    t_RBRACKET   = r'\]'
    t_LBRACE     = r'\{'
    t_RBRACE     = r'\}'
    t_COMMA      = r','
    t_PERIOD     = r'\.'
    t_OPTIONAL   = r'\?'
```

When tokens do require special processing, a token rule can be specified as a function. For example, this rule matches numbers and converts the string into a Python integer.

Listing 3: Token functions

```
    def t_ICONST(self, t):
        r'\d+'
        t.value = int(t.value)
        return t
```

In some cases, we may want to build tokens from more complex regular expressions. For example:

Listing 4: Complex regular expressions

```
class RheosLexer(object):

...

    lowercase    = r'[a-z]'
    identchar    = r'[_A-Za-z0-9-]'
    ident        = r'(' + lowercase + r'(' + identchar + r')*)'
```

```
    def t_IDENT(self, t):
        # we want the doc-string to be the identifier above
        ...
```

This is not possible to specify using a normal doc-string. The programmer would have to write the full RE, defeating the purpose of re-usable code. However, there is a way around this by using the `@TOKEN` decorator.

Listing 5: Token decoratior

```
from ply.lex import TOKEN

class CodspeechLexer(object):

...


    lowercase    = r'[a-z]'
    identchar    = r'[_A-Za-z0-9-]'
    ident        = r'(' + lowercase + r'(' + identchar + r')*)'

    @TOKEN(ident)
    def t_IDENT(self, t):
        t.type = self.keyword_map.get(t.value,"IDENT")
        return t
```

The observant reader might notice something special going on in the function `t_IDENT`. The processed string is checked against a keyword map to decide whether the token type should actually be `IDENT` or something else. The keyword map is defined as a dictionary, and the values are appended to the token list.

Listing 6: Keyword map

```
class RheosLexer(object):

...

    keyword_map = {
        # Import
        'import'          : 'IMPORT',

        # Type
        'type'            : 'TYPE',

        # Atom keywords
        'atom'            : 'ATOM',
        #'options'         : 'OPTIONS',
        'python'          : 'ATOMTYPE',
        'python-extended' : 'ATOMTYPE',
        'external'        : 'ATOMTYPE',

        # Network
        'network'         : 'NETWORK',
```

```
    'controller'      : 'CONTROLLER',

    # Header
    'in'              : 'IN',
    'out'             : 'OUT',
    'default'         : 'DEFAULT',

    # Types
    'file'            : 'FILE',
    'float'           : 'FLOAT',
    'int'             : 'INT',
    'string'          : 'STRING',
}


tokens = [
    ...
] + list(set(keyword_map.values()))
```

Since our keyword map contains multiple keys mapping to the same value and the token list can not contain any duplicates, the list of values is converted to a set before it is converted back into a list.

PYTHON YACC The `yacc.py` module is used to parse the language syntax. The grammar of a programming language is often specified in *Backus-Naur Form* (BNF). For example, some simple grammar rules for parsing types could look like this:

⟨*type*⟩ ::= `'float'`
| `'int'`
| `'string'`
| ⟨*type*⟩ ⟨*dim*⟩

⟨*dim*⟩ ::= `'[]'`
| ⟨*dim*⟩ `'[]'`

Figure 5: An example grammar for type identifiers

The identifiers *type* and *dim* refer to grammar rules comprised of a collection of *terminals* and *non-terminals*. The symbols `float`, `int`, `string` and `[]` are known as the *terminals* and correspond to raw input tokens. The *non-terminals*, such as *dim*, refer to other rules.

The *semantic* behavior of a language is often specified using syntax directed translation. Each symbol in a given grammar rule has a set of attributes associated with them along with an action. The action describes what to do whenever a particular grammar rule is recognized.

Yacc uses a parsing technique called lookahead-LR (LALR) parsing, which is based on the LR(0) sets of items, but has fewer states than typical parsers based on the LR(1) items [1]. It is a bottom up scheme

that tries to match a sequence of lexical objects against the right-hand-side of various grammar rules. Whenever a matching right-hand-side is found, the appropriate action code is triggered and the grammar symbols are replaced by the grammar symbol on the left-hand-side.

Implementing a parser in Python Yacc is fairly straight-forward. The list of tokens from the lexer module is imported and a series of functions describing the grammar productions are defined. From the grammar in Figure 5 the corresponding Python code becomes:

Listing 7: Parser example

