Model-based Generation of Workunits, Computation Sequences, Series and Service Interfaces for BOINC based Projects

Christian Benjamin Ries
Computational Materials Science and Engineering (CMSE)
University of Applied Sciences
Bielefeld, Germany
www.visualgrid.org

Christian Schröder
Computational Materials Science and Engineering (CMSE)
University of Applied Sciences
Bielefeld, Germany
Christian.Schroeder@fh-bielefeld.de

Vic Grout
Creative and Applied Research for the Digital Society (CARDS)
Glyndŵr University, United Kingdom
v.grout@glyndwr.ac.uk

***Abstract*—Berkeley Open Infrastructure for Network Computing (BOINC) is a popular Grid Computing (GC) framework which allows the creation of high performance computing installations by means of Public Resource Computing (PRC). With BOINC's help one can solve large scale and complex computational problems. A fundamental element of BOINC is its so-called workunits (WUs), each computer works on its own WUs independently from each other and sends back its result to BOINC's project server. Handling of WUs is a challenging process: (1) the order of used input files is important, (2) even more contributory components has to know how these input files are structured and on which data format are they based for an accurate WU processing. Small modifications can have a high impact to a BOINC project. Indeed scientific applications, BOINC's components, and third-party applications all have to be adjusted to have a correctly running project with desired the functionality. This can be a highly error-prone and time-consuming task. In this paper we present a Unified Modeling Language (UML) model to give a high abstraction for BOINC's WU handling. Only a model description and a corresponding code-generator are necessary to construct a WU handling infrastructure with less development and implementation effort: (a) one model to fit most WU cases and (b) essential interfaces for WU access.**

***Keywords*—BOINC, Code Generation, Modelling, UML, Work**

I. Introduction

SEt-up of a Berkeley Open Infrastructure for Network Computing (BOINC) project can be a challenging and sophisticating task. Despite the fact that it is necessary to implement a scientific application (SAPP) [10] and to establish a fully operable server infrastructure [7], moreover it is necessary to describe how SAPP and all BOINC components handle computational jobs. Here, participating clients retrieve a project specific SAPP from a BOINC project (BP) server along with so-called workunits (WUs), i.e. a number of parameter usually provided in data files of ASCII or binary format that are optionally needed by the application to perform specific tasks. The idea in this paper is to have a Unified Modeling Language (UML) model and code-generation (CG) facilities, which have to support developers with the ability to generate all required WU configurations, interfaces for opening and accessing WUs, and creating one or more computational series and sequences, i.e. different computational jobs with varied runtime configurations.

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

A. *Unified Modeling Language & Object Constraint Language*

One of the primary goals of UML is to advance the state of the industry by enabling object visual modeling tool interoperability [15]. Since version 2.2, UML has 14 different diagram types subdivided in three categories: (1) structure diagrams, (2) behavior diagrams, and (3) interaction diagrams. In this paper we use the *Class* and *State Machine* diagrams. Class diagrams are used to specify system related elements, e.g. a class can describe a SAPP. An instantiated class element is seen as an object and mostly it is an executable instance. UML state machines help to model discrete behavior through finite state-transitions systems. It can be used to visualize the current state of one system, and orthogonal regions allow to model client-server state-machines where each side is working independently. The Object Constraint Language (OCL) is used to express constraints and properties of UML model elements [16].

B. *BOINC's Workunit System*

BOINC uses a fine-grained file based system to set-up WUs for a BOINC project (BP). WUs are packages with descriptions of input and output data needed by the SAPP to perform specific tasks [1]. Before a WU can be added to a BP, it is necessary to create several input files with planned to use datasets for one computation. Two additional template files are required: (1) an input template to describe which files are used as input, how they are ordered and which flags for them are set, and (2) a result template to describe how output files must be named by the SAPP, or how big in bytes they can be [3].

C. Research Topics

To make the handling of WUs easier some questions arise and we will work on them within this paper.

- Which UML elements are necessary to create a model for WU creation?
- How can we model a sequential queue for WU progressing? The answer to this question should make it possible to have WUs with the need of pre-processed results by one or more other WUs.
- How can BOINC's validator and assimilator access result's data on a higher abstraction level? In addition, is it possible to have only one interface or description which makes it possible to allow access by all BOINC components? Here, BOINC's validator is responsible for validating a WU and developers of a BP can implement their own validator routines. The default behavior of BOINC's assimilator is storing of results within a file system.
- How can we track the lifetime of WUs when they are used in different scenarios, e.g. one WU is used within a sequential performed queue?

