The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.28 e-ISSN: 2251-7545

Code Generation Approaches for an Automatic Transformation of the Unified Modeling Language to the Berkeley Open Infrastructure for Network Computing Framework

Christian Benjamin Ries, Vic Grout
Creative and Applied Research for the Digital Society (CARDS)
GlyndŴr University, United Kingdom
www.christianbenjaminries.de | *www.visualgrid.org*

***Abstract*—This paper describes the verification process of the UML4BOINC stereotypes and the semantic of the stereotypes. Several ways enable the verification and this paper presents three different ways: (i) specifications of Domain-specific Modeling Languages (DSMLs), (ii) the use of C++-Models, and (iii) the use of Visual-Models created with Visu@lGrid [12], [17]. As a consequence, specific code-generators for the transformation of these models are implemented into applicable parts for a Berkeley Open Infrastructure for Network Com- puting (BOINC) project [1]. As for the understanding of how the transformation is realised, a brief introduction about the language-recognition and the way how code-generators can be implemented by use of ANTLR (ANother Tool for Language Recognition) [11] is given. This paper does not cover all transformations because most of them can vary, i.e. they depend on the target language (e.g. C++) and how tool-vendors handle semantic-models. In addition, steps three and four are realised within the research iterations of this paper.**

***Keywords*-BOINC, Code Generation, Modelling, UML**

I. Introduction

Running of a Berkeley Open Infrastructure for Network Computing (BOINC) project can be a very time-consuming task. Despite the fact that it is necessary to implement a scientific application (SAPP) [19] and to establish a fully operable server infrastructure [17], moreover it is necessary to describe how SAPP and all BOINC components handle computational jobs. A SAPP has be implemented by developers and for individual projects several implementations has to be repeated. By the help of UML (Unified Modeling Language) [22] one can cope this work by doing it with the help of a prominent visual programming language. The idea in this paper is to have UML extension (so-called stereotypes) and code-generator (CG), which have to support developers with the ability to generate all required implementation.

Section II gives a briefly introduction how code generation works. Section III describes how the CG methodology in this paper is supported towards to fullfil the task of supporting BOINC developers by describing. In Section IV the support of state machine of the UML version 2.4 is scoped. An abstraction of BOINC's SAPP and services is shown in the Sections V–X. In the last Section a conclusion is given.

II. Code Generation

A. Approaches

The code-generation (CG) generator-backend is often realised by one of the following three approaches [4]:

- **Patterns:** This approach allows specifying a search pattern on the model graph. A specific output is generated for every match.
- **Templates:** As the name suggests, code-files are the basis in the target language. Expressions of a macro language are inserted or replacement patterns are used for specifying the generator instructions.
- **Tree-traversing:** This kind of code generation approach specifies a predetermined iteration path over the Abstract-syntax Tree (AST). Generation instructions are defined for every node type or leaf and executed when a node or leaf is passed.

These approaches can be combined. Accordingly, this paper applies a combination of the second and third approach, i.e. when the AST is created, the leafs of the AST are transformed into code-fragments and merged in existing template files. Fig. 1 shows the CG processing parts which are used; in particular seven parts supports the verification. Section III introduces the first part (1); the result of it allows the generation of C++-code which is used to create semantic models. Such a semantic model can be created in several ways: (3) with a Domain-specific Modeling Language (DSML) description, (4) as a direct C++ implementation based on the previous created semantic model and (5) as a visual description. These approaches end up in (6) and describe a BOINC project. The description is transformed in different parts which are relevant for BOINC, e.g. configurations, implementations of services and scientific applications or the creation of workunits. For the steps between (1), (2), (6), and the creation of BOINC parts, the software library (7) *libries* is used [13].

B. ANother Tool for Language Recognition (ANTLR)

ANTLR [11] is a parser generator used to implement DSL interpreters, compilers and other translators, e.g. for the handling of specific configuration file formats. Two steps

The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.28 e-ISSN: 2251-7545

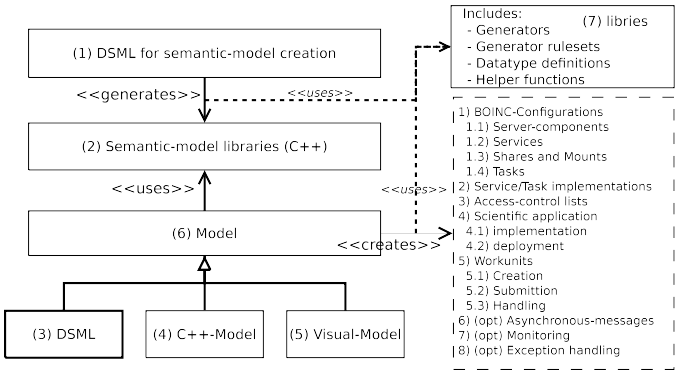

Figure 1. Architecture of the CG process; different models are used to generate a BOINC project with its used components and configurations.

are necessary for language recognition: *lexical analysis* and *parsing* [8]. Fig. 2 shows the general language recognition process. An input stream as `1 + 21 => res` consists of different bytes. This stream can be filtered by a *lexer* which splits the bytes into belonging tokens, based on an user-defined syntax or grammar: a number is defined as an `INT` (integer), the plus sign as `ADD` (addition) and `=>` is used to assign (`ASSIGN`) 'something' to `res`, the underscores are whitespaces and are not recognized. The input stream is filtered and transformed into several tokens. Every token has its own meaning, e.g. `ADD` can be used to sum-up two integers. In addition, the chain of different tokens can have a different meaning (so-called semantic). The parser is used to interpret the tokens, i.e. either an optional AST can be created and analysed; a direct interpretation is possible.

