Attila Csaba MAROSI

Önkéntes Számítási Rendszerek
Kihívásai és Formális Aspektusai

Challenges and Formal Aspects of
Volunteer Computing

Ph.D. Thesis

Thesis Advisors:
Péter KACSUK
MTA SZTAKI, LPDS
Sándor JUHÁSZ
BME, AUT

Budapest, 2016.
‘The important thing is not to stop questioning. Curiosity has its own reason for existing. One cannot help but be in awe when he contemplates the mysteries of eternity, of life, of the marvelous structure of reality. It is enough if one tries merely to comprehend a little of this mystery every day. Never lose a holy curiosity.’

-- Albert Einstein
Volunteer Computing (VC) and Desktop Grid (DG) systems collect and make available the donated resources from non-dedicated computers like office and home desktops. VC systems are usually deployed to solve a grand compute intensive problem by researchers who either don’t have access to or don’t have the resources to buy a dedicated infrastructure; or simply don’t want to maintain such an infrastructure. VC and DG paradigms seem similar, however they target different use cases and environments: DG systems operate within the boundaries of institutes, while VC systems collect resources from the publicly accessible internet. Evidently VC resembles DGs whereas DGs are not fully equivalent to VC. Contrary to “traditional grids” [1, 2] there is no formal definition for the relationship of DG and VC that could be used to categorize existing systems. There are informal attempts to categorize them and compare with grid systems [3, 4, 5]. There is also a formal model for VC, but it is based on an existing implementation [6].

In the first theorem group I evaluate DG/VC systems using formal methods: contrary to informal methods, that compare using selected characteristics, using formal methods I seize the semantics of the systems. I develop formal reference models and definitions, that make possible evaluating existing systems. Utilizing my formal models I determine the relationship of DG and VC, and their relationship with grid computing. Finally I create a formal reference model for an existing system and evaluate it against my previous DG/VC models. Finally I create a formal reference model for an existing system and evaluate it against my previous DG/VC models.

In the second half of this thesis I investigate three – what I consider – major challenges and future directions of VC: (i) workload sharing between DG/VC systems; Creating Volunteer Cloud Computing (VCC), in detail with overcoming the heterogeneity of resources and systems, and (iii) reducing the “tail-effect” for round-trip times of batches of jobs caused by the volatility of resources. I present my results in the second theorem group.

Multiple national and international R&D projects benefited from my results that I presented in this thesis. For example, based on the second theorem group I developed several open source frameworks. These were used to adapt – so far – 59 scientific applications to DG/VC systems from all over the world. Additionally implementations belonging to the methods and algorithms I developed are running in real-world environments and are part of open source systems (e.g., BOINC).


Statement of authorship

Declaration

I, Attila Csaba Marosi, confirm that this doctoral thesis has been written solely by myself, and I have used only those sources that have been explicitly identified. Every part that has been adopted from external sources, either quoted or rephrased, has been unequivocally referenced, with proper attribution to the source.

Budapest, October 5, 2016

Nyilatkozat

Alulírott Marosi Attila Csaba kijelentem, hogy ezt a doktori értekezést magam készítettem, és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos tartalomban, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Budapest, 2016. október 5.

____________________________

Marosi Attila Csaba

1A bírálatok és a védésről készült jegyzőkönyv a későbbiekben a Dékáni Hivatalban elérhetőek.
Köszönetnyilvánítás

Számos embernek tartozom hálával e dolgozat létrejöttéért, néhányat szeretnék közülük kiemelni. Mindenekelőtt a családomat, akik mindvégig támogattak és mellettem álltak. Köszönöm szüleimnek, Gabinak és fiámnak, Ádámnak, a tőlük kapott szeretetet, a végével türelmet és kitartást mialatt eme hosszú, és többször rögös út végére próbáltam eljutni.

Hálás vagyok Kacsuk Péternek, aki megteremtette az anyagi és emberi feltételeket a kutatásonhoz.

Szintén köszönettel tartozom az Automatizálási és Alkalmazott Informatikai Tanszéknek, különösen Charaf Hassannak, Vajk Istvánnak és Juhász Sándornak.

Köszönet a MTA Számítástechnikai és Automatizálási Kutatóintézetének, és munkatársaimnak az ottani Párhuzamos és Elosztott Rendszerek Laboratóriumában: Németh Zsoltnak, az ASM modellezés kapcsán nyújtott felbecsülhetetlen segítségéért, valamint a disszertáció újabb és újabb változatainak rendszeresen véleményezéséért; Kovács Józsefné és Kecskeméti Gábornak a tőlük kapott tanácsokért és véleményezésekért; valamint Lovas Róbertnek a támogatásáért.