```
def p_type(self, p):
    """
    type : FILE
         | FLOAT
         | INT
         | STRING
         | IDENT
         | type dim
    """
    if len(p) == 2:
        p[0] = csast.Type(p[1])
    else:
        p[1].type += p[2]
        p[0] = p[1]


def p_dim(self, p):
    """
    dim : LBRACKET RBRACKET
        | LBRACKET RBRACKET dim
    """
    if len(p) == 3:
        p[0] = '[]'
    else:
        p[0] = '[]' + p[3]
```

Each function has a doc string that contains the appropriate context-free grammar specification. This idea was actually borrowed from the SPARK toolkit [2]. A function takes an argument, *p*, that contains a sequence, starting at index 1, of values matching the symbols in the corresponding rule. The value `p[0]` is mapped to the left-hand-side rule, while the values in `p[1..]` are mapped to the grammar symbols on the right-hand-side. The statements in the function body implements the semantic actions of the rule. In this case, we use the parser to to build an abstract syntax tree. This is described in more detail in Section 4.2.1.

ALTERNATIVE SPECIFICATION OF LEXER AND PARSER As seen in the above examples, both the lexer and parser are defined from instances of their own classes. The easiest way, however, is to specify them directly in their own modules. The PLY documentation explains this quite well, complete with examples [4].

4.2 IMPLEMENTATION DETAILS

This section will describe the various implementation steps taken during the construction of Rheos.

4.2.1 *Abstract Syntax Tree*

The idea behind an abstract syntax tree (AST) is to represent the abstract syntactic structure of the source code in tree form. Each node in the tree represents some structure occurring in the source. The AST provides a good structure for later compiler stages since it omits details having to do with the source language, and only contains information about the essential structure of the program.

The AST is implemented using node classes for important language constructs. All these node classes extends an abstract base class. Since Python is dynamically typed, the concept of interfaces does not really exist. Interfaces, commonly referred to as "protocols", are implicit. Determining these interfaces is based on implementation introspection. The implementation of the abstract base is given in Listing 8 below [5].

Listing 8: An abstract base class for AST nodes

```
class Node(object):
    """ Abstract base class for AST nodes.
    """
    def children(self):
        """ A sequence of all children that are Nodes
        """
        pass

    def show(
        self,
        buf=sys.stdout,
        offset=0,
        attrnames=False,
        nodenames=False,
        showcoord=False,
        _my_node_name=None):
        lead = ' ' * offset
        if nodenames and _my_node_name is not None:
            buf.write(
                lead + self.__class__.__name__+ ' <' + _my_node_name
                    + '>: ')
        else:
            buf.write(lead + self.__class__.__name__+ ': ')

        if self.attr_names:
            if attrnames:
                nvlist = [(n, getattr(self,n)) for n in self.
                    attr_names]
                attrstr = ', '.join('%s=%s' % nv for nv in nvlist)
            else:
```

```
                vlist = [getattr(self, n) for n in self.attr_names]
                attrstr = ', '.join('%s' % v for v in vlist)
            buf.write(attrstr)

        if showcoord:
            buf.write(' (at %s)' % self.coord)
        buf.write('\n')

        for (child_name, child) in self.children():
            child.show(
                buf,
                offset=offset + 2,
                attrnames=attrnames,
                nodenames=nodenames,
                showcoord=showcoord,
                _my_node_name=child_name)
```

This base class also contains a pretty printing function, `show()`, that prints the entire tree below a the node from which it was invoked from.

An AST node can be specified in the following way:

Listing 9: Example of an AST node