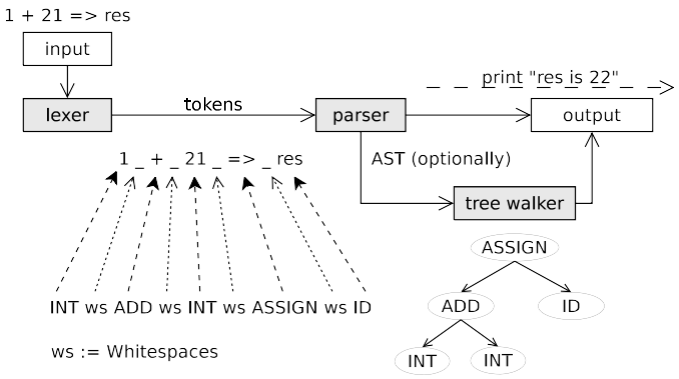

Figure 2. General process description of language recognition.

C. Abstract-syntax Tree (AST)

Generally, an AST is an abstract representation of any formatted input or token stream; it is optional, a direct interpretation can be done. The AST is created by adding mapping rules directly to the syntax of the new language:

```
multiplication: v1=INT '*' v2=INT->^(MUL $v1 $v2);
```

The operator `->` introduces the tree construction rules. The body of the rule is a list where the first element is the node type (`MUL`) and followed by the child nodes which are the factors for the multiplication in this case [5]. By means of this approach two syntax trees are defined, i.e. an input syntax on the left-hand side and an AST-syntax on the right-hand side.

D. ANTLR and the Extended Backus-Naur Form (EBNF)

The EBNF (Extended Backus-Naur Form) is a syntactic metalanguage which is a notation for defining the syntax of a language through a set of rules. Each rule names a part of the language (called a non-terminal symbol of a language) and then defines its possible forms. A terminal symbol of a language is an atom that cannot be split into smaller components of the language [7]. The EBNF has a small set of syntax rules and ANTLR applies most of the EBNFs with modifications as shown in Table I.

	EBNF	ANTLR
'a' zero or once	[a]	a?
'a' zero or more	{a}	a*
'a' once or more	a {a}	a+
'a' or 'b'	a | b & a | b	
('a' or 'b') and 'c'	(a | b) c & (a | b) c	

Table I
EXCERPT OF THE EBNF'S AND ANTLR'S SYNTAX [11], [7].

Fig. 2 shows a small summation (s1) of two integer values and an assignment to the variable *res*. The second summation (s2) allows unlimited summations, e.g. 20 + 22 + 222 => res. Listing 1 states the EBNF-syntax of this small command. Any token is defined in the last five lines, i.e. integers only consist of numbers. The last line defines whitespaces (WSs) and a special command forwards these characters to a hidden channel.

```
s1: INT ADD INT ASSIGN ID ;
s2: (INT (ADD INT)* ASSIGN ID)+ ;

ADD:    '+';
ASSIGN: '=>';
INT:    '0'..'9'+;
ID: ('a'..'z'|'A'..'Z'|'_')
    ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*;
WS: (' '|'\n'|'\r'|'\t')+ {$channel=HIDDEN;};
```

Listing 1. EBNF-syntax for the addition of two integers and assignment.

Finally, the ANTLR syntax enables the direct AST creation. Listing 2 shows how the AST on the bottom right-hand side of Fig. 2 can be described. The ANTLR operator `->` is used to map the EBNF-syntax to a valid AST-syntax. The result of this mapping and its dataset is shown in Fig. 3.

```
summation: leaf* -> leaf* ;
leaf: INT ('+' INT)* '=>' ID ->
   ^(ASSIGN ^(ADD INT*) ID);
```

Listing 2. Definition of the AST of the summation.

The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.28 e-ISSN: 2251-7545

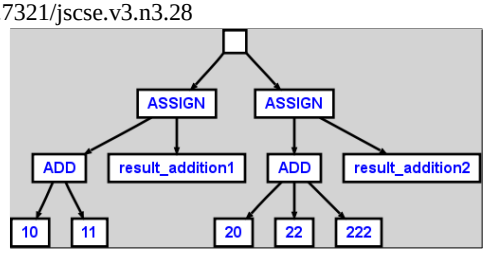

Figure 3. AST for the summation of unlimited integer values.

E. Result

Fig. 2 shows the parser during the language-recognition and with two outgoing transitions: (1) a direct transition to output and (2) a detour of AST creation are subsequently finished by an output generation. The first approach is also called "embedded translation" because it populates the semantic model from the embedding code to the parser; it implements semantic model at appropriate points in the parse [5]. The use of an additional AST can be more explicit and easier to maintain [10].

III. Generation of the Semantic Models

The research for this paper required the implementation of different approaches, in order to verify and modify models in an iterative way. All models are based on an object-oriented hierarchy; the creation is supported by a self-defined DSML and allows a compact specifying of all necessary information: (i) datatype specifications, (ii) creation of enumerations, (iii) classes with attributes and methods, (iv) derivations of classes and (v) different kinds of associations between classes and their different multiplicities, i.e. compositions and aggregations.