This paper can be seen as the conjunction of previous work [6], in which BOINC's services are described with UML to be deployable on server farms. This is why «Application» is added in Fig. 1 where previous work is followed in this paper.

The remainder of this paper is organised as follows. Section II describes the problematic of BOINC's architecture to handle different defined WUs for varied kinds of computations. Next, Section III proposes our idea as to how we can fill the gap of BOINC's problematic to utilize it with an easier and less vulnerable interface. In Section IV we use our UML model and apply it to a small case-study. Finally, Section V concludes this paper and Section VI suggests future work.

II. Problematic of BOINC's Architecture

WUs are packages with descriptions of input and output data [1]. These WUs are fundamental pieces for BOINC and contain information on how these data are defined and formatted, i.e. binary data or plain text and functionalities to describe how several data items can be used. The flexibility to define arbitrary structured WUs and input files can be a complex issue. It has been shown, that WUs within a BP are crucial elements and are essential for the BP success [9], [11]. At the time a BP is being established it must be defined how all BOINC components have to handle WUs, otherwise WUs will stop immediately wrongly configured and, as a result, without proper working components. A BOINC administrator needs answers to several questions before a BP can be set-up as a fully operable system. Certainly we think about our computational concern and how we can solve this problem firstly. In this paper we will not discuss this difficulty, previous work has focused on this field of activity [5], [10]. In this paper we will discuss a solution for the following questions:

- Is all information about WU's structure available at the beginning of it's use or are they gathered continuously during BOINC's runtime? Here, it is also important to define how continuously created WUs differ from each other. It is necessary to know if their content differs and if they must be restructured or not, e.g. if different sigma values have to be set for statistical computations.
- How should WUs be opened and how should all potentially contained sub-elements be handled by a SAPP? Are the nested data defined as plain-text, or as encrypted text, or maybe a binary format?
- It is not only the WU input files that are important. The result files are also essential for the success of a computation. In the later BOINC process they must be validated and subsequently stored by an assimilator to make results usable for particular later cases.
- The assimilation process is used to store results, but what if one WU does not have enough results? E.g. one WU is distributed to three hosts, a minimum of two results must be returned but in one scenario two hosts are too late — deadline is reached — and only one result is available. In this case, BOINC's transitioner will flag the missing results as *overdue*, then directly flagged as *ready for assimilation* by BOINC's validator [2] and after this the assimilation process could create a duplicated WU for a retry. This can be done periodically until the WU is completely returned and successfully validated, or after some failed tries the available results can be stored in a database or on the file system which can be defined for failed results.
- Under some circumstances a computation relies on different sets of runtime parameters or they must adhere to a sequence of different runs, i.e. a result of a WU must be used as input for another WUs. In this case, the results must be converted to the right format of a new WU and it can be necessary to modify mentioned attributes for the different purposes of a WU, e.g. an unit conversion can be required before a WU result is usable for subsequent computations.

BOINC's architecture relies heavily on a fragile methodology; if one or more software components are misconfigured or disabled the WU handling chain will be stopped on the failed element, i.e. if the validator is not working properly no validation of returned WU results is executed and as a consequence the WU will never complete.

BOINC's WU consists of two template files, additional input files and, during computation, created output files. Template files are based on an XML [12] format and therefore they are not really human readable and XML-tags can be misspelled very easily. More important is the fact that all input files must be described within this template file and must have a specific order. In the header of the input template the numbering of input files is defined. After this part each file has optional attributes, e.g. a file is sticky and will not be deleted after one computation on one host. A similar approach is used for the description of result files. These files and the part of BOINC's framework for WU creation are elementary and every BOINC