```
class Header(Node):
    def __init__(self, ident, doc, inputs, outputs, coord=None):
        self.ident = ident
        self.doc = doc
        self.inputs = inputs
        self.outputs = outputs
        self.coord = coord

    def children(self):
        nodelist = []
        if self.ident is not None:
            nodelist.append(("ident", self.ident))
        if self.doc is not None:
            nodelist.append(("doc", self.doc))
        for i, child in enumerate(self.inputs or []):
            nodelist.append(("inputs[%d]" % i, child))
        for i, child in enumerate(self.outputs or []):
            nodelist.append(("outputs[%d]" % i, child))
        return tuple(nodelist)

    attr_names = ()
```

Python also does not support multiple dispatch at the language definition or syntactic level, nor does it support method overloading. However, the visitor pattern can be implemented using method introspection. Another base class for visiting nodes is defined:

Listing 10: The NodeVisitor class

```
class NodeVisitor(object):
    def visit(self, node):
        """ Visit a node.
        """
```

```
        method = 'visit_' + node.__class__.__name__
        visitor = getattr(self, method, self.generic_visit)
        return visitor(node)

    def generic_visit(self, node):
        """ Called if no explicit visitor function exists for a
            node. Implements preorder visiting of the node.
        """
        for c_name, c in node.children():
            self.visit(c)
```

Listing 11: An example use of the NodeVisitor

```
class ConstantVisitor(NodeVisitor):
    def __init__(self):
        self.values = []

    def visit_Constant(self, node):
        self.values.append(node.value)

...

cv = ConstantVisitor()
cv.visit(node)
```

4.2.2 *Type-checker*

Rheos has a quite interesting type system, which makes type-checking a non-trivial task. The type-checker class will take the resulting AST from the parser and use the visitor pattern to traverse the tree, taking appropriate actions at every node while building an environment.

TYPE-CHECKER When type-checking Rheos, most of the steps are straight-forward, but there are some cases where it becomes very complicated. Primitive types and literals already contain their type information from the parser stage. Type-checking of new types is a question of checking their elements and adding the definition to the environment, in order to make them recognizable to the rest of the program.

Parametrized components typically can not be gradually instantiated, but this is only partially true for Rheos, since components can be instantiated without any inputs and have them connected afterwards. On the other hand, components that require meta-arguments must have all of them supplied at instantiation.

Resolving meta types can only be done when a component requiring meta-type arguments is instantiated. A copy of the referred component is placed in the local context and given a new name. The meta arguments are type-checked to make sure they are of the same meta type as the required arguments. If they are of the wrong type, or the number of arguments given does not match the number of arguments

required, the type-checker raises an exception. If all these checks are passed, the type-checker continues with retrieving the type of the argument, and makes a variable substitution on the types of the instantiated component. When this is done, type-checking of the substituted type expression is resumed as if it were in the middle of checking an ordinary type expression.

ENVIRONMENT The current environment is implemented to mimic the structure of the actual Rheos code. Components, new types and instantiated components are all stored in a record of *{key : value}*-pairs. The keys are the names of the entry, while the value is a representation of their types. Types are generalized to a couple of different objects; `Component, Newtype, Generictype` and `Type`. These objects are instantiated and added to the environment by the type-checker in a way that makes it possible to reference their elements using ordinary object operations, kind of like how it is done in the Rheos language. For example, to fetch the type `t` of input-parameter `a` from component `comp`, stored in the environment, one would write something like:

Listing 12: Look-up of types in the environment