Nélkülük nem készülhetett volna el ez a dolgozat.
1 Introduction
1.1 Motivation and goal of this thesis .................................. 1
1.2 Structure ......................................................... 2

2 Formal models for desktop grids and volunteer computing
2.1 Characterization of desktop grids and volunteer computing ....... 6

2.1.1 Related work .................................................. 10

2.2 Modelling desktop grids ........................................... 12

2.2.1 Assumptions .................................................. 12

2.2.2 Universes and signatures ...................................... 14

2.2.3 Initial state .................................................... 18

2.2.4 Functionalities ................................................. 19

2.2.5 Job phase transition rules ..................................... 21

2.2.6 Correspondence of M\(_{GRID}\) and M\(_{GROUND-DG}\) .............. 25

2.2.6.1 Equivalence of user mapping service in M\(_{GRID}\) and M\(_{GROUND-DG}\) .... 25

2.2.6.2 Equivalence of resource mapping service in M\(_{GRID}\) and M\(_{GROUND-DG}\) .... 25

2.2.7 Definition for Desktop Grid .................................... 26

2.2.7.1 Discussion of existing middleware .......................... 27

2.3 Modelling volunteer computing ..................................... 29

2.3.1 Assumptions .................................................. 30

2.3.2 MVC\(_{-VOTE}\) : model for voting based result certification ....... 32

2.3.2.1 Universes and signatures .................................. 32

2.3.2.2 Initial state ................................................ 35

2.3.2.3 Functionalities ............................................. 36

2.3.2.4 Job phase transition rules ................................ 39

2.3.3 Correspondence of M\(_{GROUND-DG}\) and MVC\(_{-VOTE}\) ASM models .... 43

2.3.3.1 Notion of completeness and equivalence ................... 43

2.3.3.2 Proof of correspondence ................................ 45

2.3.4 MVC\(_{-SPOT}\) : model for spot-checking based result certification .... 52

2.3.4.1 Universes and signatures .................................. 52

2.3.4.2 Initial state ................................................ 52

2.3.4.3 Functionalities ............................................. 52

2.3.4.4 Job phase transition rules ................................ 53

2.3.4.5 Correspondence of M\(_{GROUND-DG}\) and MVC\(_{-SPOT}\) ....................... 53

2.3.5 Definition for volunteer computing ................................ 54

2.3.5.1 Discussion of existing middleware .......................... 54

2.4 A formal model of BOINC ........................................ 56
2.4.1 Assumptions ............................................. 56
2.4.2 Universes and signatures ................................. 58
2.4.3 Initial state ........................................... 62
2.4.4 Functionalities ........................................ 64
2.4.5 Job phase transition rules ............................... 67
2.4.6 Correspondence of M_{VC-VOTE} and M_{BOINC} ASM models ...... 74
2.4.7 Discussion ............................................ 75

3 Challenges and future directions of volunteer computing 77

3.1 Towards federating volunteer computing ................. 79
  3.1.1 Federation at a glance ................................ 79
    3.1.1.1 Introduction of scenarios ......................... 80
    3.1.1.2 Forming federations .............................. 82
  3.1.2 Hierarchy of desktop grids ............................. 84
    3.1.2.1 Extending M_{BOINC} for use in hierarchy ........... 85
3.2 Towards volunteer clouds .................................. 90
  3.2.1 Introduction ....................................... 91
  3.2.2 Definition for volunteer cloud ....................... 93
  3.2.3 Abstraction frameworks for volunteer computing .... 94
    3.2.3.1 Related work ..................................... 95
    3.2.3.2 Abstraction A1: Cross-middleware abstraction ...... 96
    3.2.3.3 Abstraction A2 and A3-1: Middleware and environment abstraction ......................... 99
    3.2.3.4 Abstraction A3-2: Full abstraction ................. 103
  3.2.4 Evaluation ......................................... 117
3.3 Improving batch completion times ........................ 118
  3.3.1 Related work ....................................... 119
  3.3.2 Improving completion time with on-demand reliable resources .. 121
    3.3.2.1 Job assignment ................................... 121
    3.3.2.2 Evaluation ....................................... 126
  3.3.3 Improving batch completion times with redundancy adaptation .. 131
    3.3.3.1 Evaluation ....................................... 132

4 Conclusions .............................................. 137
  4.1 Applicability of the results ............................ 137
  4.2 Future work ........................................... 138

A Appendix ............................................... 141
  A.1 Scientific applications using DC-API, GenWrapper or GBAC ...... 141
  A.2 Workload transfer for the Hierarchical Desktop Grid ............ 145
  A.3 Abstract State Machines ................................ 149
  A.4 ASM for grid computing ................................ 151
  A.5 GenWrapper examples .................................. 153
  A.6 DC-API use case: Java applications for BOINC .................. 154
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.7  Wikipedia pre-processing for Cross-Language Plagiarism Search</td>
<td>157</td>
</tr>
<tr>
<td>A.7.1 Workflow</td>
<td>158</td>
</tr>
<tr>
<td>A.7.2 KOPI Server and portal</td>
<td>159</td>
</tr>
<tr>
<td>A.7.3 SZTAKI Desktop Grid</td>
<td>160</td>
</tr>
<tr>
<td>A.7.4 Wikipedia pre-processing on SZTAKI Desktop Grid</td>
<td>161</td>
</tr>
<tr>
<td>A.8  Augmenting EDGeS@home with on-demand reliable resources</td>
<td>163</td>
</tr>
<tr>
<td>A.8.1 EDGeS@home volunteer desktop grid</td>
<td>164</td>
</tr>
<tr>
<td>A.8.2 Augmenting virtual organizations with desktop grids and volunteer computing</td>
<td>165</td>
</tr>
<tr>
<td>A.8.2.1 Augmenting with an existing volunteer computing project</td>
<td>166</td>
</tr>
<tr>
<td>A.8.2.2 Augmenting with a new desktop grid project</td>
<td>166</td>
</tr>
</tbody>
</table>