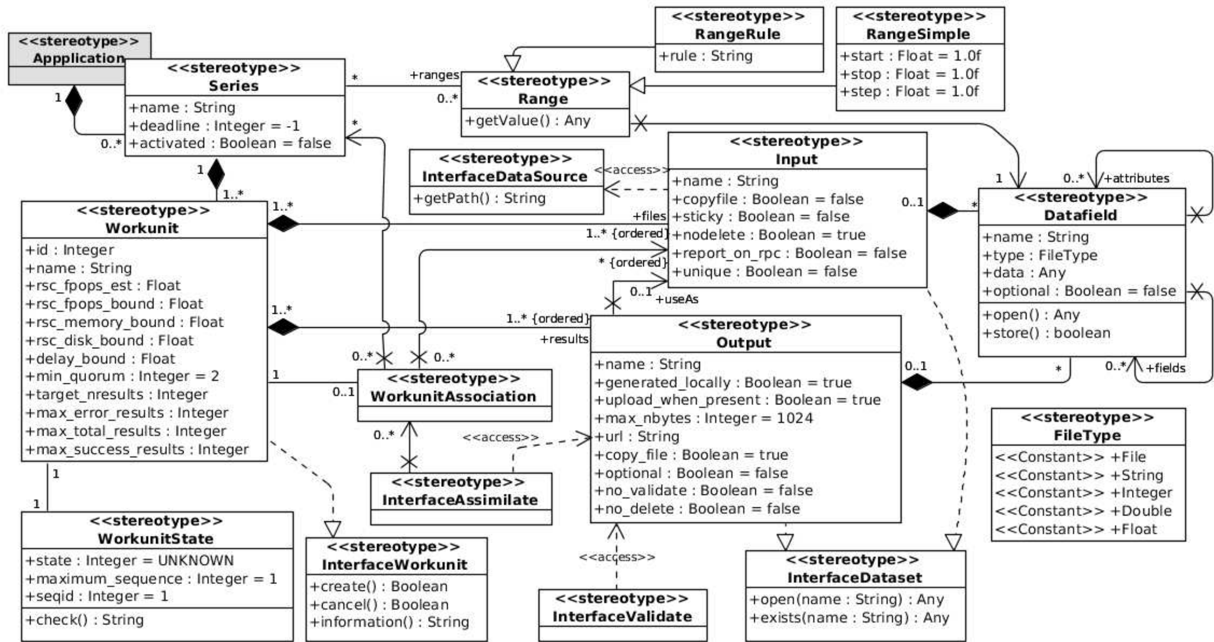

Fig. 1. Unified Modeling Language (UML) class diagram to abstract workunit structures. In the top area different *Ranges* for values within *Input*'s *Datafields* can be assigned. On the right hand-side *Datafield* allows to configure an arbitrary format for input and output files, the enumeration *FileTypes* provides different standard formats. In case one *Datafield* has *#File* as value for *type* no associated *attributes* or *fields* are allowed. Three stereotyped interfaces helps to access input and output files: (1) «InterfaceDataset», (2) «InterfaceValidate», and (3) «InterfaceAssimilate». Associations enable one to set-up different workunit processing scenarios: (1) static processing, (2) continuous processing, and (3) dynamic processing as seen in Fig. 2.

administrator or developer must give attention to this process. Several steps are required to add WUs for one BP: (1) one or more input files must be copied to BOINC's download hierarchy, (2) mentioned template files must be created, and (3) all input files must be arranged in the right order when BOINC's functionalities for WU creation are called. Each change within one of these steps has an impact on the other steps and must be adapted.

III. Modeling of Work Packages for BOINC

For computations with BOINC it is necessary to have one or more WUs which contain descriptive information on how to execute these computations. WUs could contain several files, e.g. additional configurations, data sets, definitions of algorithms and arbitrary extra files. Fig. 1 shows one part of our UML definition for WU definition. Here, we can define WU's input and output files and multiple data fields for these files.

The three stereotypes «Workunit», «Input», and «Output» are directly based on BOINC's WU system. All tag-values of these three stereotypes are directly mapped to attributes of BOINC's templates, the only exception is tag-value *unique*. If *unique* is *true* all input files are renamed to be unique within a BP. The other presented stereotypes are extensions to fulfil our UML model.

«Workunit» must be associated by «Series» and that must be associated to «Application» [6]. This association unites previous work with this paper.

A. Input-/Output Files and Datafields

«Input» is used to describe input files and «Output» describes result files. A WU can own several file instances and each of them can have distinct «Datafields». «Datafields» are used to describe data for input files, the data format is not restricted and for this reason two methods are defined: (1) *open()* is used to access data, and (2) *store()* is used to add datasets. The reason for these methods is, that the embedded data can have different formats, i.e. values have to be encrypted during saving or specific embedded function calls must be used during data access in case a file is packed as a ZIP-archive [18]. These functions can set by a developer to supply special opening and storing methods for currently unknown data types. There is no reason to allow a «Datafield» to be used by «Input» and «Output» at the same time, as a consequence only one owner of the root «Datafield» is allowed. This root and all other instances of «Datafield» have two associations which can be used to create tree structures with several pieces of information for a WU embedded in a «Input» file. With this methodology different structures are possible, e.g. a XML structure can be created as shown in Listing 1. The use of these associations is restricted, if one «Datafield» is associated by *attributes*, then it can not have additional associations. Each «Datafield» has the tag-values *name*, *type*, *data*, and *optional*. *Name* must be user-defined at any time when a «Datafield» is used, the other tag-values are optional and their use depends on the task. Listing 1 shows the use of the first three tag-values:

name *person*, *interests*, and *topics* are names,

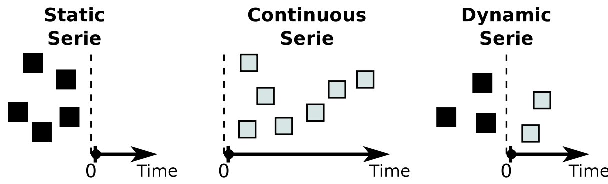

Fig. 2. Our UML model allows us to define «Series» in three different ways: (1) all WUs must be available before a BP is started, (2) during runtime WUs are created and added continuously, and (3) a mix of the first and second; some WUs are available at the beginning and during runtime additional WUs are added to one BP.

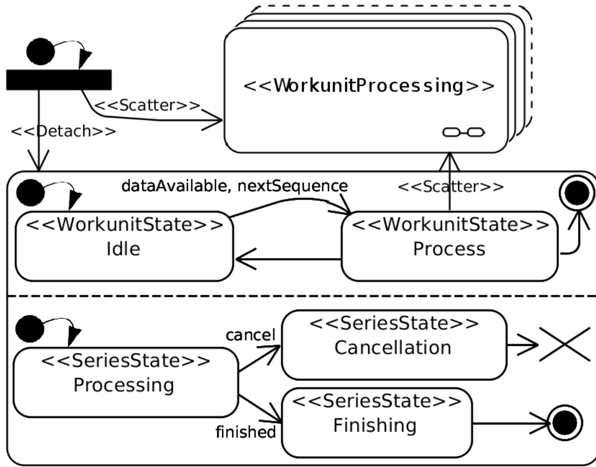

Fig. 3. First part of our UML statechart diagram to monitor instances of «Workunit» and «Series». State's top region is responsible for WU monitoring and creates new WUs when data is available or next WU in a sequence has to be performed. The bottom region monitors a «Series» and handles canceling events for a «Series» instance or if its finished and results can be merged.

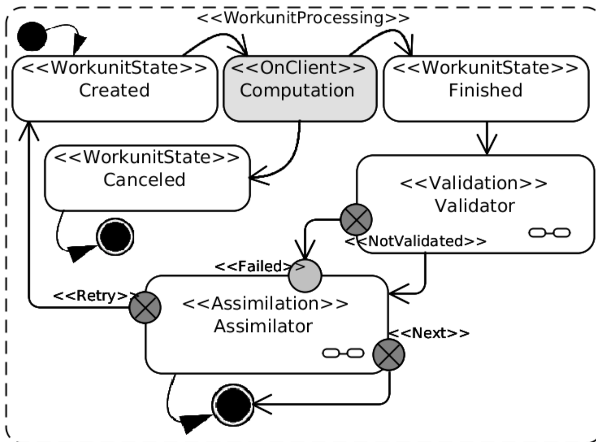

Fig. 4. Second part of our UML statechart diagram for WUs. During computation of one WU, clients can decide to cancel current WU, and therefore it has a changed WU state. When it is finished it will be validated, if this validation failed the exit pseudostate is used. The followed assimilation state can decide to retry this WU and a new WU is created with same «Input» values. If this WU is in a sequence, "Next" is used otherwise the statechart is finished.

type *"C.B.Ries"* and *"Research, Sport"* are of the enumeration type *FileType::String*, and

data mentioned string values are the real embedded information.

```
<person name="C.B.Ries">
  <interests topics="Research, Sport"/>
</person>
```

Listing 1. Example of «Datafield» usage to define a XML structure.

B. Rule-based Creation of Datafield Values

During WU creation data fields of input files or the input files themselves can be specified by «Range». Therefore values can be generated by «Range» specializations: (a) «RangeRule» and (b) «RangeSimple». With «RangeRule» a rule-set for value creation can be defined. For this purpose the tag-value *rule* can be filled with a user-defined rule, e.g. each WU within a specific «Series» can have a corresponding mode for algorithms. «Range» defines an operation *getValue()* which is used to query the related «Datafield» value. As shown in Fig. 1 each rule can only modify one «Datafield». «RangeSimple» is used to have a range-loop for one specific «Datafield», for this reason three tag-values are defined: (1) *start*, (2) *stop*, and (3) *step*. In combination with different additional rules each call of *getValue()* can increment the lower-bound value *start* by *step* to the upper-bound *stop*. One «Range» can be owned by several «Series», as a consequence it must be possible to retrieve which «Series» is calling *getValue()*. For this reason tag-value *activated* is specified. When this tag-value is valued by *true*, the association between «Range» and «Series» can be used to query the currently used «Series». This allows «Range» to access all information of a «Series» with associated «Workunits».