Listing 3 shows an example of the created DSML. Line 1 specifies additional header files. Lines 2-3 indicate a mapping of used datatypes within the DSML which are therefore mapped in the target's programming language datatypes. Line 4 specifies an enumeration usable as a datatype. The last lines specify three classes: *Project*, *Host* and *NIC*. *Project* has one attribute *name* and one association to one or more *Host* instances. *Host* itself is a composition of one or more *NIC* instances. The CG produces setter and getter methods and routines automatically in order to check the multiplicities during the adherence of instances to associations or compositions. Finally, every class gets a factory and destroyer functionality [6].

```
includes { "General.h" }
datatype String  "std::string";
datatype Integer "int";
OperatingSystem[Ubuntu=1, UbuntuServer=2] { }
Project { name : String [1];
          association hosts : Host [1..*];  }
Host    { composition network : NIC [1..*];
          // association : NIC [1..*];       }
NIC     { device : String [1];
          ipvalue : String [0..1];          }
```

Listing 3. DSML description for the creation of a class-hierarchy.

IV. Generation of Statemachines

UML's StateMachine package has been used for the research of this paper. There are some state machine implementations for the combination with C++ [2], [3], [9] which are partly based on the UML specification. All of them are restricted in their use (e.g. no support for orthogonal regions, no entry and exit points). Concerning this, the *UML 2 Statemachine for C++* (UML2STM4CPP) [20] and UML's StateMachine in version 2.4 support have been created and applied. UML2STM4CPP's DSL *Ries Statemachine* (RSM) enables the creation of hierarchy state machine constructs. Fig. 4 shows a small state machine which is based on an initialisation state, choice state, two system states, and a final state. Listing 4 shows an excerpt of the DSL definition for this state machine.

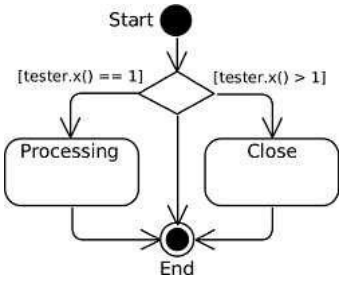

Figure 4. UML Pseudostates with guarded and default transition.

Listing 4 shows the DSL definition for the state machine in Fig. 4. The state machine has two ar- eas: (1) `States{...}` defines states and regions, and (2) `Transitions{...}` defines transitions, guards and events between states and regions. The DSL defines all used nodes, e.g. decision node, initial node, action node, and final node. Finally, the DSL is an equipment with extra code anchors:

- `<[onEntry CODE-]>` is executed on entering.
- `<[-CODE-]>` is the main part.
- `<[onExit CODE-]>` is executed on state's leaving.

The combination of these three anchors with the first anchor in lines 7-9 allows fulfilling the state machine with arbitrary behaviour.

```
Anchors <[-- #include <Tester.h>
            Tester tester; int x;--]>
Statemachine Pseudostates {
  States {
    Initial Start
    Simple Processing
      <[onEntry /* do some action */ --]>
      <[--      tester.processing(); --]>
      <[onExit  /* do some action */ --]>
    Simple Close
    Final End
  }
  Transitions {
    R: Start [tester.x() == 1]
             / tester.setX(2) == Processing
    R: Start [tester.x() > 1]
```

The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.28 e-ISSN: 2251-7545

```
                / tester.setX(3) == Close
      R: Start == End
      R: Processing == End
      R: Close / tester.show()    == End
  }
}
```

Listing 4. A UML2STM4CPP example of the state machine in Fig. 4.

V. Scientific Application

BOINC offers an Application-programming Interface (API) for the procedural implementation of scientific applications. UML is mainly an object-orientated specification and modelling approach. Generally, some tasks have to be done whenever a new scientific application (SAPP) has to be implemented. In order to avoid these incidents, an Object-orientated Programming (OOP) abstraction is implemented and set up on the top of BOINC's API [14], [19]. Fig. 5 shows the abstraction layers of this approach. The OOP layer provides a top-down approach where most of BOINC's functionalities are merged in objects and used as building blocks, e.g. BOINC's checkpoint mechanism is abstracted by the class `Ries::BOINC::MVC::Checkpoint` [13]. The biggest issue of a SAPP is the computation's core

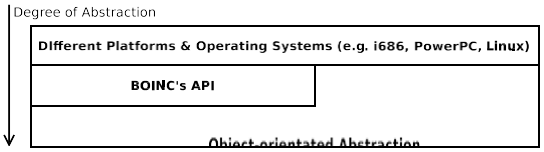

Figure 5. Object-orientated Programming layer for BOINC.

which provides the "intelligence" and creates engineering or scientific results. Fig. 6 shows an excerpt of the OOP approach. In this case, two scientific applications are given: (i) *LMBoinc* [18] and (ii) *Spinhenge* [21]. In both cases, only the `doWork` method has to be implemented. The

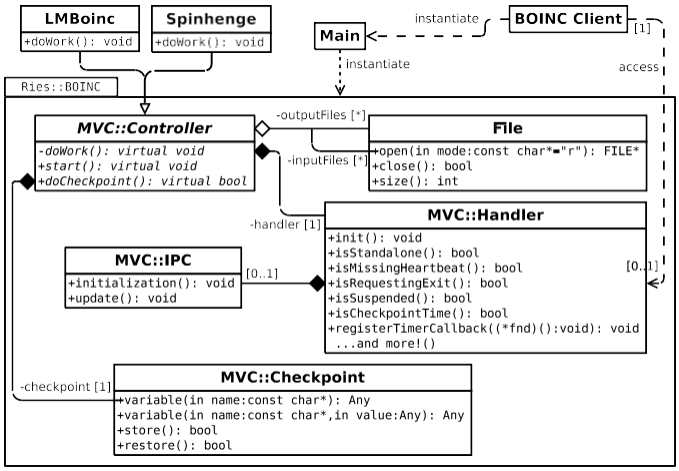

Figure 6. Object-orientated Abstraction of BOINC's API.

computation is called within *Main* and responsible for the instantiation of the BOINC framework and the OOP layers. *Main* is instantiated by BOINC's client and has access to the `MVC::Handler` which provides information about the computation state.