```
# fetch the type of comp.in.a
t0 = env['comp'].inp.a

# fetch the type of an element from a new type
# type setting (
#   int[] a ,
#   float b
#   )
t1 = env['setting'].a
```

4.2.3 *XML generation*

There is an implemented XML generator for an earlier version of Rheos. Due to signifacant changes of the language description, other aspects were prioritized and XML generation was left for the developers of Copernicus to update. The new language description that emerged was in fact so different from the original, that Copernicus needed updates to incorporate those changes.

The XML generator is implemented using the same visitor pattern as the type-checker. Visiting the different nodes in the abstract syntax tree produces corresponding XML code, and traversing the while tree will yield an entire definition, complete with indentations.

4.2.4 *Emacs mode*

The Emacs mode provides nothing more than syntax highlighting. The mode was created mainly to provide a more appealing look to the example code developed during design and testing of the Rheos. The syntax highlighting adds some understanding of what the code actually represents, which made it easier to add and change specific parts of the DSL.

4.3 FUTURE WORK

4.3.1 *Lexer and Parser*

As described in Section 3.10.2, there is a need for a `transpose` primitive. This would have to be specified as a special keyword in the lexer and also be addressed separately in the parser.

4.3.2 *Type-checker*

What remains to be done and future work implementation-wise has a lot to do with the type-checker. Since it was decided to add polymorphic stage in the project, the type-checker had to be completely parametrized typing to allow for generic components at a very late re-written. This change proved to be very time-consuming.

The environment was re-written at the same time to make it more powerful and intuitive to work with. Before the current implementation, the environment consisted of a lot of different records and lists, and did not perform well on look-ups.

Part III

RESULTS

5

CONCLUSIONS

5.1 THE PROBLEM AT HAND

As the main objective was to find a solution to Copernicus problem of describing its projects, Rheos is an answer to this projects main goal. It is possible to describe computational project for Copernicus and Rheos supports all current aspects of the Copernicus system.

As required Rheos is a text-based way of describing project networks in the form of a descriptive domain-specific language. Rheos supports arbitrary plug-ins, since you can describe how and what Copernicus should execute. As long as Copernicus as access to that executable any kind of plug-in is possible to use in Rheos.

Rheos is simple because of its limited capabilities. The point of the DSL is to describe the project and not to execute or evaluate any code, but it is still a powerful tool for the users.

5.2 RHEOS VERSUS XML

To demonstrate the simplicity and descriptive difference of Rheos compared to XML, the following Rheos code and the XML code in Appendix B both describes the project in Figure 4.

```
import gromacs

type mdrun_output '''Set of mdrun outputs'''
  ( file : conf
  , file : xtc
  , file : trr
  , file : edr )