Bibliography

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>169</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Proposed formal models and their relations. ........................................... 3

2.1 Basic elements of the ASM model for Desktop Grids. ............................ 15

2.2 Job phase transition diagram with corresponding rules for the
MGROUND−DG model. ................................................................. 21

2.3 Result certification methods based on [70] ................................. 32

2.4 Basic elements of the MvC−VOTE ASM model. ................................. 33

2.5 Simplified job phase transition diagram for the MvC−VOTE model. .... 38

2.6 ASM refinement scheme, where \(\equiv\) denotes equivalence of states of interest
in corresponding states. .................................................................. 44

2.7 ASM refinement component diagrams: (m,n)-trapezoid; (0,n)-triangle; and
(m,0)-triangle ........................................................ .......................... 45

2.8 Computation segments SEG\_INITIALS (a.) and SEG\_INIT (b.) for normal
executions in MGROUND−DG and in MvC−VOTE with corresponding states. 48

2.9 SEG\_START computation segment for normal executions in
MGROUND−DG and in MvC−VOTE with corresponding states. .......... 48

2.10 SEG\_STOP computation segment for normal executions in
MGROUND−DG and in MvC−VOTE with corresponding states. .......... 49

2.11 Computation segment SEG\_EXEC for normal executions in
MGROUND−DG and in MvC−VOTE with corresponding states. .......... 49

2.12 Computation segment SEG\_TERM for normal executions in
MGROUND−DG and in MvC−VOTE with corresponding states. .......... 50

2.13 Job instance (and abstract job) failure in MGROUND−DG and in
MvC−VOTE with corresponding states. ........................................ 51

2.14 Abort in MGROUND−DG and abort request or failure for a job instance in
MvC−VOTE with corresponding states. ........................................ 51

2.15 On the right side the different nodes (host, manager, ui) used in
MvC−VOTE. On the left side their counterparts in M\_BOINC. .......... 59

2.16 Basic elements of the ASM model M\_BOINC. ................................. 62

2.17 Job phase transition diagram for the M\_BOINC model. Dashed arrows and
job phases with dashed frames represent the three states and their transitions
contracted into \(\text{inprogress}\) job phase via (m,0)-trapezoid refinement
(see Section 2.3.3.1 and Figure 2.7 for more details). ....................... 68

3.1 Possible architecture types for interconnecting volunteer computing systems 80

3.2 Proposed scenarios for desktop grid and volunteer computing federations . 81

3.3 The path of a job through nodes during its state transitions in the different
models: (a) MGROUND−DG; MvC−VOTE; MvC−SPOT and M\_BOINC; (b)
M\_FED−BOINC→M\_BOINC ......................................................... 84