A. State Machine of a Scientific Application

The fundamental idea of the state machine for the SAPP is to reduce the complexity based on the fact that only relevant functions are used for creating various state nodes, e.g. a state "Initialization" is used for initializing BOINC's runtime environment or "Computing" is used for processing the core computation; thus, it will call `doWork()`. SAPP's state machine possesses three Regions:

- **R1** Is used for the handling of asynchronous-messages,
- **R2** is used for the checkpointing process, and
- **R2** specifies the computation of the SAPP.

The Transitions of the processing states and the three regions can be triggered by external events and are able to call actions when they are used.

B. Mapping of Activities and Actions

Most of UML4BOINC's stereotypes can be mapped directly to BOINC's API-calls or with less code-complexity. The current section illustrates how the CG from UML to implementation code can be done.

1) «Atomic»: Atomic functions cannot be interrupted and have to be finished before subsequent actions or activities are executed. Listing 5 shows a small code-snippet. Line 2 opens and line 5 closes the atomic area; thus, any action or activity is called and cannot be interrupted between these calls. Fig. 7 shows a UML model consisting of an atomic area. It executes an activity with an initialisation and final node and then executes the "transferMoney()" action. Exceptions cannot interrupt the execution.

```
/// Start: <<Atomic>>
boinc_begin_critical_section();
  transferMoney(); /*...*/
boinc_end_critical_section();
/// End: <<Atomic>>
```

Listing 5. Code-implementation example of BOINC's atomic mechanism.

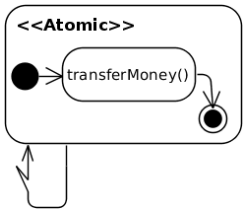

Figure 7. Code-mapping for UML4BOINC's «Atomic».

The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.28 e-ISSN: 2251-7545

2) «Action» & type=#FractionDone: This type of «Action» is used in order to inform the BOINC client how much work has been accomplished. It is a value between zero and one, i.e. a percentage descriptor. The mapping of «Action» in BOINC's API is a single line: boinc_fraction_done (fdone). *fdone* can be an input pin and requires a floating-point value (PrimitiveType::Real [22]) which has to be calculated by the scientific application developer.

3) «Action» & type=#FileInfo: It enables querying information of specific files. Listing 6 implements «FileInformation» directly as a C++-class, namely tag-values as attributes and the operations as methods. «Action» is mapped directly: (a) the input pin parameter *path* is used as `query`'s parameter, and (b) the output pin *finfo* as a new variable which assigns the returning value of `query`.

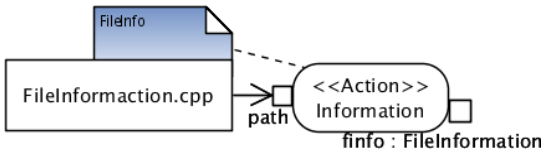

Figure 8. «Action» to query file information.

```
class FileInformation {
public:
 enum FileKind {Unknown=0, File=1,
                Directory=2, Symlink=3};
 static FileKind type;
 static std::string cwd, absolutePath;
 static double filesize, dirsize;
 static double totalSpace, freeSpace;

 static FileInformation query(const char *path) {
  FileInformation fi;
  if(is_file(path))      type=FileKind::File;
  else if(is_dir(path)) type=FileKind::Directory;
  else if(is_symlink(path))
                        type=FileKind::Symlink;
  else                  type=Unknown;
  char buffer[4096] = {'\0'};
  boinc_getcwd(buffer); cwd = buffer;
  relative_to_absolute(path, buffer);
  absolutePath = buffer;
  file_size(path, filesize);
  dir_size(path, dirsize);
  get_filesystem_info(totalSpace, freeSpace);
  return fi;

 }
};
/* ...initialise default values! */
int main(int argc, char **argv) {
 FileInformation finfo =
    FileInformation::query("FileInformation.cpp");
 /*...*/
}
```

Listing 6. «FileInformation» and «Action» for file information queries.

4) «Action» & type=#Locking: BOINC provides a functionality of preventing access to specific files, i.e. the BOINC-structure `FILE_LOCK` is used for locking and unlocking a file. Every file needs its own locking instance. Fig. 9 shows «Action» with one input pin to lock/unlock the file "FileInformation.cpp". The state of the lock/unlock-call is stored in the output pin *state* after the execution. When something goes wrong, an (exception named) *ExceptionLocking* which can be caught and handled individually, will be raised. Every locking mechanism is only usable in its context. When a global lock has to be used for specific files, it is necessary to define its lock instance in a global way.

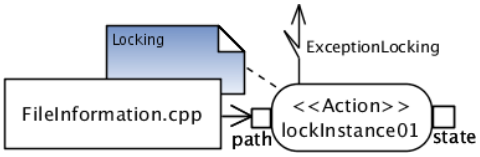

Figure 9. «Action» for locking/unlocking files.

5) «Action» & type=#FileHandling: Fig. 10 and Fig. 11 show how UML modelling can be realised when UML4BOINC's «Action» and *type=#FileHandling* are used. «Action»'s characteristic is specified by the input pin *mode*; it states how many input pins can exist in one action. The notes in Fig. 10 and Fig. 11 describe the modes with one and two additional input pins. These actions and individual characteristics can be directly transformed into C++-implementations. Table II shows the related mapping of *mode*'s value to the suitable BOINC API-call. This is a trivial approach and some modes can need more specific implementation, e.g. the mode `Open` can be used for virtual filenames or/and for filenames with immediate access. In case the filename is a virtual name, it is necessary to resolve the physical filename. Listing 7 shows a general approach for file opening. Various problems can arise when the file is in a specific file format, e.g. it is a ZIP-archive and the opening differs from mentioned approaches. Opening approaches can be handled directly by the CG or implemented in the OOP abstraction layer. In this regard, CG is mostly a tool-specific task and can be handled in different ways.

```
std::string inputDataIn;
boinc_resolve_filename_s(path, inputDataIn);
int res = boinc_fopen(inputDataIn, filemode);
```

Listing 7. General code-implementation for file opening.