network project_network '''Project network'''
  in ( file[] conf      '''The simulation parameters'''
     , file[] mdp
     , file[] top )
  out ( mdrun_output[] results )
  { c = mdrun(grompp(conf, mdp, top).out.tpr)
    out.results <- c.out.results }
```

5.3 STATE OF THE IMPLEMENTATION

While the Rheos never got to a state where it could be integrated with Copernicus, and still has some implementation steps that needs to be addressed, the project as a whole got very far. Over the course of the last months, Rheos has evolved from just being a replacement for the XML-description of Copernicus' input to a much more powerful language. Much time was spent on design and re-design of the language. New features was continuously added to the language which made it hard to for the project to reach the point where Rheos was integrated with Copernicus. It is important to have a powerful tool so that Rheos is useful to the users, and that is why adding features was prioritized. The integration with Copernicus was left to its developers.

5.4 WHAT RHEOS BECAME

5.4.1 *A New Approach*

Rheos is a completely new solution for input to systems like Copernicus. The language is designed specifically for Copernicus and its users. The idea is for users to fast and easily understand and use Rheos as a minor step when running computations on a Copernicus system. This is not the purpose for other distributed processing systems. The need for a solution like Rheos has probably not been encountered before Copernicus was developed, because of reasons like this.

The intention behind software like Hadoop, is to supply a platform on which developers can create new applications without having to worry about coordination of resources, since it is taken care of by the framework. This is not the desired approach for Rheos and Copernicus.

5.4.2 *Programing style*

Using Rheos for describing computational projects is a sort of metaprogramming. Rheos describes and manipulates how a Copernicus system should evaluate a project. It is not the intention for users to think of it this way, but to just to supply a problem and get an answer. Rheos meet this intention since the users describes workflows and send them to a Copernicus system.

Rheos has a flow-based nature, which is necessary to correctly describe projects in Copernicus. There are also some functional aspects to the language, which makes describing projects easier and provides a good overview of networks as the code will be dense. The user can choose to describe its project as both a workflow and with some functional ways of initializing components in the network.

5.4.3 *Powerful Description Language*