3.4 Roles in the hierarchy of DG/VC projects ........................................... 85
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>The split architecture of the Gateway: it acts both as a Host and as a UI.</td>
<td>85</td>
</tr>
<tr>
<td>3.6</td>
<td>Cloud computing service models and their relationship.</td>
<td>93</td>
</tr>
<tr>
<td>3.7</td>
<td>Taxonomy of frameworks for application adaptation and development</td>
<td>96</td>
</tr>
<tr>
<td>3.8</td>
<td>Abstraction frameworks for DG/VC systems: A1 (Cross-middleware), A2 (Middleware), A3-1 (Environment) and A3-2 (Full)</td>
<td>96</td>
</tr>
<tr>
<td>3.9</td>
<td>DC-API application components</td>
<td>98</td>
</tr>
<tr>
<td>3.10</td>
<td>Legacy application using (a) the BOINC wrapper; and (b) GenWrapper</td>
<td>100</td>
</tr>
<tr>
<td>3.11</td>
<td>Architecture of the sandbox</td>
<td>108</td>
</tr>
<tr>
<td>3.12</td>
<td>Run-time (real, in seconds) of the test work unit</td>
<td>110</td>
</tr>
<tr>
<td>3.13</td>
<td>GBAC: application, inputs and outputs</td>
<td>112</td>
</tr>
<tr>
<td>3.14</td>
<td>GBAC: server side related parts</td>
<td>114</td>
</tr>
<tr>
<td>3.15</td>
<td>GBAC: execution of a sample task</td>
<td>115</td>
</tr>
<tr>
<td>3.16</td>
<td>A virtual appliance for GBAC is composed of a base image, an optional application overlay image and an instance image</td>
<td>117</td>
</tr>
<tr>
<td>3.17</td>
<td>Task reassignment in the non-interactive QoS process in EDGI for EDGeS@home</td>
<td>127</td>
</tr>
<tr>
<td>3.18</td>
<td>PDF of the running time of the Autodock application in the non-augmented desktop grid</td>
<td>128</td>
</tr>
<tr>
<td>3.19</td>
<td>Performance comparisons of the augmented and non-augmented desktop grids</td>
<td>128</td>
</tr>
<tr>
<td>3.20</td>
<td>Execution of cloud augmented batches 1,3,4 and 5 and non-augmented batches 3 and 5.</td>
<td>130</td>
</tr>
<tr>
<td>3.21</td>
<td>Typical time lapse for a batch (a,c,e) and its jobs in round trip time order (b,d,f) for an English, French and German Wikipedia dataset</td>
<td>135</td>
</tr>
<tr>
<td>A.1</td>
<td>Flow of the workload transfer in a block: steps 1-6.</td>
<td>146</td>
</tr>
<tr>
<td>A.2</td>
<td>Flow of the workload transfer in a block: steps 7-12.</td>
<td>146</td>
</tr>
<tr>
<td>A.3</td>
<td>Flow of the workload transfer in a block: steps 13-21.</td>
<td>147</td>
</tr>
<tr>
<td>A.4</td>
<td>Sample GenWrapper work unit shell script</td>
<td>153</td>
</tr>
<tr>
<td>A.5</td>
<td>Part of a more complex GenWrapper shell script used by CancerGrid</td>
<td>153</td>
</tr>
<tr>
<td>A.6</td>
<td>A DC-API Java application on BOINC</td>
<td>154</td>
</tr>
<tr>
<td>A.7</td>
<td>Structure of the DC-API enabled Java application</td>
<td>156</td>
</tr>
<tr>
<td>A.8</td>
<td>The two main parts of the KOPI workflow: (i) the preprocessing of Wikipedia data on SZTAKI Desktop Grid, and (ii) the plagiarism search at the KOPI Portal</td>
<td>158</td>
</tr>
<tr>
<td>A.9</td>
<td>The architecture used for Wikipedia data preprocessing</td>
<td>159</td>
</tr>
<tr>
<td>A.10</td>
<td>The interface of the KOPI Portal</td>
<td>160</td>
</tr>
<tr>
<td>A.11</td>
<td>EDGI Infrastructure to bridge gLite jobs to BOINC DGs</td>
<td>165</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Characteristics comparison: <em>Main aspect</em></td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Characteristics comparison: <em>Voluntarism aspect</em></td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Characteristics comparison: <em>Resources aspect</em></td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>Characteristics comparison: <em>Applications aspect</em></td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>Characteristics comparison: <em>Administration aspect</em></td>
<td>9</td>
</tr>
<tr>
<td>2.6</td>
<td>Characteristics Comparison: <em>Security aspect</em></td>
<td>10</td>
</tr>
<tr>
<td>2.7</td>
<td>Basic elements of the ASM model: on the left side the base Grid ASM model</td>
<td>14</td>
</tr>
<tr>
<td>2.8</td>
<td>Basic elements of the ASM model: on the left side the M&lt;sub&gt;GROUND–DG&lt;/sub&gt;</td>
<td>32</td>
</tr>
<tr>
<td>2.9</td>
<td>Components of BOINC on the server side. Components denoted with &quot;*&quot; are</td>
<td>57</td>
</tr>
<tr>
<td>2.10</td>
<td>Basic elements of the ASM model: on the left side the M&lt;sub&gt;VC–VOTE&lt;/sub&gt;</td>
<td>59</td>
</tr>
<tr>
<td>2.11</td>
<td>Values of different state variables of a workunit for different job phases.</td>
<td>62</td>
</tr>
<tr>
<td>2.12</td>
<td>Values of different state variables of a result for different job phases.</td>
<td>63</td>
</tr>
<tr>
<td>3.1</td>
<td>Summary of frameworks and their supported abstractions</td>
<td>97</td>
</tr>
<tr>
<td>3.2</td>
<td>Comparison of major hypervisors for desktops and emulation software</td>
<td>105</td>
</tr>
<tr>
<td>3.3</td>
<td>Capabilities and requirements of different tools supporting application</td>
<td>118</td>
</tr>
<tr>
<td>3.4</td>
<td>Cloud (IaaS) resource providers and number of resources for EDGeS@home</td>
<td>126</td>
</tr>
<tr>
<td>3.5</td>
<td>Statistics for round trip times of batches of jobs from Wikipedia dumps</td>
<td>133</td>
</tr>
<tr>
<td>A.1</td>
<td>Applications 1-11. using GBAC, GenWrapper (GW) and DC-API</td>
<td>141</td>
</tr>
<tr>
<td>A.2</td>
<td>Applications 12-28. using GBAC, GenWrapper (GW) and DC-API</td>
<td>142</td>
</tr>
<tr>
<td>A.3</td>
<td>Applications 29-43. using GBAC, GenWrapper (GW) and DC-API</td>
<td>143</td>
</tr>
<tr>
<td>A.4</td>
<td>Applications 44-59. using GBAC, GenWrapper (GW) and DC-API</td>
<td>144</td>
</tr>
</tbody>
</table>
Nomenclature