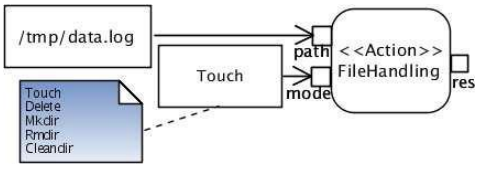

Figure 10. «Action» for file handling with two parameters.

6) «Action» & type=#TrickleDown: Asynchronous-messages are not handled immediately. The SAPP is responsible for collecting all messages. The collection

The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.28 e-ISSN: 2251-7545

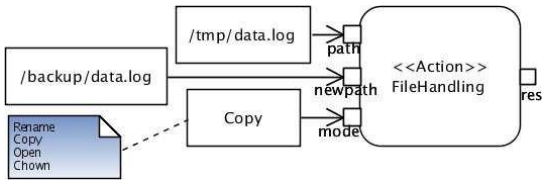

Figure 11. «Action» for file handling with three parameters.

Mode	BOINC's Function Call
one additional input pin	
Touch	int res = boinc_touch_file (path);
Delete	int res = boinc_delete_file (path);
Mkdir	int res = boinc_mkdir(path);
Rmdir	int res = boinc_rmdir(path);
Cleandir	int res = clean_out_dir (path);
two additional input pins	
Copy	int res = boinc_copy(path, newpath);
Chown	int res = boinc_chown(path, owner);
Rename	int res = boinc_rename(path, newpath);
Open	FILE *res = boinc_fopen(path, filemode);

Table II
MAPPING OF UML TO BOINC'S API.

process can be done in the first region of the state machine. The messages are provided by the output pin of «Action». Listing 8 shows the receiving of trickle-messages and a suggestion for handling them.

```
char trickleFilename[32] = {'\0'};
int retval = boinc_receive_trickle_down(
    trickleFilename, 32);
FILE *trickleFile = boinc_fopen(trickleFilename, "
    r");
if(trickleFile) {
 /// (1) read in the trickle-message content
 /// (2) parse the content
 /// (3) create a specific TrickleMessage instance
 ///     and fill this instance with content of
     the message

}
```

Listing 8. Approach of how the trickle-message handlers can be implemented on the client-side.

7) «TrickleUp»: Fig. 12 shows UML4BOINC's stereotype «TrickleUp» for sending messages from every host to a BOINC project. Generally, this stereotype can only be used by clients. Listing 9 shows an example of how the stereotype can be implemented. A vector container for storing string values is defined and provides the planned content of the message which is created in lines 5-10. The sending itself is done in line 11. The stereotype does not provide an output pin for handling the return value of the sending function. Thus, it is checked in order to enable error treatment. However, the reception of messages and their handling on the server-side require more work. Thus, the content of the message must be parsed and handled individually.

```
std::vector<std::string> messageFields;
/* ...fill messageFields with data! */
if(messageFields.size() >= 3) {
```

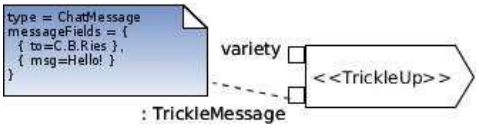

Figure 12. Code-mapping for UML4BOINC's «TrickleUp».

```
 char msgUp[1024] = {'\0'};
 snprintf(msgUp, 1024,
  "<to>%s</to>\n<from>%s</from>\n<msg>%s</msg>\n",
  messageFields[0],
  messageFields[1],
  messageFields[2]
}
int ret = boinc_send_trickle_up(
              (char*)variety, msgUp);
if(ret) { /* Error-handling, ... */ }
```

Listing 9. Trickle-message handling on the client-side.

Listing 10 shows how the set-up for activating message handling has to be done by programming when on client-side. The handling can be activated during the initialisation of the SAPP, i.e. in the state "Initialization". Section X describes the server-side.

```
int main(int argc, char **argv) {
 BOINC_OPTIONS options;
 options.handle_trickle_ups = true;
 options.handle_trickle_downs = true;
 int returnValue = boinc_init_options(&options);
 ...
}
```

Listing 10. Enabling of BOINC's asynchronous-messages on clients.

8) «Timing»: Periodically executed function-calls can be implemented in a SAPP, e.g. to calculate an average value or to handle input-/output calls. BOINC provides a mechanism which enables adding of specific functions. Accordingly, a function can be used as callback-function by BOINC's internal. UML4BOINC specifies «Timing» in order to add several different function calls as shown in Fig. 13. Only «Timing» has to be added to BOINC's callback mechanism. Listing 11 shows how the generated code can look like. The first three lines are dummy implementations and can be replaced by more complex action flows. The last execution is compared to its individual delay value (Line 5) by `timingFunction`. When the current time possesses less than the last `timingMoments` value plus the `timingDelays` value, then the action is executed. This implementation approach does not allow executing the three functions in an orthogonal way. Finally, the `timingFunction` has to be registered for execution within BOINC's runtime environment and this is done in line 29.

```
void Checkpointing()     { /* ... */ }
void calculateAverage()  { /* ... */ }
void calculateMedia()    { /* ... */ }

int timingDelays[]   { 15, 5, 30 };
int timingMoments[] {  0, 0,  0 };
```