C. Continuous Creation of Workunits for Series

As seen in Fig. 2 a BP can have different scenarios for WU creation:

- **Static** All WUs are created before a «Series» will be created. Only these known WUs are handled by a BP.
- **Continuous** This configuration has no WUs at the beginning of a «Series». WUs are created on demand, e.g. when new data packages are available or when a time slot is reached.
- **Dynamic** In this configuration the previous two possibilities are merged.

These three approaches are supported by our model. «InterfaceDataSource» (IDS) is an interface which is implemented by a «Service» component for WU creation [6]. This component could have several connections to data sources. When these data sources signals new available data packages, IDS provides with the help of *getPath()* a file path which is usable for «Input» and a corresponding «Datafield» has *#File* as *type*.

Fig. 3 shows the first statechart diagram for our modelling approach. Depending on your BOINC scenario you can define how WUs are created. For all mentioned scenarios the statechart will always start at the initial point in the top-left corner. Immediately the process is subdivided into two parts with two transitions stereotyped by «Scatter» and «Detach». The BOINC's WUs are independently processed on

the client side from other processes. In addition to this WUs are structure elements and that's why they are not conceived to have a behavior. Other components have to deal with them and as a consequence these components can have behaviour definitions. While all WUs are public within BOINC's domain any component has access and can modify them.

«Detach» creates two orthogonal regions for the lifetime monitoring of one «Series» and all associated WUs. The top region is responsible for WU monitoring and the bottom region monitors the current «Series». The transition between "Idle" and "Process" is triggered by *dataAvailable* and *nextSequence*. In this transition *dataAvailable* is called by the IDS, and thereupon the file path is used to define a new WU. Fig. 4 shows the statechart of a single WU. In that statechart "Assimilation" has a "Next" named exit pseudostate and *nextSequence* is triggered when this exit is entered. As a result a new WU is created. It is defined that this exit pseudostate can only be used when a WU is part of a sequence as described in the next section. As at the initial point of this statechart, all available and new WUs are scattered and within this statechart are monitored. The top region is left when no more WUs are in process. The bottom region monitors the lifetime of a «Series» and "Processing" is only left under two circumstances: (1) the «Series» has to be canceled and (2) processing is complete and all results can be merged, which can be done in "Finishing", e.g. an average over all results of a monte-carlo simulation can be calculated.

D. Sequences of Workunits

In [4] a system for remote creation of chained WUs is shown, where one result can be used as input for other WUs. In our model we can handle a similar task. «WorkunitAssociation» enables one to define a «Series» with sequential computations. «InterfaceAssimilate» delegates these computations and can have an association to «WorkunitAssociation». As mentioned in the previous section the "Next" exit pseudostate in Fig. 4 is used when one WU is assimilated and has additional WUs to be performed. The following pseudocode demonstrates how the assimilation process can decide if one WU is in a sequence and if a WU follows:

```
Let ws As workunitAssociation.workunit.workunitState
If ws.seqid < ws.maximum_sequence Then
 For ro In Output
  Set workunitAssociation.input = ro
    Where
      workunitAssociation.input.name = ro.useAs.name
 EndFor
EndIf
ws.seqid = ws.seqid + 1
```

A new WU is filled with «Datafield» values of one «Output» when they are associated by *useAs*. As a consequence *useAs* must only be set when «Output» is used for one «Input» configuration. The fact is, when no *useAs* is available then it makes no sense to check for a sequence.

In the case when all results of a WU are required, BOINC's assimilator can create additional WUs with a duplicated configuration, e.g. a WU is missed to complete a sequence and is too often canceled by BOINC clients or WU's *delay_bound*[1] is reached. For this occurrence the WU can be copied and added to a «Series».

When a WU is part of a sequence, the WU's name has a special format to distinguish WUs. A similar approach for rBOINC is used [4]. rBOINC defines a specialized WU name and we modify this format to **"NNN-SEQ-XX-YY"**:

- **NNN** is the name of the WU, and
- **SEQ** is a start pattern for a sequence description.