API  
Application Programming Interface

ASM  
Abstract State Machines

BE-DCI  
Best-Effort Distributed Computing Infrastructure

BOINC  
Berkeley Open Infrastructure for Network Computing

CDS  
Conventional Distributed System

CDS  
Conventional Distributed Systems

CPU  
Central Processing Unit

DC-API  
Distributed Computing Application Programming Interface

DCI  
Distributed Computing Infrastructure

DCS  
Distributed Computing System

DG  
Desktop Grid

DGVCS  
Desktop Grids and Volunteer Computing Systems

EDGeS  
Enabling Desktop Grids for e-Science

EDGI  
European Desktop Grid Initiative

GBAC  
Generic BOINC Application Client

GC  
Global Computing

HPC  
High Performance Computing

HTC  
High Throughput Computing

KVM  
Kernel Virtual Machine

MPI  
Message Passing Interface

MTC  
Many-Task Computing

NAT  
Network Address Translation

NIST  
National Institute of Standards and Technology

PS  
Parameter Sweep

SZDG  
SZTAKI Desktop Grid
List of Tables

UPNP  Universal Plug and Play
VC    Volunteer Computing
VCC   Volunter Cloud Computing
VCS   Volunteer Computing System
VM    Virtual Machine
VMI   Virtual Machine Image
VMM   Virtual Machine Monitor
VO    Virtual Organization
XW-HEP XtremWeb for High Energy Physics
Chapter 1

Introduction

Contents

1.1 Motivation and goal of this thesis .......................... 1
1.2 Structure ................................................. 2

1.1 Motivation and goal of this thesis

Compute capacity is either used or wasted. Unused capacity cannot be stored for later use. The term cycle-scavenging refers to (distributed) systems that utilize this idle computing cycles of connected computers. Desktop Grids (DGs) form a subset of the Best Effort - Distributed Computing Infrastructures (BE-DCIs). BE-DCIs – as the name suggests – offer no guarantees for service quality, they often utilize volatile, heterogeneous, non-dedicated resources for solving compute intensive tasks. DGs cycle-scavenge institutional desktop computers to solve compute intensive problems. On the other hand Volunteer Computing (VC) utilizes the idle resources (i.e., CPU cycles and storage) of private donated computers. These are typically home desktop computers behind firewalls and routers that can be considered even more heterogeneous and volatile. The inaccessibility and unavailability characteristics make running traditional HPC (High Performance Computing) workloads like tightly-coupled MPI (Message Passing Interface) and most HTC (High Throughput Computing) applications on volunteer computing unfeasible. Contrary to these MTC (Many Task Computing), especially embarrassingly parallel or parameter study [7, Section 1.4.4] problems where there is no dependency/communication between jobs are well suited. In this case the same computation is performed using different input parameters that are usually achieved by partitioning a large parameter space. Evidently VC resembles DGs whereas DGs are not fully equivalent to VC. However due to their similarities, they are often used as synonyms or in an interchanged meaning. I make an attempt in this thesis to formalize them and their relationship. Formal modeling is applied that tries to grasp the semantic of their functionalities (opposed to comparisons based on properties or features). Further I use this modeling to formalize the Berkeley Open Infrastructure for Network Computing (BOINC) [8] VC system. The resulting models help categorizing existing DG/VC systems and aim to be a foundation for formalizing other middleware.

The second half of this thesis is dedicated to three – what I consider – major challenges and future directions of VC. First I propose a method that allows the collaboration (federation) of VC and DG projects.
Cloud Computing [9, 10] (CC) offers simple and cost effective outsourcing in dynamic service environments and allows the construction of service-based applications extensible with the latest achievements of diverse research areas, such as Grid Computing, Service-oriented computing, business processes and virtualization. CC is built using dedicated and reliable resources and provides uniform seemingly unlimited capacities. Volunteer Computing on the other hand uses volatile, heterogeneous and unreliable resources. As the second challenge (of VC in this thesis) I make an attempt starting from a definition for Cloud Computing to identify the required steps and formulate a definition for what can be considered as the next evolutionary stage for Volunteer Computing: Volunteer Clouds. Among the many idiosyncrasies of VC to overcome (e.g., volatility, heterogeneity, reliability, responsiveness, scalability, etc.). Heterogeneity exists in VC at different levels. Resources can have a diverse set of hardware components (e.g., CPU type, features and speed; available memory, bandwidth and disk, etc.) and second they run a diverse set of software components (e.g., operating system, installed libraries, etc.). VC/DG middlewares can have their own set of requirements for applications in order to be compatible with them. Applications on their own have requirements (e.g., depend on specific software libraries and packages). The vision of Cloud Computing promises to provide a homogeneous environment. My goal is to identify methods and propose solutions that tackle the heterogeneities and thus, make a step towards Volunteer Clouds. (In this thesis the terms Volunteer Clouds and Volunteer Cloud Computing are used interchangeably.)

As mentioned previously VC is suited mainly for embarrassingly parallel or parameter study applications. These are usually run in form of batches where each batch represents an experiment. The total execution time of the batch from the submission until the finish of the last job is referred as makespan. The makespan of a batch is determined by the slowest (i.e., last finishing) job. The volatility and unreliability of the resources in VC can lead to prolonged makespan even if it is caused by a small fraction of jobs of a batch. This phenomena is referred as the “tail-effect” [11]. Tackling this is another step towards Volunteer Clouds. As the last challenge (of VC in this thesis) I am proposing algorithms that address prolonged makespan from different aspects derived from real world scenarios.