The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.28 e-ISSN: 2251-7545

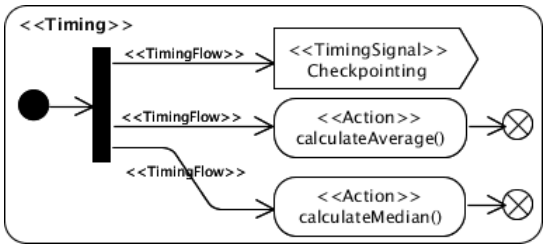

Figure 13. Code-mapping for UML4BOINC's «Timing».

```
FUNC_PTR timingFunctions[] {
 Checkpointing, calculateAverage, calculateMedia
};

void timingFunction() {
 static bool firstRun = true;
 if(firstRun) {
  for(int i=0; i<3; i++)
   timingMoments[i] = time(NULL);
  firstRun = false;
 }
 for(int i=0; i<3; i++) {
 int timeEdge = timingMoments[i]+timingDelays[i];
  if(timeEdge > time(NULL))
   continue;
   (timingFunctions[i])();
   timingMoments[i] = time(NULL);
 }
}

int main(int argc, char **argv) {
 boinc_init();
 boinc_register_timer_callback(timingFunction);

 // Keeps the application running...
 for(int i=0; i<600; i++) sleep(1);
}
```

Listing 11. «Timing» & «TimingFlow» and their embedded actions.

9) «WaitForNetwork»: Fig. 14 shows the two outgoing transitions. Listing 12 shows an approach of how «WaitForNetwork» can be implemented. The structural feature is directly created in lines 2-7. The true-guarded transition is implemented by lines 12-13, and the false-guarded by lines 15-16.

```
/// Start: <<WaitForNetwork>>
#define TRIES 10
boinc_need_network();
bool netavailable = false;
for(int _try=0; _try < TRIES; _try++) {
 netavailable = (boinc_network_poll()?true:false);
}
/// End: <<WaitForNetwork>>

if(netavailable) {
 /// Transition: [true]
 std::cout << " NETWORK" << std::endl;
} else {
 /// Transition: [false]
 std::cout << " NO NETWORK" << std::endl;
}
```

Listing 12. «WaitForNetwork» to query network connectivity.

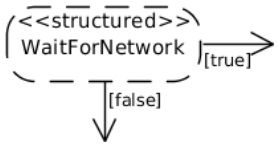

Figure 14. «WaitForNetwork» to query network connectivity.

VI. SERVICES

Services are only present on the server-side of a BOINC project; it needs information about the individually services: firstly, the location of the executability, its parameters for starting, and whether it is enabled or disabled; secondly, the application itself has to be implemented or a pre-implemented version has to be installed. The services are modelled within the (a) Infrastructure and (b) Application diagrams; in case the service is a task the (c) Timing diagram is also relevant. This part is covered by additional work [16].

VII. WORK

BOINC's work processing is based on a small state-chain, namely *start* → *creation* → *processing* → *validation* → *assimilation* → *end* [15]. Fig. 15 shows how series and workunits can be modelled by a visual approach provided by Visu@lGrid [12], [17]. Different series can be created within Work branch **(1)**. Every series contains a direct description of the used workunit, the input and output files. Any file can have individual datafields to set-up the runtime parameter for the computation. The visualisation **(2)** shows the series with input/output files. *images.zip* is a physical file which can be added manually or dynamically in this case and by the use of «InterfaceDataSource». **(3)** and **(4)** show how the relation between the first four series and a fifth serie can be described. A detailed explanation can be found in additional work [15].

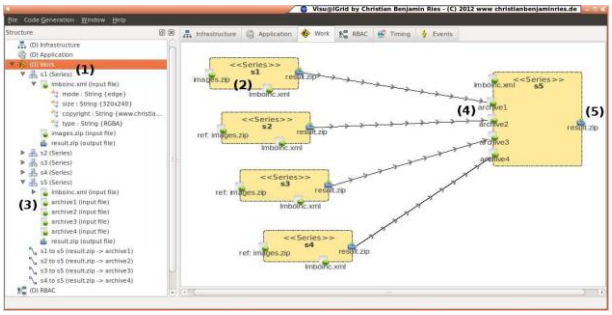

Figure 15. Visual modelling for the specification of BOINC workunits; The Series and the relation among themselbes can be modelled with associations between input and output files.

VIII. WORK VALIDATION

Workunit specifications state on the one hand how output files have to be opened and on the other hand how the «InterfaceValidate» can be used. It provides access to

The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.28 e-ISSN: 2251-7545

the output files for validation processes. Thus, the interface «InterfaceDataset» provides an universal way to open these output files. BOINC's validation framework exists on a high-level and low-level abstraction [14, Section 9.3]. This paper covers only the high-level abstraction which is based on three steps: (i) *initialisation*, (ii) *comparing*, and (iii) *cleanup*. Only the second step has to be filled with functionality for a successful validation. As shown in Listing 13, the result has to be stored in the fifth parameter: (a) *false* or (b) *true*, i.e. the computational result is *not valid* or *is valid*. The first & second and third & fourth parameters are used to handle the result files and result raw datasets. Fig. 16 shows how the Listing 13 can be modelled in UML by use of a UML Activity. The parameters of the function `compare_results` are modelled as UML ParameterNodes. The comparing statement in line 4 is transformed into a UML DecisionNode and two outgoing transitions with equipment guards. These guards are the real validation parts and decide whether the computational results are valid or not, i.e. the output UML ParameterNode is valued with *false* or *true*. This approach can be added to the UML StateMachine and extended by «Validation».

```
int compare_results(RESULT & r1, void *data1,
                    RESULT & r2, void *data2,
                    bool & match) {
 match = (r1.cpu_time >= 10 && r2.cpu_time >= 10);
 return 0;
}
```

Listing 13. Part of BOINC's high-level validation framework.