Here the embedded string **"-XX-YY-"** is defined as follows: **XX** is the current sequence id and **YY** is used for the maximum number of sequences. This WU name format is used to select sequenced WUs in section III-F.

E. State of Workunit Computation

«WorkunitState» is associated by «Workunit» (WU) and from the beginning of its existence the state of a WU can be queried at any time. The tag-value *state* holds the current state and can be valued with the following variables:

- **CREATED** WU is created.
- **FAILED** WU has failed and can not be finished.
- **COMPUTATION** WU is in progress and one or more clients work on it.
- **DONE** Enough clients have worked on one WU and it can be moved to the validation and assimilation process.
- **VALIDATION** WU has to be validated.
- **ASSIMILATION** WU has to be assimilated.
- **CANCELED** WU is canceled by an administrator or by other processes, e.g. when sequenced WUs have failed or are canceled.
- **FINISHED** WU is finished and ready for later use, e.g. to create a new «Series» or to use their computational results.

For the UML model it is important what the state of a WU is, as a matter of fact the state value is responsible for deciding which actions are performed during the WU processing, i.e. when a WU fails the assimilation process has to decide if it should be performed again. The accessory method *check()* is used to query the current state of a WU and returns a descriptive text value, i.e. the string contains the current state with additional more precise information such as the *timestamp* of the last check or how long a WU is currently processing. The other two tag-values *maximum_sequence* and *seqid* are used for the «WorkunitAssociation» in the next section.

F. Cancelation of Series and Workunits

A WU can be canceled at any time. This is done by *InterfaceWorkunit::cancel()* and it is necessary to cancel WUs which are related to a cancelled WU, i.e. when the current WU is cancelled and has associated WUs, these must be also cancelled because they can never be processed with missing «Input» values. «WorkunitAssociation» has an additional OCL operation to query which WUs are in the current sequence:

[1] Deadline of one workunit.

```
WorkunitAssociation::querySequencedWorkunits(
  seqid : Integer, name : String) : Set(Input);
querySequencedWorkunits =
  self.series->select( s | s.workunit->select(
      w | w.workunitState.seqid > seqid
          AND
          -- NNN-SEQ-XX-YY
           w.name.substring(1,
               w.name.strpos("-SEQ")) = name
    )
  )
```

With this OCL statement, WUs can be selected which are later defined within a sequence and as a consequence they can be cancelled. Following cancelled WUs can have other associated WUs and they must be cancelled. The call of *Workunit::cancel()* is used to cancel one WU. «Series» can be cancelled with this concept, it is enough to call *cancel()* during the iteration of all associated WUs.

G. Service Interfaces

BOINC has several components which need to access «Input», «Output» and their embedded «Datafield» fields. Fig. 1 shows three interfaces to access them: (1) «InterfaceDataset», (2) «InterfaceValidate» and (3) «InterfaceAssimilate». With the help of this structure all functionalities can be generated and this makes the access more comfortable. Changes in the model are automatically resolved and interfaces are always valid for use.

IV. Case-Study

Our case-study modifies a movie, i.e. a movie is fragmented in single image sequences and basic image processing algorithm are applied to these sequences, some results could be seen on the project website [8]. Fig. 5 shows our use-case where one video is added by *C.B.Ries*, with the help of inotify [13] one BP is triggered by *dataAvailable* and WUs are created on demand. During WU adding it has to be clear which kind of data format is used, i.e. in our use-case we add a complete movie and subsequent implementations has to prepare this movie for WU creation. In this scenario we create ZIP-archives automatically and fill them with a number of image sequences.

Five «Series» are defined, in this case only the first fourth can be processed immediately. As shown in Equation 1 the fifth «Series» needs the result of the first four «Series».

$$\begin{bmatrix} Series\ 1\ (normalize) \\ Series\ 2\ (painting) \\ Series\ 3\ (negate) \\ Series\ 4\ (edge) \end{bmatrix} \Rightarrow\ Series\ 5\ (merge) \qquad (1)$$

All «Series» instances have a different runtime configuration, e.g. in «Series» number four the image is manipulated by an edge algorithm. The fifth «Series» merges all previous results where for each image sequence they are added to a 2×2 raster image. On the right-hand side of Fig. 5 these different configurations are shown where the bottom configuration shows the mode “edge” for image manipulation and in the top configuration “merge” is assigned.