1.2 Structure

In particular this thesis is divided into 4 chapters and an appendix. The first contains the introduction, motivation and description of the structure of the thesis and the final chapter summarizes, details future work and the applicability of the results attained in this thesis. Chapters 2 and 3 each discuss a theorem group. Their content is as follows:

Chapter 2  This chapter first makes a comparison of volunteer and desktop grid computing based on key characteristics and attributes. It establishes a non-formal distinction between the two and serves as a foundation for the formal models introduced late in the chapter. It introduces the formal method used in the chapter and the related works (models) used in the chapter. Following these I define the formal models $M_{GROUND-DG}$ for desktop grids, $M_{VC-VOICE}$ and $M_{VC-SPOT}$ for volunteer computing and $M_{BOINC}$ for
BOINC in particular. The model of BOINC has a twofold role: it serves as validation for the previous models but also aims to be guide how other VC middleware can be formalized based on the previous models. The relationship of the different models are shown in Figure 1.1. (Model $M_{FED-BOINC}$ on the same figure is discussed in Chapter 3.) The relations and the implications of the relations are also discussed in this chapter. Related publications are [12], [13], [14], [15] and [16].

![Figure 1.1: Proposed formal models and their relations.](image)

Chapter 3  In this chapter I discusses different aspects of volunteer computing. The chapter is divided into three segments:

**Section 3.1** discusses federating volunteer computing. I define the notion and possibilities for federating VC. I introduce federation approaches to the previous formal model $M_{BOINC}$. I show how these approaches can be used to achieve a hierarchy of BOINC projects yielding $M_{FED-BOINC}$ (see Figure 1.1). Based on the relation of the models (discussed in Chapter 1) these methods can be applied to DGs as well. Related publications are [17], [18], [19], [20] and [21].

**Section 3.2** discusses volunteer clouds. I define the notion of Volunteer Cloud Computing based on a definition for Cloud Computing. I introduce and discuss three abstraction frameworks that help Volunteer Computing achieve the service models and characteristics defined for Volunteer Clouds. Related publications are [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37] and [21].

Finally **Section 3.3** discusses the volatility of volunteer resources that makes computing batches of jobs problematic. A single delayed job will delay the completion of a whole batch. In this section I present three algorithms that are validated through real world scenarios and show their effectiveness. The scenarios are detailed further in Appendices A.7 and A.8. Related publications are: [38], [39], [40] and [41].
Volunteer Computing resembles Desktop Grids whereas DGs are not fully equivalent to VC. Due to their similarities, they are often used as synonyms or in an interchanged meaning. This chapter makes an attempt to formalize their characteristics and relationship. To this end formal modeling is applied that tries to grasp the semantic of their functionalities – as opposed to comparisons based on properties, features, etc. I apply this modeling to
formalize the Berkeley Open Infrastructure for Network Computing (BOINC) VC system. The results of this work are formal models based on the Abstract State Machines (ASM) that (i) allow categorizing existing middleware and (ii) are aimed serving as a foundation for formalizing other VC Systems (VCS’). Figure 1.1 in Section 1.2 shows the models and their relation, from those models the first four are discussed in this chapter. The models are developed using the abstract state machines (ASM) framework and build on a previous work that formalized (service) Grid Computing in general (see Section A.4).

2.1 Characterization of desktop grids and volunteer computing

This section makes an attempt to make a brief informal comparison between desktop grids and volunteer computing. I make the comparison based on the following aspects: Main, Voluntarism, Resources, Applications, Security and Administration.

The Main aspect serves as a summary of the generic differences between the two. It is detailed in Table 2.1. The other five aspects can be regarded as direct consequences of these. The purpose of DGs is to solve compute intensive problems arising at an institution using the available resources of the institution. These may be dedicated or non-dedicated (cycle-scavenging) resources. All resources are owned, thus managed and maintained by the same institution. Contrary VC is usually used to solve a single grand scientific challenge where the researchers do not have access to a large set of computing capacity or do not want to maintain one for the experiment. VC uses non-dedicated hosts donated by individuals and connected through via the Internet. As a downside there is no direct control over these. The upside is that these resources are managed and maintained by their donors VC only needs to maintain the motivation of the donors towards itself.

<table>
<thead>
<tr>
<th>Main</th>
<th>Desktop Grids</th>
<th>Volunteer Computing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>- Solve (compute-intensive) scientific problems from researchers inside the institution</td>
<td>- Typically solve a single compute-intensive scientific problem</td>
</tr>
<tr>
<td>Resources</td>
<td>- Owned by the institution</td>
<td>- Donated voluntarily by individuals (Donors)</td>
</tr>
<tr>
<td>Method</td>
<td>- Dedicated resources and cycle-scavenging</td>
<td>- Cycle-scavenging</td>
</tr>
<tr>
<td>Problem Type</td>
<td>- Compute-intensive  (data-intensive possible)</td>
<td>- Compute-intensive</td>
</tr>
<tr>
<td></td>
<td>- Independent tasks: &quot;embarrassingly&quot; parallel applications</td>
<td>- Independent tasks: &quot;embarrassingly&quot; parallel applications</td>
</tr>
</tbody>
</table>

Table 2.1: Characteristics comparison: Main aspect
2.1. Characterization of desktop grids and volunteer computing

The Voluntarism aspect details the motivation, incentives and rewards for donating resources (see Table 2.2). For DGs resources are owned by organizations and the person sitting in front of the computer has no influence (sometimes also no knowledge) and thus no motivation for contributing to the DG. On the other hand in VC each person who owns a computer can donate its idle resources for a grand (scientific) challenge. To attract donors (and their computers) it must be transparent (e.g., explaining its goals and past results), attractive (e.g., fancy website, low barrier for newcomers, etc.), should forge a community and support it (e.g., social media, forums, etc.) and also provide rewards for donors and foster competition among them (e.g., with gamification techniques). The main challenge is keeping the motivation and retaining existing donors while attracting new ones.

