Proceedings of the International Multiconference on Computer Science and Information Technology pp. 663–670
ISBN 978-83-60810-27-9
ISSN 1896-7094

A Modeling Language Approach for the Abstraction of the Berkeley Open Infrastructure for Network Computing (BOINC) Framework

Christian Benjamin Ries
University of Applied Sciences Bielefeld
Computational Materials Science & Engineering
Wilhelm-Bertelsmann-Str. 10,
33602 Bielefeld, Germany
Christian_Benjamin.Ries@fh-bielefeld.de

Thomas Hilbig, Christian Schrñder
University of Applied Sciences Bielefeld
Computational Materials Science & Engineering
Wilhelm-Bertelsmann-Str. 10,
33602 Bielefeld, Germany
{Thomas.Hilbig, Christian.Schroeder}@fh-bielefeld.de

***Abstract*—BOINC (*Berkley Open Infrastructure for Network Computing*) is a framework for solving large scale and complex computational problems by means of public resource computing. Here, the computational effort is distributed onto a large number of computers connected by the Internet. Each computer works on its own workunits independently from each other and sends back its result to a project server. There are quite a few BOINC-based projects in the world. Installing, configuring, and maintaining a BOINC based project however is a highly sophisticated task. Scientists and developers need a lot of experience regarding the underlying communication and operating system technologies, even if only a handful of BOINC related functions are actually needed for most applications. This limits the application of BOINC in scientific computing although there is an ever growing need for computational power in this field. In this paper we present a new approach for *model-based development* of BOINC projects based on the specification of a high level abstraction language as well as a suitable development environment. This approach borrows standardized modeling concepts from the well-known *Unified Modeling Language (UML)* and *Object Constraint Language (OCL)*.**

I. Introduction

VOLUNTEER computing technologies allow to realize low-cost high-performance computing projects in certain application areas. A very prominent framework based on the principle of Public-Resource Computing (PRC) is BOINC (*Berkley Open Infrastructure for Network Computing*). BOINC provides an Application Programming Interface (API) with about one hundred functions of different categories, e.g. *filesystem operations*, *process controlling* and *status message handling* [1], [2]. A few of the most important functions are listed in section I-B. PRC is based on a server-client communication infrastructure mechanism. Here, the client retrieves a project specific application from the server along with a so-called workunit, i.e. a number of parameters usually provided in data files of simple ASCII or binary format that are optionally needed by the application to perform a specific task. BOINC is strongly focused on autonomic applications, each BOINC project has its own server, applications and tasks. The client executes the application, i.e. performs the calculations and sends its results back to the server which assembles these into a "global" result or stores these results at specific places.

This project is funded by the German Federal Ministry of Education and Research

The setup of a BOINC project heavily relies on the use of file-based scripting techniques. For instance, the programming language *Python* is utilized for the creation of a standard BOINC server infrastructure with database initialization, website and administration interface configuration and an optional BOINC test application. A few scripts are implemented using GNU BASH to sign executable files with encryption keys and yet another script uses the C shell (csh) to monitor network traffic. Extensible Markup Language (XML) files are used for the runtime server configuration. *All* files have to be edited *manually* by the project developer, scientist or administrator which bears the risk of making a large number of typical errors. For example, wrong spelling of parameter names or simulation relevant values as shown in I-A would have a significant effect on the system's integrity and the application's performance [8]. One way to cope with these problems is to provide a tool support which allows to automate the manipulation, generation, and checking of all necessary scripts.

In this paper we discuss a model-based approach for the development of BOINC projects that makes it possible to implement application specific changes while always keeping a valid configuration of the BOINC infrastructure. We give an idea on how to create a proper high-level domain specific language (DSL) in order to develop and maintain a complete BOINC application. This DSL forms the basis of an easy-to-use programming environment along with a high-level programming language and a suitable development process. Additionally, we present a way to model a complete BOINC installation including the client application for different target computer architectures and processor types. Aspects of single- and multicore processor units (CPU) and graphics processing units (GPU) are also discussed.

A. Typical errors during the BOINC server configuration process

The following list shows examples of typcial errors that can occur due to manual editing of the BOINC server configuration files. As a consequence of these errors one can expect a significant effect on the system's integrity and application performance.