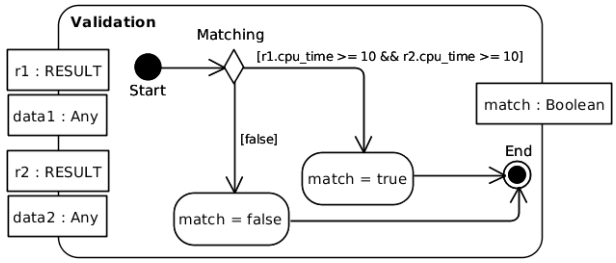

Figure 16. Part of BOINC's high-level validation framework in UML.

IX. Work Assimilation

The «InterfaceAssimilate» can be used for the assimilation of computational results. It provides access to output files and then enables interface «InterfaceDataset» to open the output files. BOINC's assimilation framework provides a function which has to be filled with assimilation routines, i.e. by use of the standard implementation or an own specification and how output files and results have to be stored. Generally, BOINC's framework provides two targets for storing: (i) storing in a specific file-directory and (ii) storing in a database. The destination of storing is specified within the CG process and by the Infrastructure diagram. The infrastructure specification contains an «Assimilator» which can be extended by ports in order to use file-directories or database tables for storing.

X. Asynchronous Messages

This section gives an example of how asynchronous-messages can be transformed into code and by use of the relevant UML4BOINC stereotypes on server-side. When asynchronous-messages are used on both sides, the message handling must be enabled in any case on server- and client-side. On the server-side a specific XML configuration has to be set.

A. Server-Side Handling

BOINC provides a default implementation for the handling of asynchronous-messages (AMs), i.e. a kind of ping-pong system. This examplary application iterates over a BOINC database table and queries unhandled AMs. The queried messages are processed by a specific function and therefore named `handled_trickle(DB_MSG_FROM_HOST&)`. The parameter is the database entry and contains information about the sender and the message send which is text-based. Fig. 17 shows how an AM is specified by UML4BOINC. «TrickleMessage» specifies a chat message and contains one message value which is defined by three additional datafields: (i) *to*, (ii) *from*, and (iii) *msg*. This chat message has one receiver: *C.B.Ries*. The specification is not always available during runtime. Chat messages are transmitted primarily from users; the specification does not possess any association to receivers at this moment, only textual information about the planned receivers. The association is created on server-side when message receivers are queried from the user database of a BOINC project. Fig. 18 shows this detail through another viewpoint. One user transmits the «TrickleMessage» *1: chatMessage01* to the BOINC project. The message contains only a description of the targeting message's receivers. The «TrickleMessage» *1.1: chatMessage02* has a different format. *messageFields* has modified the information into an absolute information of the targeting receiver. The replacement of the targeting

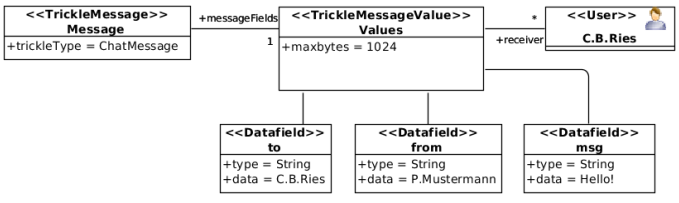

Figure 17. Instantiation of the stereotypes for asynchronous-messages.

receiver information needs to be handled on the server-side. Additional UML Actions can be specified for this case; it is not done in this research. Listing 14 shows the pseudocode of the relaying mechanism for chat messages on the server-side. Information of the received message is filtered: (i) the sender host, (ii) the sending user, and (iii) the names of

The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.28 e-ISSN: 2251-7545

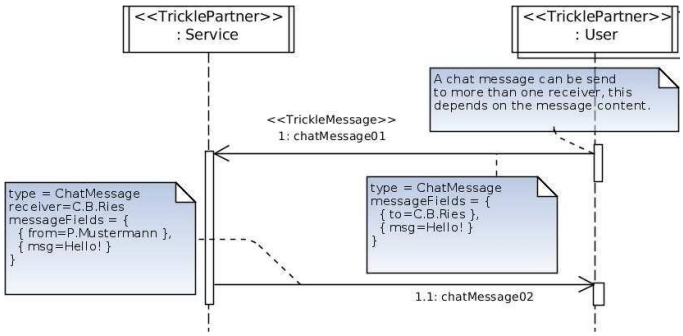

Figure 18. Asynchronous-messages within a BOINC project.

the target users. The received message is parsed into the format for the targeting users in the next step. Thus, the message is send to all hosts of an user and accordingly to every workunit. As a matter of fact, BOINC's AMs need the information about the workunit and which message has to be handled. When a message shall target a specific user, the sender can not forsee when and where it will be read by the receiver. According to that, the server sends it to all workunits.

```
handle_trickle(MSG_FROM_HOST& mfh) {
 hostFrom =  HOST(mfh)
 userFrom =  USER(mfh)
 usersTo  = USERS(mfh)

 m = MAP_RECEIVED_MSG_TO_SEND_MSG(mfh)

 do user = Send_Message_to_Users(usersTo)
  do workunit = Send_Message_to_Workunit(user)
   SEND_MESSAGE(workunit, m)
  done
 done
}
```

Listing 14. Pseudocode for the trasnfer of chat messages between users.