The DSL is stronger than just a set of macros. With Rheos it is possible to express generic parallel algorithms, which would be too advanced, or even impossible, with ordinary macros for just adding components and connections. Pre-defined code would be easier for users to understand and use with the definition and the type-system available. As the code describes networks it should be easier to understand what the network actually look like.

The strong type-system makes Rheos a much safer way of building networks compared to the old input system where computations would fail on runtime when Copernicus actually would build the project networks. The polymorphism makes the type-system powerful for a description language. It makes using pre-defined code much easier for users, where they can for example map functions with any kind of inputs.

With Rheos, Copernicus can now be used by non-developer users and they can still use it as a powerful tool. This is very important since many of the intended users are included in this category, and was the inspiration for the entire project. It is still possible to use Rheos in a complicated way, which developers do. Since Rheos supports arbitrary executables, developers can easily add new modules to the language. In the end Rheos has met the needs of Copernicus and made it easy for users to describe complex or big computations in a simple way.

Part IV

APPENDIX

A

A GRAMMAR OF RHEOS

This part describes the complete grammar of Rheos in BNF.

GRAMMAR PRODUCTIONS

⟨*entrypoint*⟩ ::= ⟨*opt_cr*⟩ ⟨*program*⟩
| ⟨*empty*⟩

⟨*program*⟩ ::= ⟨*top_def_list*⟩ ⟨*opt_cr*⟩

⟨*top_def_list*⟩ ::= ⟨*top_def*⟩
| ⟨*top_def_list*⟩ ⟨*cr*⟩ ⟨*top_def*⟩

⟨*top_def*⟩ ::= ⟨*import_Stmt*⟩
| ⟨*newtype_decl*⟩
| ⟨*atom_decl*⟩
| ⟨*network_decl*⟩

IMPORT STATEMENT

⟨*import_stmt*⟩ ::= `import` ⟨*package_path*⟩

⟨*package_path*⟩ ::= ⟨*package_identifier*⟩
| ⟨*package_path*⟩ `.` ⟨*package_identifier*⟩

⟨*package_identifier*⟩ ::= ⟨*ident*⟩

NEWTYPE

⟨*newtype_decl*⟩ ::= `type` ⟨*ident*⟩ ⟨*docstring*⟩ ⟨*cr*⟩ ⟨*lparen*⟩ ⟨*type_decl_list*⟩
⟨*rparen*⟩

⟨*type_decl_list*⟩ ::= ⟨*type_decl*⟩
| ⟨*type_decl_list*⟩ ⟨*comma_sep*⟩ ⟨*type_decl*⟩

⟨*type_decl*⟩ ::= ⟨*type*⟩ `:` ⟨*ident*⟩

ATOM DECLARATION

⟨*atom_declaration*⟩ ::= `atom` ⟨*atomtype*⟩ ⟨*header*⟩ ⟨*optionblock*⟩

⟨*atomtype*⟩ ::= `python`
| `python-extended`
| `external`

⟨*optionblock*⟩ ::= `option` ⟨*opt_cr*⟩ ⟨*lparen*⟩ ⟨*atom_option_list*⟩ ⟨*rparen*⟩

⟨*atom_option_list*⟩ ::= ⟨*atom_option*⟩
| ⟨*atom_option_list*⟩ ⟨*comma_sep*⟩ ⟨*atom_option*⟩

⟨*atom_option*⟩ ::= ⟨*option_ident*⟩ `:` ⟨*option_ident*⟩

⟨*option_ident*⟩ ::= ⟨*ident*⟩
| ⟨*option_ident*⟩ `.` ⟨*ident*⟩

NETWORK DECLARATION

⟨*network_decl*⟩ ::= `network` ⟨*header*⟩ ⟨*network_block*⟩

⟨*network_block*⟩ ::= ⟨*statement_block*⟩

ATOM AND NETWORK HEADERS

⟨*header*⟩ ::= ⟨*ident*⟩ ⟨*metaparams*⟩ ⟨*docstring*⟩ ⟨*cr*⟩ ⟨*inputs*⟩ ⟨*outputs*⟩

⟨*metaparams*⟩ ::= `<` ⟨*metaparam_list*⟩ `>`
| ⟨*empty*⟩

⟨*metapara_list*⟩ ::= ⟨*metaparam*⟩
| ⟨*metaparam_list*⟩ ⟨*comma_sep*⟩ ⟨*metaparam*⟩