- *uldl_dir_fanout* is the parameter that contains the number of subdirectories inside the upload and download directories on the server computer. A wrong value set here may dramatically slow down the system's server performance because of too many hard disk drive accesses.
- *shmem_key* names the allocated memory that is needed for the interprocess communication (IPC) between all BOINC applications on every BOINC project server. It is required that this value is unique, never changed during the runtime and is used by all BOINC server applications.
- *msg_to_host* must be included in the BOINC server configuration to enable sending of trickle-down messages[1] to the BOINC client nodes.
- *tasks* describes a set of parameters for applications which should execute in a cycle period, i.e. a crontab. Suitable values are needed to avoid problems like extremely large logging files or a outdated statistics, i.e. how many workunits are left for working or have errors during the computation.
- *daemon* contains a set of command descriptions. It is useful to start more than one daemon process to get a good load balancing of user requests, e.g. when the BOINC projects are much in demand.

One of our goals is the automatic determination of these most important parameters for different target computer architectures and processor types [20].

B. The BOINC Application Programming Interface

BOINC offers few example applications in which the number of lines of code range from 38 to 308. The first one only includes some elementary functions and no BOINC specific commands, e.g. a for-loop which just keeps the processor busy for one second. The second one is a more useful example since it contains BOINC specific function calls, e.g. how to retrieve the name of the checkpoint file or the actual processing state. Implementing a complex scientific application using BOINC is far more complicated and requires a broad experience of the developer. However, one can show that only 23 different BOINC functions are necessary to create a successfully running research relevant distributed computing application [23], [21].

II. State of the Art

Model-driven engineering (MDE) is becoming the dominant software engineering paradigm to specify, develop and maintain software systems. For example, Brunelière *et al.* [6] propose a *Modeling as a Service* (MaaS) initiative for cloud computing projects with an emphasis on topics like scalability, tool interoperability, and the definition of modeling mash-ups as a combination of MDE services from different vendors. Moreover, the definition of domain-specific languages (DSL) [10], [26], [27] along with the development of tools which support the developer during the DSL conception [9], [17], [26] are currently under investigation. However, none of the above mentioned approaches is related to public resource computing nor can it be directly used for the modeling process of BOINC projects. In the following chapters we therefore propose a first modeling language approach for the abstraction of the BOINC framework.

[1]Trickle messages are asynchronous, ordered, and reliable messages between the BOINC server/clients and let applications communicate with the server during the execution of a workunit.

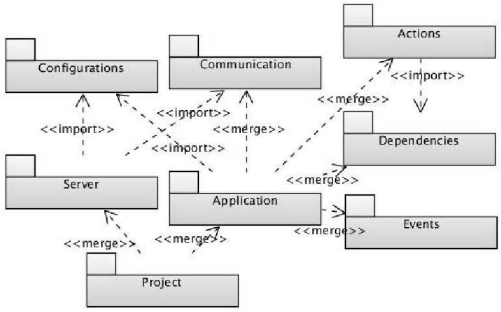

Fig. 1. Logical packages and dependencies of the BOINC functionalities

III. Abstraction of the BOINC Framework

The BOINC framework offers many very useful functionalities that help the developer to create his application. As a first step towards our modeling language approach we have subdivided the BOINC functionalities into different logical packages as shown in Fig. 1. Each package contains functions that cover a specific aspect during the development process [22] and can be used independently from functions of other packages which minimizes the number of dependencies. The whole BOINC project including the server installation components and application specific implementations is contained in the package *Project*. This package directly depends on the packages *Server* and *Application*. The package *Application* contains all application specific implementations and depends on the following child packages:

- *Events* - This package describes the abstractions for all possible events that can occur during the execution, e.g. exceptions like *'File not found'* or *'Segmentation fault'*. It contains routines for clearly defined error handling.
- *Actions* - Every execution statement is gathered within this package including wrapper routines to call third-party applications, i.e Matlab, or other domain-specific simulation tools.
- *Dependencies* - All libraries that are needed for the whole project are contained in this package. This includes the BOINC libraries as well as application specific runtime libraries and external sources.

- *Communication* - This package includes the BOINC core client component which is the interface between the client installations and project servers and performs the information exchange between them.
- *Configurations* - In this package one keeps all system relevant parameters coded in XML files.

The complete server installation and maintenance process is provided by the package *Server*. This package describes the relationships between all components of a complete BOINC server installation, i.e. all configuration files, a list of the parameters for the workunits, description of installed applications with the corresponding architecture and processor targets. The package *Server* imports its required information from the contents of the *Configurations* and *Communication* packages.