<table>
<thead>
<tr>
<th>Voluntarism</th>
<th>Desktop Grids</th>
<th>Volunteer Computing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Donor motivation</td>
<td>×</td>
<td>- Contributing to a grand scientific challenge</td>
</tr>
<tr>
<td>Expected reward for donors</td>
<td>×</td>
<td>- Virtual credits/ rewards proportional to the contribution</td>
</tr>
<tr>
<td>Feedback for donors</td>
<td>×</td>
<td>- Information about progress of computation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Estimation of remainder of computation for current tasks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Profile and history about amount of contribution</td>
</tr>
<tr>
<td>Donor/desktop attraction</td>
<td>×</td>
<td>- Attractive webpage describing scientific goals, results and current status</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Social aspects: medium for discussion and sharing experience (e.g., forum, Facebook group)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- GUI for managing the tasks on own resources</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Eye candy, e.g., fancy screensaver</td>
</tr>
<tr>
<td>Challenges</td>
<td>×</td>
<td>- Attract more resources and donors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Retain existing resources and donors</td>
</tr>
</tbody>
</table>

Table 2.2: Characteristics comparison: Voluntarism aspect

Table 2.3 details the Resources aspect. The characteristics of the resources are mainly determined whether they are institutional ones or donated by individuals. Institutional resources are more or less homogeneous in terms of hardware and software. They are connected by higher speed networking with symmetric upload and download bandwidths. They are usually less prone to failure as they are managed by some central entity. They tend to have a lower volatility, since they usually have a pre-set availability during time of day, e.g., during business hours they are not available or limited part of their resources (i.e., memory, processor time, disk) is usable. Contrary in VC resources are owned by individuals meaning diverse software and hardware environments. These resources are connected to the Internet via ADSL or cable modems usually, and tend to have a lower bandwidth
that is asymmetrical (i.e., upload bandwidth is fraction of the download one). Also these computers tend to be behind NAT (Network Address Translation) and/or firewalls that makes them accessing from the Internet not possible. The reliability (in terms of hardware/software errors) is considered worse since each resource is maintained independently by their owner with different competence and effort. Volatility is expected to be higher since each donor has different computer usage, e.g., some allow the VCS to run whenever the computer is running, some only when it is idling, some have their computer running 24/7 and some turn it on for only shorter periods. For DGs there is no significant challenge here, they should be able to handle gracefully the non-dedicated nature of the resources. Contrary in VC mechanisms are needed that overcome the high volatility, low reliability, heterogeneity and network idiosyncrasies of the resources. There is no guarantee that a job will finish and that it will produce the correct result.

Table 2.3: Characteristics comparison: Resources aspect

<table>
<thead>
<tr>
<th>Resources aspect</th>
<th>Desktop Grids</th>
<th>Volunteer Computing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>- Desktop and server class&lt;br&gt;- Within an institution (e.g., university or company)</td>
<td>- Desktop computers&lt;br&gt;- Owned by individuals</td>
</tr>
<tr>
<td>Allocation</td>
<td>- Based on decision by in-charge person</td>
<td>- Voluntarily by the owner (donor)</td>
</tr>
<tr>
<td>Spatial distribution</td>
<td>- Local (LAN or WAN)</td>
<td>- Global (Internet)</td>
</tr>
<tr>
<td>Network</td>
<td>- High speed and low latency&lt;br&gt;- 100Mbit+ Local Area Network (LAN)&lt;br&gt;- Symmetric: same upload and download bandwidth&lt;br&gt;- Usually direct connection between client and server&lt;br&gt;- Dedicated IP addresses</td>
<td>- Variable speed and latency&lt;br&gt;- Usually ADSL, cable or even dial-up&lt;br&gt;- Asymmetric: lower bandwidth for uploads&lt;br&gt;- No direct connection between client and server (NA(P)T, firewalls)&lt;br&gt;- No dedicated IP addresses for the resources</td>
</tr>
<tr>
<td>Heterogeneity</td>
<td>- Low heterogeneity (e.g., CPU, Memory, OS and software environment)</td>
<td>- High heterogeneity</td>
</tr>
<tr>
<td>Dedication</td>
<td>- Resources are usually dedicated during non-business hours and owner prioritized or not usable during business hours (non-dedicated, desktop class)&lt;br&gt;- Some additional dedicated resources (server class)</td>
<td>- Owner prioritized (non-dedicated) resources</td>
</tr>
<tr>
<td>Volatility</td>
<td>- Low volatility</td>
<td>- Highly volatile</td>
</tr>
<tr>
<td>Challenges</td>
<td>×</td>
<td>- Volatility, heterogeneity, dedication and network</td>
</tr>
</tbody>
</table>