The fifth «Series» is not started until the other «Series» has finished. It is important to notice that the SAPP is always the same, only the input files are changed. In the first four computations only two files are necessary: (1) configuration for the mode of configuration and (2) the mentioned ZIP-archive with movie sequences. The last computation is altered and needs five files: (1) configuration as before with different values and (2) all four input files which are created by the other four computations.

This use-case describes a *dynamic series* where some WUs available on BP’s start and additional WUs created during runtime. In the case that one of the first four series is canceled, the fifth series can never be started because of missing input data. This is solved by our statechart construction in Fig. 3 where the “Cancelation” state is responsible for cancelling all related WUs and «Series» instances.

V. Conclusion

In this paper we describe a UML model for WU creation and how the lifetime of WU series and individual WUs can be monitored. We have shown that only one model description is necessary to allow BOINC related components to access WU’s input and output files, i.e. BOINC’s validator and assimilator must not be changed to access the WU, all necessary code elements can be generated with one model description. With our model it is possible to set-up different computational scenarios where WUs are generated statically, continuously or mixed by these two approaches. WUs can be added to a process before a BP is started or they can be added on-demand during the runtime of a BP. With the help of UML statechart diagrams we can set-up a BP configuration where we can define how *overdue* or *absent* WUs are handled. It is possible to recreate them or if it is wished, the related computational series and related WUs are aborted.

The proposed UML modeling approach can help to reduce errors during administration of BPs. As a matter of fact, in traditional BPs it is necessary to reconfigure and reimplement several parts when only one configuration is changed, i.e. if the format of computational results is changed then all related components such as BOINC’s validator and assimilator have to be similarly changed. Furthermore, adding or removing input files for one computation has an impact on several BOINC parts: (1) non well readable XML input files must be changed, (2) the call of BOINC’s WU creation tools has to be altered, (3) altered files has to be copied to BOINC’s download hierarchy, and (4) (maybe) input files have to be generated or prepared. Our modeling approach solves all these problems with the help of UML and OCL.

VI. Future Work

WU’s performance can have restrictions, e.g. the use of floating-point operations or allocation of hard disk space can be limited. Currently it is not clear if UML can help to detect perfectly fitted values for this purpose. During our use-case tests we noted a large number of failed WUs because of

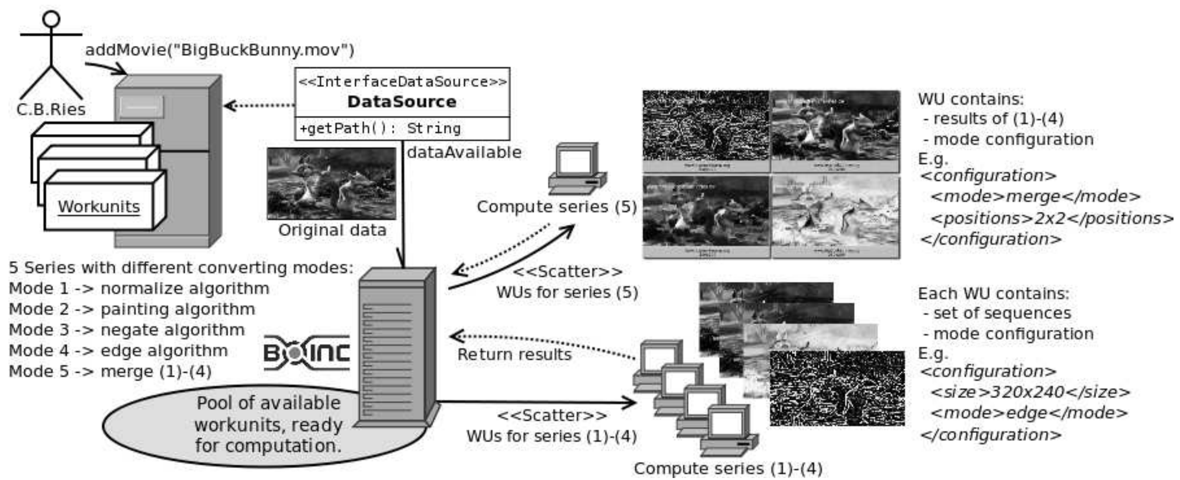

Fig. 5. Use-case to modify a movie: *C.B.Ries* adds a new movie to one server, running applications prepare this movie and fragment it into image sequences, thereupon these images are zipped into ZIP-archives. *DataSource* notifies a BP which automatically adds all announced WUs to this BP. Adding of one ZIP-archive implies four added WUs to this BP but with different configurations, each «Series» instance has a different mode for computation: *normalize*, *painting*, *negate*, and *edge*. These WUs are processed and the computational results are used as input for a fifth «Series», again with a changed mode for computation: *merge*.

wrongly adjusted boundary values for the restrictions mentioned. The set-up of this values has to be more precise and at best automatic. In future work we will work on this objective.