IV. The Modeling Language Approach for the Abstraction of BOINC

Nowadays, models and model-based techniques are the fundamental means by which engineers are able to cope with otherwise unmanageable complexity and reduce design risk. In particular, software models have the distinct advantage that they can be *evolved* from high-level views of possible designs into actual implementations.

The *Unified Modeling Language (UML)* – a widely adopted, widely supported and customizable industry standard – plays a key role in modern software development. With the possibility of creating standardized UML profiles it provides the fundamentals for a true engineering-oriented approach to the construction of software. That is, system models can be used to understand and assess designs and predict design risks in meaningful (e.g., quantifiable) ways. Full automatic code generation from UML models facilitates preservation of proven model properties in the final implementation.

In our approach to a model-based development of BOINC projects we focus on the use of quasi standard software tools available within the Eclipse development environment. These tools enable us to develop all necessary components, like diagram editors, including graphical representations of modeling elements, code generators, etc. in one and the same development environment. Specifically, the code generation should be realized using a template engine which could also be used to generate important documentation files.

A. Graphical modeling environment

Throughout our project we exclusively use the Eclipse Modeling Framework (EMF) as released by the Eclipse Model Development Tools (MDT) project [26]. A detailed description of all components can be found in [9]. A key feature of our approach is the definition of a suitable standardized UML2 profile [22]. Here, we make use of the EMF-based implementation of the *UML2* and *UML2 Tools* subproject. EMF specifically allows implementing of constraints based on the Object Constraint Language (OCL) [18].

Fig. 2 gives an overview about a simplified modeling process. Here, the developer uses graphical modeling elements within diagrams to design an application as described in Sec. IV-B below. For a more detailed modeling of the application logic the developer is required to use a textual DSL as described in Sec. IV-D. The *Graphical Model* will be implemented using the Graphical Modeling Framework (GMF). GMF includes EMF, the Graphical Editing Framework (GEF) and Draw2D. GEF is a framework built upon the Model-View-Controller (MVC) pattern and handles the view and logical components with own instances. The controller handles the logic between these instances. The *Textual Model* and the *Graphical Model* are synchronized, i.e. every change in one of the models causes changes in the other one and each model will automatically be adjusted. Both models will be transformed into one (yet to be defined) so-called *VisualGridML Genmodel*. This can be done by exploiting the model-to-model transformation framework Query/View/Transformation (QVT). The QVT operation mapping language is capable of dealing with multiple input and output models and also supports OCL statements.

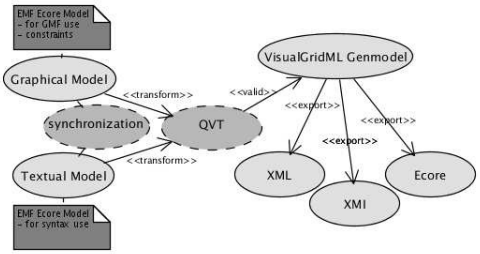

Fig. 2. Simplified view of the modeling process

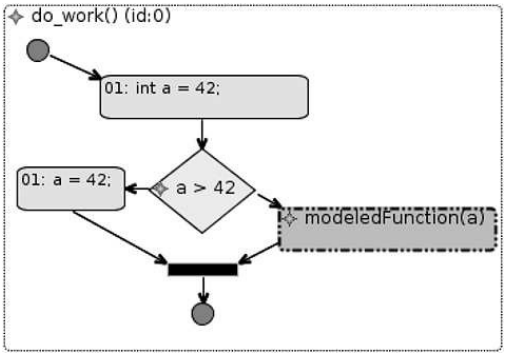

Fig. 3. Example flowchart diagram for instruction sequences.

One of the key issues of our approach is to define a proper *VisualGridML Genmodel*. The *VisualGridML Genmodel* can be exported into different other formats, e.g. XML, XML Metadata Interchange (XMI), or Ecore. The *VisualGridML Genmodel* will only be generated when the previously created models are valid. In order to support the developer during the verification and validation process adequate error messages and output comments will be created.