⟨*metaparam*⟩ ::= `type` ⟨*ident*⟩
| `func` ⟨*ident*⟩
| `list` ⟨*ident*⟩

⟨*inputs*⟩ ::= `in` ⟨*lparen*⟩ ⟨*input_list*⟩ ⟨*rparen*⟩ ⟨*cr*⟩
| `in` ⟨*no_params*⟩ ⟨*cr*⟩

⟨*outputs*⟩ ::= `out` ⟨*lparen*⟩ ⟨*output_list*⟩ ⟨*rparen*⟩ ⟨*cr*⟩
| `out` ⟨*no_params*⟩ ⟨*cr*⟩

⟨*input_list*⟩ ::= ⟨*input*⟩
| ⟨*input_list*⟩ ⟨*comma_sep*⟩ ⟨*input*⟩

⟨*output_list*⟩ ::= ⟨*output*⟩
| ⟨*output_list*⟩ ⟨*comma_sep*⟩ ⟨*output*⟩

⟨*no_params*⟩ ::= `(` ⟨*opt_cr*⟩ `)`

⟨*input*⟩ ::= ⟨*type*⟩ ⟨*ident*⟩ ⟨*docstring*⟩
| `?` ⟨*type*⟩ ⟨*ident*⟩ ⟨*docstring*⟩
| ⟨*type*⟩ ⟨*ident*⟩ `default` ⟨*constant*⟩ ⟨*docstring*⟩

⟨*output*⟩ ::= ⟨*type*⟩ ⟨*ident*⟩ ⟨*docstring*⟩
| ⟨*type*⟩ ⟨*ident*⟩ `default` ⟨*constant*⟩ ⟨*docstring*⟩

STATEMENTS

⟨*stmt_block*⟩ ::= ⟨*opt_cr*⟩ ⟨*lbrace*⟩ ⟨*stmt_list*⟩ ⟨*rbrace*⟩
| ⟨*opt_cr*⟩ ⟨*stmt_block_empty*⟩

⟨*stmt_block_empty*⟩ ::= `{` ⟨*opt_cr*⟩ `}`

⟨*stmt_list*⟩ ::= ⟨*stmt*⟩
| ⟨*stmt_list*⟩ ⟨*cr*⟩ ⟨*stmt*⟩

⟨*stmt*⟩ ::= ⟨*controller_stmt*⟩
| ⟨*connection_stmt*⟩
| ⟨*assignment_stmt*⟩

⟨*controller_stmt*⟩ ::= `controller (` ⟨*ident*⟩ `)`

⟨*connection_stmt*⟩ ::= ⟨*param_ref*⟩ `<-` ⟨*expr*⟩

⟨*assignment_stmt*⟩ ::= ⟨*ident*⟩ `=` ⟨*component_stmt*⟩

⟨*component_stmt*⟩ ::= ⟨*ident*⟩ ⟨*component_args*⟩
| ⟨*ident*⟩ ⟨*metaargs*⟩ ⟨*component_args*⟩

⟨*metaargs*⟩ ::= `<` ⟨*type_list*⟩ `>`

⟨*type_list*⟩ ::= ⟨*type*⟩
| ⟨*type_list*⟩ ⟨*comma_sep*⟩ ⟨*type*⟩

⟨*component_args*⟩ ::= ⟨*lparen*⟩ ⟨*expr_list*⟩ ⟨*rparen*⟩
| ⟨*lparen*⟩ ⟨*rparen*⟩

EXPRESSIONS

⟨*expr*⟩ ::= ⟨*constant*⟩
| ⟨*expr_ref*⟩

⟨*expr_list*⟩ ::= ⟨*expr*⟩
| ⟨*expr_list*⟩ ⟨*comma_sep*⟩ ⟨*expr*⟩

⟨*constant*⟩ ::= ⟨*fconst*⟩
| ⟨*iconst*⟩
| ⟨*sconst*⟩

⟨*fconst*⟩ ::= *floating point constant*

⟨*iconst*⟩ ::= *integer constant*

⟨*sconst*⟩ ::= *string constant*

⟨*ident*⟩ ::= *identifier*

⟨*expr_ref*⟩ ::= ⟨*expr_ref_list*⟩

⟨*expr_ref_list*⟩ ::= ⟨*expr_ref_1*⟩
| ⟨*expr_ref_list*⟩ ⟨*recordindex*⟩
| ⟨*expr_ref_list*⟩ ⟨*arrayindex*⟩
| ⟨*expr_ref_list*⟩ `.` ⟨*expr_ref_1*⟩

⟨*recordindex*⟩ ::= `(` *integer constant* `)`

⟨*arrayindex*⟩ ::= `[` *integer constant* `]`

⟨*expr_ref_1*⟩ ::= ⟨*expr_ref_2*⟩
| ⟨*lparen*⟩ ⟨*component_stmt*⟩ ⟨*rparen*⟩

⟨*expr_ref_2*⟩ ::= `in`
| `out`
| ⟨*ident*⟩

TYPES AND DOCSTRINGS

⟨*type*⟩ ::= `file`
| `float`
| `int`
| `string`
| ⟨*metatype*⟩
| ⟨*type*⟩ `[ ]`

⟨*metatype*⟩ ::= ⟨*expr_ref_2*⟩
| ⟨*metatype*⟩ ⟨*recordindex*⟩
| ⟨*metatype*⟩ `*`
| ⟨*metatype*⟩ `.` ⟨*expr_ref_2*⟩

⟨*docstring*⟩ ::= *an optional docstring*

SPECIAL PRODUCTIONS

⟨*cr*⟩ ::= *a non-empty sequence of new lines*

⟨*opt_cr*⟩ ::= *a sequence of new lines that may be empty*

⟨*lbrace*⟩ ::= `{` ⟨*opt_cr*⟩

⟨*rbrace*⟩ ::= ⟨*opt_cr*⟩ `}`

⟨*lparen*⟩ ::= `(` ⟨*opt_cr*⟩

⟨*rparen*⟩ ::= ⟨*opt_cr*⟩ `)`

⟨*empty*⟩ ::= *empty production*

⟨*comma_sep*⟩ ::= ⟨*opt_cr*⟩ `,` ⟨*opt_cr*⟩

B

EXAMPLE CODE

This is the XML code describing the project network from Figure 4.