During the writing of this paper implementation with the support of this model are hard-coded and can not be changed — they are not flexible during runtime. One idea is to add a Domain-specific language (DSL) to describe WUs, series and sequences of computation. It could be possible to interpret this DSL during the runtime of a BP, to change the behavior of this BP and to generate code for all necessary components for WU handling on-demand.

Additional thought should also be spent on how our presented model can be used for conventional supercomputers, where other technologies like Message Parsing Interface (MPI) [14] or OpenMP [17] are used. It should be clear that MPI has to process workunits like BOINC, although admittedly with less input files; instead it uses more numerical values which are communicated between all involved computation nodes. The fact is that BOINC can be perfectly used to solve embarrassing parallel computational problems with less communication between all involved nodes. MPI enables one to use a distributed computing environment with several autonomous interacting nodes to achieve a common goal.

REFERENCES

[1] D. P. Anderson, C. Christensen, and B. Allen. “Designing a Runtime System for Volunteer Computing.” in *Proc. ACM/IEEE SC*, 2006, Article No. 126

[2] BOINC. “Backend program logic,” Internet: http://boinc.berkeley.edu/-trac/wiki/BackendLogic [Version 2]

[3] BOINC. “Submitting jobs,” Internet: http://boinc.berkeley.edu/trac/wiki/-JobSubmission [Version 19]

[4] T. Giorgino, M. J. Harvey and G. De Fabritiis. “Distributed computing as a virtual supercomputer: Tools to run and manage large-scale BOINC simulations”. *Computer Physics Communications*, vol. 181, February, 2010

[5] C. B. Ries. “BOINC - Hochleistungsrechnen mit Berkeley Open Infrastructure for Network Computing.” Berlin Heidelberg: Springer-Verlag, 2012

[6] C. B. Ries, C. Schröder, and V. Grout. “Approach of a UML Profile for Berkeley Open Infrastructure for Network Computing (BOINC),” in *Proc. ICCAIE*, 2011, pp. 483-488

[7] C. B. Ries, C. Schröder, and V. Grout. “Generation of an Integrated Development Environment (IDE) for Berkeley Open Infrastructure for Network Computing (BOINC),” in *Proc. SEIN*, 2011, pp. 67-76

[8] C. B. Ries and C. Schröder. “Public Resource Computing mit Boinc.” Linux-Magazin, vol. 3, pp. 106-110, March 2011. Internet: lm-boinc.sourceforge.net

[9] C. B. Ries and C. Schröder. “ComsolGrid - A Framework For Performing Large-Scale Parameter Studies Using Comsol Multiphysics and Berkeley Open Infrastructure for Network Computing (BOINC),” in *Proc. COMSOL Conf.*, Paris, 2010

[10] C. B. Ries, T. Hilbig, and C. Schröder. “A Modeling Language Approach for the Abstraction of the Berkeley Open Infrastructure for Network Computing (BOINC) Framework,” in *Proc. IEEE-IMCSIT*, 2010, pp. 663-670

[11] C. B. Ries. “ComsolGrid - Konzeption, Entwicklung und Implementierung eines Frameworks zur Kopplung von COMSOL Multiphysics und BOINC um hoch-skalierbare Parameterstudien zu erstellen.” M.Sc. thesis, University of Applied Sciences Bielefeld, Germany, 2010.

[12] W3C. “Extensible Markup Language (XML) 1.0 (Fifth Edition),” Internet: http://www.w3.org/TR/REC-xml/

[13] J. McCutchan, R. Love, and A. Griffis. “inotify - monitoring file system events,” Linux man pages(7)

[14] Message Passing Interface. “The Message Passing Interface (MPI) standard,” Internet: http://www.mcs.anl.gov/research/projects/mpi/ [18th February 2012]

[15] Object Management Group. “OMG Unified Modeling Language (OMG UML) Superstructure.” formal/2010-05-05, May, 2010.

[16] Object Management Group. “Object Constraint Language.” Version 2.2, Feb., 2010

[17] OpenMP. “The OpenMP API Specification for Parallel Programming,” Internet: http://www.openmp.org [18th February 2012]

[18] PKWARE. “APPNOTE.TXT - .ZIP File Format Specification.” Version 6.3.2, Sept., 2007