B. Graphical modeling elements

The proper definition of suitable graphical modeling elements is vital for our modeling language approach for the

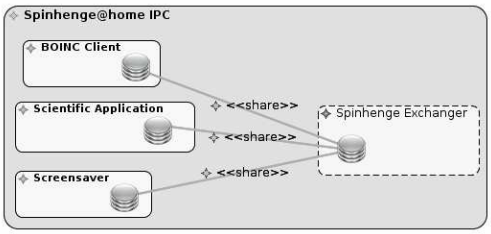

Fig. 4. Interprocess Communication within a BOINC application

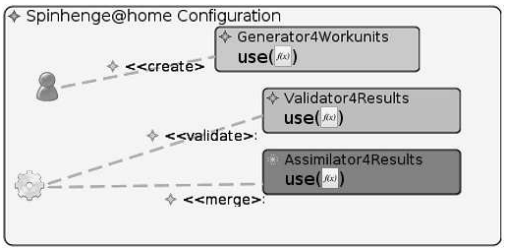

Fig. 5. Elements for configuration purpose

abstraction of BOINC. In the following we present some examples along with sample modeling fragments. Fig. 3 shows a flowchart of a high-level description of instruction sequences. The graphical notation is adapted from the UML2 flowchart definition [17, Fig. 12.36].

Fig. 3 shows an example that uses seven graphical modeling elements. As mentioned above these modeling elements are defined in the *Actions* package and have the following meaning [22]:

- *Start*, which describes the entry point of an application.
- *Stop*, which defines the end of execution, i.e. after that no other instructions will be executed and the application will shut down.
- *Action* is a modeling element which can execute native C/C++ instructions.
- *Decision* describes an `if-else` condition.
- *Join* merges two or more subdivided instruction sequences.
- *Execute* executes external C/C++ functions which are implemented in header and source files.

In Fig. 4 we show an adapted version of a UML2 collaboration diagram [17, Tab. 9.1, 9.2]. The dashed rounded rectangle on the right hand side specifies a shared data space. This data space is defined using one ore more port descriptors. Different ports could include different data descriptions and could also be connected with different so-called handlers. The three elements on the left hand side are examples of such handlers. Each handler handles a specific functionality and is connected with one application implementation. This example contains handlers for the *BOINC Core Client*[2], a scientific application, and a component for a screensaver session [1]. The handler can have only one data port. Data ports can only be connected to data ports of shared data space. Connections between ports may also be named as shown in this example.

In Fig. 5 we show an adapted version of a UML2 Use-Case diagram [17, Fig. 16.10] which contains elements of the *Configuration* package [22]. The elements on the left hand side are the actors for the use cases of the right hand side. The use cases contain a reference to predefined functions which can be modeled with a flowchart diagram. The dashed line connects the use cases with the appropriate actor and defines the type of execution. The lower actor describes the BOINC server which could be a cronjob. The upper actor describes a user. In this example the user defines workunits.

C. Aspect-Oriented Programming and Feature-Oriented Programming

Aspect-Oriented Programming (AOP) aims at separating and modularizing cross-cutting concerns [12]. The idea behind AOP is to implement so-called cross-cutting concerns as *aspects* and the core (non-cross-cutting) features are implemented as *components* [5]. Using *pointcuts*[3] and *advices*[4], an aspect weaver glues aspects and components at *join points* together. Fig. 6 shows on the left hand side (1) two aspects (A1 and A2) which extend the class definitions (C1 and C2). In AOP aspects can be added to the programming logic where ever functions are called. It is also possible to replace any functionality dynamically or to define the order of execution by precedence.

In Feature-Oriented Programming (FOP) the program functionalities can extended during the compilation and execution process, e.g. two or more functions can be combined to create extented features [7]. Fig. 6 shows on the right hand side (2) a simple overview of FOP. Here, F4 inherits the properties and methods of F1. Additionally, F6 refines F4, which can be done during runtime or while the compiling process creates the application.

Aspects and features in their current representation are intended for solving problems at different levels of abstraction [4], [14], [16]. Whereas aspects in AspectC++ [24] act on the level of classes and objects in order to modularize cross-cutting concerns, features act on the software architecture level [3].

For our approach, we are expecting that AOP and FOP are suitable methodologies for dynamic binding of applications as stated in Sec. IV-E. For example, the BOINC project result file format must be defined and created manually by the scientist or developer for a specific scientific application. This file can be generated during the code generation process as soon as definitions of the BOINC validator and BOINC assimilator are existing. After that, the BOINC project can be deployed with error-free validator and assimilator configurations.