```
<?xml version="1.0"?>
<cpc>
    <import name="gromacs" />

    <type id="conf_array" base="array" member-type="file">
        <desc>An array of conf files</desc>
    </type>

    <type id="mdp_array" base="array" member-type="file">
        <desc>An array of conf files</desc>
    </type>

    <type id="top_array" base="array" member-type="file">
        <desc>An array of conf files</desc>
    </type>

    <type id="mdrun_output" base="record">
        <desc>Set of mdrun outputs</desc>

        <field type="file" id="conf" />
        <field type="file" id="xtc" />
        <field type="file" id="trr" />
        <field type="file" id="edr" />
    </type>

    <type id="mdrun_output_array" base="array" member-type="file">
        <desc>An array of conf files</desc>
    </type>

    <function id="project_network" type="network">
        <desc>Project network</desc>
        <inputs>
            <field type="conf_array" id="conf">
                <desc>The simulation parameters</desc>
            </field>
```

```
            <field type="mdp_array" id="mdp" />
            <field type="top_array" id="top" />
        </inputs>
        <outputs>
            <field type="mdrun_output_array" id="results" />
        </outputs>
        <network>
            <instance id="grompp" function="gromacs::grompp"/>
            <instance id="mdrun" function="gromacs::mdrun"/>

            <connection src="self:ext_in.conf"
                        dest="grompp:in.conf" />
            <connection src="self:ext_in.mdp"
                        dest="grompp:in.mdp" />
            <connection src="self:ext_in.top"
                        dest="grompp:in.top" />

            <connection src="grompp:out.tpr"
                        dest="mdrun:in.tpr" />

            <connection src="mdrun:out.results"
                        dest="self:ext_in.results" />
        </network>
    </function>

</cpc>
```

BIBLIOGRAPHY

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques & Tools*. Pearson/Addison Wesley, 2nd edition, 2007.

[2] John Aycock. SPARK. [online], May 2012. URL http://pages.cpsc.ucalgary.ca/~aycock/spark/.

[3] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–614, August 1978.

[4] David Beazley. PLY (python lex-yacc). [online], April 2012. URL http://www.dabeaz.com/ply/.

[5] Eli Bendersky. pycparser. [online], April 2012. URL http://code.google.com/p/pycparser/.

[6] Björn Bringert, Johan Broberg, Paul Callaghan, Markus Forsberg, Ola Frid, Peter Gammie, Patrik Jansson, Kristofer Johannisson, Antti-Juhani Kaijanaho, Ulf Norell, Michael Pellauer, and Aarne Ranta. The bnf converter. [online], May 2012. URL http://bnfc.digitalgrammars.com/.

[7] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23:187–200, 2001.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1): 107–113, January 2008.

[9] I. Foster, Yong Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop*, GCE '08, November 2008.

[10] Apache Software Foundation. Hadoop. [online], May 2012. URL http://hadoop.apache.org/.

[11] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in Dataflow Programming Languages. *ACM Computing Surveys*, 36(1):1–34, March 2004.

[12] Donald E. Knuth. Computer programming as an art. *Communications of the ACM*, 17(12):667–673, December 1974.

[13] Rainer Leupers, Lieven Eeckhout, Grant Martin, Frank Schirrmeister, and Nigel and Topham. Virtal Manycore Platforms: Moving Towards 100+ Processor Cores. In *Design, Automation Test in Europe Conefrence Exhibition*, pages 1–6, March 2011.

[14] Kevin Modzelewski, Jason Miller, Adam Belay, Nathan Beckmann, III Gruenwald, Charles, David Wentzlaf, Lamia Youseff, and Anant Agarwal. A Unified Operating System for Clouds and Manycore: fos. Technical Report MIT-CSAIL-TR-2009-059, CSAIL, November 2009. URL http://hdl.handle.net/1721.1/49844.

[15] J. Paul Morrison. *Flow-Based Programing: A New Approach to Application Development*. Createspace, 2nd edition, 2010.

[16] J. Paul Morrison. Flow-based programming. [online], April 2012. URL http://www.jpaulmorrison.com/fbp/.

[17] Amit Patel. YAPPS (Yet Another Python Parser System). [online], May 2012. URL http://theory.stanford.edu/~amitp/yapps/.

[18] Sander Pronk, Per Larsson, Iman Pouya, Gregory R. Bowman, Imran S. Haque, Kyle Beauchamp, Berk Hess, Vijay S. Pande, Peter M. Kasson, and Erik Lindahl. Copernicus: A new paradigm for parallel adaptive molecular dynamics. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2011.

[19] Zhanyong Wan and Paul Hudak. Functional Reactive Programming from first principles. In *Proc. ACM SIGPLAN'oo Conference on Programming Language Design and Implementation (PLDI'oo)*, 2000.