Listing 14 shows how little UML Action is needed in order to implement the pseudocode. It requires five actions and additional structural activities, i.e. a for-loop or do- while loop can be used to iterate over all datasets. Fig. 19 shows how this can be realised through UML. The received message is specified as a UML ObjectNode and used as input value for the three UML Actions: (i) `HOST`, (ii) `USER` and (iii) `USERS`. The first two are not used in this example. The third action is used in order to filter all users and query the information from the database. The user information is parsed to the first iterative element which consequently iterates over all users. The individual users are parsed to the second iterative; it enables an iteration over all workunits of the specific users. Finally, a pre-parsed message is send to a specific user's workunit. The handling of AMs is less complex on the client- than on the server-side. The client only receives and transmits messages from and to a BOINC project. It must not query any other information from a database. The Transmission of AMs is provided by «TrickleUp» and the reception is supplied by «Action».

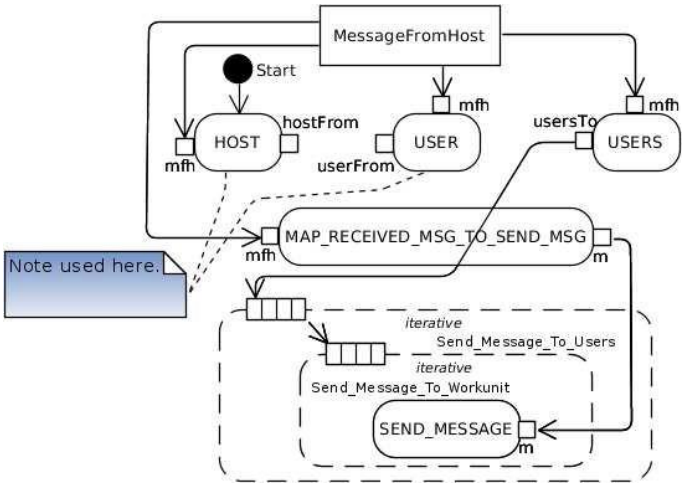

Figure 19. UML modelling for asynchronous-messages on the server-side.

XI. Conclusion

This paper presented a CG approach for the transformation of UML4BOINC's stereotypes into an implementation. At first, different CG approaches have been introduced followed by examples of language recognition with ANTLR. Subsequently, the direct transformation of models into executable code demonstrates how an AST can be used for CG. In addition, it has been exemplified how a semantic model can be generated automatically through a specific DSML. Accordingly a DSML named RSM has been introduced; it enables the creation of a state machine which is based on a scientific application. Secondly, an object-orientated abstraction of BOINC's API has been presented briefly. However, the CG transformation is not based on strict rules. Developers can decide how they implement its BOINC parts and scientific applications.

References

[1] D.P.Anderson, "BOINC: A system for public-resource computing and storage," in *Grid Computing 2004 Proceedings Fifth IEEEACM International Workshop* (R. Buyya, ed.), pp. 4-10, IEEE Computer Soc, 2004.

[2] A. H. Dönni, "The Boost Statechart Library," http://www.boost.org/doc/libs/1_46_0/libs/statechart/doc/-index.html [Online. Last accessed: 10th December 2012], 2007.

[3] C. J. Henry, "Meta State Machine (MSM)," http://www.boost.org/doc/libs/1_49_0/libs/msm/doc/HTML/-index.html [Online. Last accessed: 23rd November 2012], 2010.

[4] M. Feikas, "How to represent Models, Languages and Transformations," *System*, 2006.

[5] M. Fowler, *Domain-Specific Languages*, vol. 5658 of Lecture Notes in Computer Science. Addison-Wesley Professional, 2010.

Doi: 10.7321/jscse.v3.n3.28 e-ISSN: 2251-7545

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissies, *Design Patterns*, vol. 47 of *Addison Wesley Professional Computing Series*. Addison Wesley, 1995.

[7] "Information technology – Syntactic metalanguage – Extended BNF." 1996

[8] T. A. Morgensen, "Basics of Compiler Design Extended edition," *Analysis*, 2008

[9] I. A. Niaz, *Automatic Code Generation From UML Class and Statechart Diagrams. Dissertation*, University of Tsukuba, 2005

[10] T. Parr, "Translators should use tree grammars," tech rep., University of San Francisco, San Francisco, CA, 2004.

[11] T. Parr, *The Definitive ANTLR Reference – Building Domain-Specific Languages*. Pragmatic Programmers, ISBN: 978-0978739256, 2007.

[12] C. B. Ries, "Visu@lGrid – Integrated Development Environment for BOINC," http://visualgrid.sourceforge.net [Online. Last accessed: 2012]

[13] C. B. Ries, "libries – Research library for Visu@lGrid," http://libries.sourceforge.net [Online. Last accessed: 2012]

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

[15] C. B. Ries, C. Schröder, and V.Grout. "Model-based Gener- ation of Workunits, Computation Sequences, Series and Ser- vice Interfaces for BOINC based Projects," in *Proc. SERP'12 (Worldcomp)*, Las Vegas (NV) 2012

[16] 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

[17] 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

[18] C. B. Ries and C. Schröder. "Public Resource Computing mit Boinc." Linux-Magazin, vol. 3, pp. 106-110, March 2011. Internet: lmboinc.sourceforge.net

[19] 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

[20] C. B. Ries, "UML 2 Statemachine for C++," https://sourceforge.net/projects/uml2stm4cpp/ [Online. Last accessed: 2013]

[21] "Spinhenge@home." http://spin.fh-bielefeld.de [Online. Last accessed: 15th December 2012]

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