D. Domain-specific language for BOINC

Domain-specific languages (DSL) are language definitions tailored to the development needs of specific problem domains

[2]The BOINC Core Client communicates with schedulers, uploads and downloads files, and executes and coordinates applications.

[3]The point of concern to execute an *advice*.

[4]Additional code that should apply to the existing model.

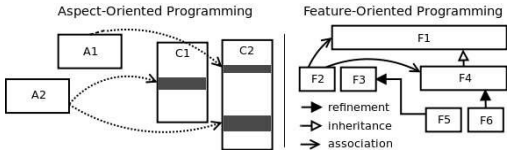

Fig. 6. (1) Two aspects extend two classes, (2) Two features refine two other features

[10]. For example the Structured Query Language (SQL) is designed for database queries. Another vital part of our modeling language approach to BOINC is a proper definition of a DSL for BOINC applications. Here, a key issue is the use of Xtext which allows to create an application by code generation. Xtext offers in combination with Xpand a template-based code generation engine [26]. By using Xtext and Xpand is has been shown that it is possible to create a complete BOINC application [21].

Generally speaking, all BOINC functionalities can be defined using a set of specific language elements. Our model-driven approach allows to develop applications independently of the target type, i.e. for CPU or GPU targets. Furthermore, BOINC offers various diagnostic parameters which enable or disable checks, e.g. *memory leaks* or *heap violations*. With the help of DSL statements these specific options can enabled or disabled for a subsequent automatic code generation. For example, the following code fragment defines a single processor environment which enables the above mentioned diagnostic flags:

```
target cpu single;

diagnostics {
 dumpcallstack
 heapcheck
 memoryleakcheck
 redirectstderr
 tracetostderr
}
```

In order to create applications with multicore or GPU computing support the following statement can be used:

```
target cpu mode multi with 10;
  // or
target gpu dim(10) block(4);
```

The first line describes a multi-thread application with up to 10 threads. The last statement enables the support of GPU computing. In GPU computing the process is splitted into *dim* threads, executed in 4 *blocks* [13]. It is also possible to include manual implementations or third-party libraries. The following examples show this in more detail. The required dependencies to third-party libraries or functionalities could be also defined with only a few lines of code. This code is used for the "make" process of an executable application. On Linux or Unix like operating systems a makefile is generated whereas on Windows systems it is possible to generate different project files which can be imported by integrated development environments like Visual Studio or Eclipse. Using this feature one can realize a *platform-independent* approach.

```
includes AppInclude {
 "~/boincadm/framework"
 "~/boincadm/src/api"
 "~/boincadm/src/lib"
}

libraries AppLibrary {
 "/lib", "pthread"
 "~/boincadm/framework", "visualgrid"
 "~/boincadm/src/api", "boinc_api"
 "~/boincadm/src/lib", "boinc"
}
```

It is common to use parameter files for the BOINC applications. The native way of doing this is to create *mapping files* on the server with a few parameters. Whenever clients connect to the server, they retrieve the application and all necessary parameter files. The following DSL fragment describes this procedure using the `infile` statement. The content is specified as an XML tree and could be iterated with a reference, e.g. `ObjectName1`. As a consequence of, binary data must be base64 encrypted [25]. Each reference contains the parameter as a *struct* or *class* definition.

```
infile "metropolis_data.xml"
                  as ObjectName1;
infile "param.jj" as ObjectName2;
infile "param.nn" as ObjectName3;
infile "param.ww" as ObjectName4;
```

After the execution of the client application the results are stored in *result files* defined by the statement `outfile` and are uploaded to the server.

```
outfile "metropolis_out.erg"
                  as ObjectResult1;
```

In general there exist several ways using a modeling process to develop a certain client application. As a matter of fact, every developer differs in his way to develop applications and it is necessary that the DSL supports this variety. For example, it is possible to link third-party libraries to the application using DSL statements. Furthermore, the application code can also be directly implemented into the `worker` part. The following DSL fragment contains an example which is used in [21]. Here, the `worker` starts the environment for execution instructions with the name `Spinhenge`. The statement `exec` defines pointcut expressions which are used by the AOP weaver process to deploy the scientific application. In this definition, the pointcut expression describes the working function in Fig. 8.

```
worker Spinhenge {
 exec "void %::Spinhenge::doWork(...)";
}
```

This statement could be replaced by other instructions, e.g.

```
worker Spinhenge {
 cpp {
  int a = 42;
 }
 action(modeledFunction(a));
}
```

Here, `cpp` starts an inline code area which contains native C/C++ statements. The variable `a` is available right after its definition and optional initialization within the context. It can be used by DSL defined functions like `modeledFunction(variable)`. Third-party applications can be executed using the DSL statement `wrapper`. It allows to call an external application, so called *legacy application*, and only one call is allowed to realize. Optional parameters for the application can be set in the *Configuration* package.

```
worker Spinhenge {
  wrapper("Matlab", "Argv[1] Argv[2]" [,
       weight, checkpoint_filename,
       fraction_done_filename, ...]);
}
```

These optional parameters are defined by the wrapper interfaces [11], [15], [19].

```
screensaver Spinhenge {
  render "% Screensaver
             ::Spinhenge::doWork(...)";
}
```

During the execution of an application different events can occur, e.g. events which could also be logged for diagnostic analysis in the above mentioned definitions. Furthermore an exception handling is described by the following DSL definition:

```
handle TypeOfException (: optionalName) {
 /* to be defined handler */
}
```

Predefined exception handlers are reusable by other exceptions. This is enabled by using the `ref` statement which uses the optional name `optionalName` of the previous example.

```
handle AnotherTypeOfException
   ref optionalName;
```

The BOINC framework uses interprocess communication (IPC) to exchange data between different application instances, e.g. scientific application, and screensaver. The native way to use IPC needs the definition of shared variables, e.g. in a C/C++ `struct` or `class` definition and furthermore different functions which handles these variables. A strict well-formed definition would be easier to use and reduces the effort of changing code parts in the source files. The DSL statement `exchange` describes the structure of the IPC with C/C++ language elements like datatypes which are usable in every application. The keyword `feature` defines an AOP pointcut expression which is used to assign values to the variable on the left hand side, e.q. `update_time` keeps the delta in milliseconds between the value updates. The listed AOP pointcut expressions are implemented in the *Visual Grid Framework* which is described in Sec. IV-E.

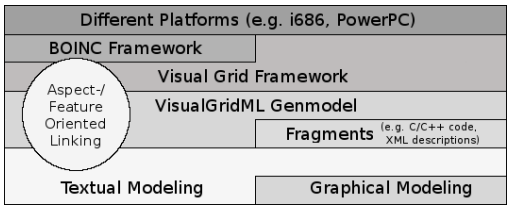

Fig. 7. *Visual Grid Framework* Abstraction Layer

```
exchange {
 double update_time :
   feature "% Boinc::updateTime(...)";
 double fraction_done :
   feature "% Boinc::fractionDone(...)";
 double cpu_time :
   feature "% Boinc::cpuTime(...)";
}
```

To handle BOINC Trickle Messages the `TrickleUp` and `TrickleDown` commands are stated in the following listing. The `TrickleUp` needs two AOP pointcut expressions to check if a trickle up must handled and the trickle up handler itself.

```
 TrickleUp "bool checkTrickleUp(...)"
     do "% handleTrickleUp(...)";
 TrickleDown "% handleTrickleDown(...)";
```

The handler defined by `TrickleDown` is called frequently to manage the incoming messages by the BOINC server, e.g. command to abort one workunit or informations of the current BOINC credit points.

Furthermore, general descriptive information about the application can be defined with the statement `info`.

```
info {
 author="Christian Benjamin Ries";
 email="cries@fh-bielefeld.de";
 license="FH Bielefeld";
 description="Spinhenge@home Example";
 project="Spinhenge@home";
 version="3.16";
}
```

E. Visual Grid Framework

Fig. 7 shows the proposed *Visual Grid Framework* layer which is defined between the layer that describes the underlying computer hardware, the BOINC framework, and the *VisualGridML Genmodel*. The *Visual Grid Framework* offers a less complex access to the BOINC functionalities and handles the creation of applications for different platform targets, e.g. Intel 686 based 32-bit or 64-bit architectures.

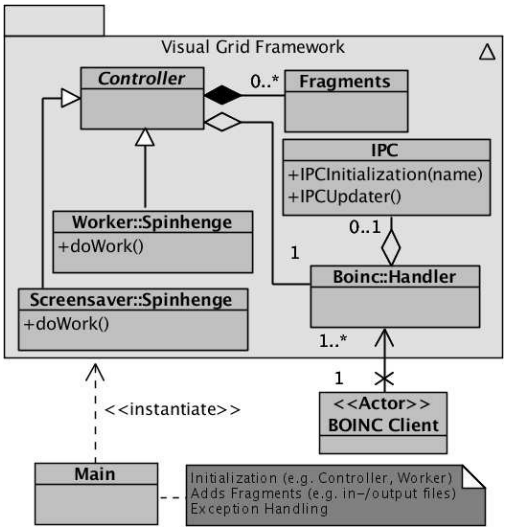

Fig. 8. Composition of the Visual Grid Framework Approach

The left side describes one approach to generate the function calls and logical program parts between the textual-/graphical modeling tools, fragments, *VisualGridML Genmodel*, *Visual Grid Framework*, and BOINC Framework. In Fig. 8 we show how the *Visual Grid Layer* works.

The `Main` class is generated by the code generation process and contains the implementation of the starting routines. This routine instantiates the interface to the BOINC framework as well as to the scientific application. The entry point of the scientific application is called using the *doWork()* functions of the *Worker::Spinhenge* and *Screensaver::Spinhenge* classes. These two classes are specializations of the abstract *Controller* class. The *Controller* class keeps track of all necessary data management, e.g. input, and output data files, checkpoint definitions, etc. The *IPC* class (Interprocess-Communication) is completely generated and implements the initialization and update routines. This class is used by the *Worker::Spinhenge* and *Screensaver::Spinhenge* classes for the exchange of data. The external *BOINC Client* class is an actor and therefore represents just the interface to the client. The BOINC Client has the full control of the executed applications and processes.

As a consequence, there exist two ways to generate an application within the *Visual Grid Framework*, (1) creation of a class which is derived from the abstract *Controller* class, (2) the definition of AOP *joinpoints* and *advices* as defined by the DSL in Sec. IV-D.

As a first test of our approach we have created an application which is similar to the BOINC sample to perform a transformation of lowercase to uppercase texts. The BOINC sample is based on approximately 400 lines of code, including the makefile, C++ header, source files, and 31 BOINC specific function calls. In contrast to this, our generated application contains only about 90 lines of code, including 60 lines of DSL code, and 30 lines of application specific code which performs the transformation. All defined dependencies, e.g. for the initialization of the process or exception handling, are resolved by generating files. Furthermore, a makefile is generated which on execution builds the complete application for the defined client platform architecture.

V. Conclusion

We have presented a first modeling language approach for developing Public Resource Computing applications on the basis of BOINC. We have demonstrated that the complete BOINC framework can be divided into a few logical packages that provide the necessary graphical and textual model elements to allow for a model-based development of applications with subsequent source code generation. Our approach is technically realized by using standardized and well-defined technologies [26]. Thus far, we have just implemented a small part of the BOINC functionalities. Key features like graphical and textual modeling elements, the *VisualGridML Genmodel* have been defined to an extent that we could show the general feasibility of our approach. However, further investigations with regard to the abstraction of the system architecture, dependencies, error checking, and the transformation to different target languages are needed. We have successfully performed a first test of our approach by modeling an existing BOINC application with just a few lines of DSL code using external libraries for the core computational routines and abstraction of the BOINC functionalities. However, most steps of our modeling process are still performed manually, and we are currently working on the creation of a unified development environment that supports the wide range of technologies, including a one and only graphical modeling framework which minimizes the need of textual modeling fragments, automatic dependencies resolving, completely error-free code generation, and higher support for legacy applications.

References

[1] D. P. Anderson, *BOINC: A System for Public-Resource Computing and Storage*, 5th IEEE/ACM International Workshop on Grid Computing. November 8, 2004, Pittsburgh, USA

[2] D. P. Anderson, C. Christensen, and B. Allen, *Designing a Runtime System for Volunteer Computing*, IEEE Computer, 2006

[3] S. Apel, T. Leich, M. Rosenmùller, and G. Saake, *FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C++*, Technical Report, Department of Computer Science, Otto-von-Guericke University, Magdeburg, Germany, 2005

[4] S. Apel, T. Leich, and G. Saake, *Aspectual Mixin Layers: Aspects and Features in Concert*, In Proceedings of International Conference on Software Engineering (ICSE), 2006

[5] S. Apel and D Batory, *When to Use Features and Aspects? A Case Study*. In Proceedings of ACM SIGPLAN 5th International Conference on Generative Programming and Component Engineering (GPCE'06), Portland, Oregon, October 2006

[6] H. Brunelière, J. Cabot, and F. Houault, *Combining Model-Driven Engineering and Cloud Computing*, AtlanMod, INRIA RBA Center & EMN, France, Nantes, 2010

[7] D. Batory, J. N. Sarvela, and A Rauschmayer, *Scaling Step-Wise Refinement*, IEEE Transactions on Software Engineering (TSE), 30(6), 2004

[8] T. Estrada, M. Taufer, and D. P. Anderson, *Performance Prediction and Analysis of BOINC Projects: An Empirical Study with EmBOINC*, in J Grid Computing, Springer, 2009

[9] R. C. Gronback, E. Gamma, L. Nackmann, and J. Wiegand, *Eclipse Modeling Project, A Domain-Specific Language (DSL) Toolkit*, Addison-Wesley, 2009, ISBN: 978-0-321-53407-1

[10] A. Hessellung, *Domain-Specific Multimodeling*, IT University of Copenhagen, Denmark, 2008
[11] P. Kacsuk, J. Kovacs, Z. Farkas, A. C. Marosi, G. Gombas and Z. Balaton, *SZTAKI Desktop Grid (SZDG): A Flexible and Scalable Desktop Grid System*, Journal of Grid Computing, 2009
[12] G. Kiczales et al., *Aspect-Oriented Programming*, In Proceedings of European Conference ob Object-Oriented Programming (ECOOP), 1997
[13] D. Kirk, and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-On Approach*, Morgan Kaufman Publ Inc, 2010, ISBN: 978-0123814722
[14] K. Lieberherr, D. H. Lorenz, and J. Ovlinger, *Aspectual Collaborations: Combining Modules and Aspects*, The Computer Journal, 46(5), 2003
[15] A. C. Marosi, Z. Balaton, and P. Kacsuk, *GenWrapper: A Generic Wrapper for Running Lagacy Applications on Desktop Grids*, 3rd Workshop on Desktop Grids and Volunteer Computing Systems (PCGrid 2009), 2009 May, Rome, Italy
[16] M. Mezini and K. Ostermann, *Variability Management with Feature-Oriented Programming ans Aspects*, In Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2004
[17] OMG Adopted Specification formal/2009-02-02, *OMG Unified Modeling LanguageTM (OMG UML)*, Superstructure, Version 2.2, OMG, 2009
[18] OMG Adopted Specification formal/2010-02-01, *OMG Object Constraint Languag*e, Version 2.2, OMG, 2010
[19] C. B. Ries, and C. Schrñder, *ComsolGrid - A framework for performing large-scale parameter studies using Comsol Multiphysics and BOINC*, COMSOL Conference, Paris, France, 2010
[20] C. B. Ries, *Performance measuring and automatic calibration of BOINC installations*, University of Applied Sciences Bielefeld, Germany, unpublished
[21] C. B. Ries, T. Hilbig, C. Schrñder et al., *Spinhenge@home - Monte Carlo Metropolis*, Version 3.16, University of Applied Sciences Bielefeld, Germany, http://spin.fh-bielefeld.de
[22] C. B. Ries, T. Hilbig, and C. Schrñder, *UML 2.2 Profile: Visu@lGridML*, University of Applied Sciences Bielefeld, Germany, unpublished
[23] C. Schrñder, "Spinhenge@home - in search of tomorrow's nanomagnetic application", *to appear in Distributed & Grid Computing - Science Made Transparent for Everyone. Principles, Applications and Supporting Communities*, 2010
[24] O. Spinczyk, D. Lohmann, and M. Urban, *Advances in AOP with AspectC++*, Software Methodologies, Tools and Techniques (SoMeT 2005), IOS Press, September, 2005, Tokyo, Japan
[25] T. Imamura, B. Dillaway, and E. Simon, *XML Encryption Syntax and Processing*, W3C, December, 2002, http://www.w3.org/TR/xmlenc-core
[26] Xtext - programming language framework, Xpand - a template language, http://www.eclipse.org/modeling/mdt
[27] U. Zdun, *Concepts for Model-Driven Design and Evolution of Domain-Specific Languages*, In Proceedings of the International Workshop on Software Factories OOPSLA, pp. 1-6, October, 2005