The type of supported applications is detailed by the Applications aspect. This is affected by the characteristics of the Resources. DGs and VCS’s are both suited mainly for HTC/MTC workloads as identified by several works, e.g., [42, 4, 5] and shown in Table 2.4. While DGs are set up by an organization for its internal goals, and usually runs multiple applications, a VC deployment is aimed at solving a single grand challenge in
2.1. Characterization of desktop grids and volunteer computing

the form of a single application. A VC deployment supporting multiple grand challenges, i.e., multiple applications is referred as an umbrella project. Any application deployed in a VCS should be prepared and hardened to run under diverse system architectures, operating systems and other software environments. The process of adapting applications to the requirements of any environment is referred as porting. Porting and supporting legacy applications (see Section 1.1) on DGs is a tedious process and for VC it is even harder since multiple hardware, operating systems and additional software environments must be supported.

<table>
<thead>
<tr>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
</tr>
<tr>
<td>- “embarassingly parallel” – HTC</td>
</tr>
<tr>
<td>- (HPC)</td>
</tr>
<tr>
<td>Number</td>
</tr>
<tr>
<td>- Multiple applications</td>
</tr>
<tr>
<td>Platform</td>
</tr>
<tr>
<td>- Usually a single binary is enough per application</td>
</tr>
<tr>
<td>Challenges</td>
</tr>
<tr>
<td>- Support legacy applications</td>
</tr>
<tr>
<td>- Porting applications</td>
</tr>
<tr>
<td>Desktop Grids</td>
</tr>
<tr>
<td>- “embarassingly parallel” – HTC</td>
</tr>
<tr>
<td>- Mainly single application, but multiple also possible (“Umbrella”)</td>
</tr>
</tbody>
</table>

Table 2.4: Characteristics comparison: Applications aspect

<table>
<thead>
<tr>
<th>Administration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server/Manager</td>
</tr>
<tr>
<td>- Full control</td>
</tr>
<tr>
<td>- Managed by professionals</td>
</tr>
<tr>
<td>Resources</td>
</tr>
<tr>
<td>- Centralized management by professionals</td>
</tr>
<tr>
<td>- Full control</td>
</tr>
<tr>
<td>Challenges</td>
</tr>
<tr>
<td>- Resource management is problematic when resources belong to different organizational units (e.g., departments at a university)</td>
</tr>
</tbody>
</table>

Table 2.5: Characteristics comparison: Administration aspect

The Administration aspect is detailed in Table 2.5. In a DG each host is administered usually by a centralized entity (e.g., IT department), thus they have full control over them. The challenge is when hosts are maintained by multiple organizations, e.g., part of the hosts donating resources are owned by a different department. In VC administrators have no control over the hosts of the donors. They can only motivate and reward. This has a large effect on reliability and trust, which are discussed next.

As Table 2.6 lists, in a volunteer environment donated host cannot be trusted, thus no
result returning from these hosts can be trusted either. This is detailed by the *Security* aspect. Effective validation mechanisms are required for checking results or filtering out erroneous and malicious hosts. For DGs erroneous results caused by failing hardware is less likely while malicious hosts are not present at all. In VC the donors need to trust the project that their computer is used only for the advertised purposes (e.g., calculation of a scientific problem). There is no such trust in DGs where the users of the computers most likely are not aware of the DG.

As conclusion we can see that VC is able to cope with not just the volatility, but the diversity and unreliability of connected resources, while DGs are suited for a controlled environment where the volatility is lower and resources are trusted.

### 2.1.1 Related work

Trieflinger [42] argues that the most used volunteer computing platform – BOINC – will not become an universal platform for desktop grid computing as it is lacking support for “beyond embarrassingly parallel applications”. He states that DGs are currently suited mainly for HTC tasks and proposes a method how they could support HPC workloads. In his thesis he does not distinguish between DGs and VC. Both are referred as desktop grid computing with characteristics and use cases that resemble volunteer computing rather than desktop grids. He identifies the following as major characteristics of DGs: volatility, heterogeneity, restricted connectivity and occasionally occurring error conditions. He concludes that service and desktop grids are fundamentally different and unifying them by dissolving one into another has little prospect.

I think although these assumptions are valid, however there is no need to dissolve one into another. There are other ways to create a combined infrastructure using both DG/VC and dedicated resources (from clouds or grids) and exploit its benefits. Dedicated resources are reliable but have costs associated with deployment and management (i.e., grids) or directly with usage (i.e., clouds). Volunteer resources are the opposite. Augmenting one with the other would allow to (a) transfer critical workloads from volunteer resources to
dedicated ones; and (b) transfer non-critical workloads to cheap or free volunteer resources. The goal of the EU funded EDGeS (Enabling Desktop Grids for e-Science) project and its follow up project EDGI (European Desktop Grid Initiative) was the interoperability between service and desktop grids (and clouds when used as compute nodes) by providing bridges for workload transfer in all directions [20, 43, 44, 45].

Kondo et al. [46] investigate the CPU availability of volunteer desktop grid resources: analyze temporal characteristics and construct a “server equivalent” for desktop grids based on the findings. They used the Entropia commercial desktop grid platform [47] with over 220 hosts to characterize CPU availability and develop a performance model. Their cluster equivalence ratio states that "Given an \( N \)-host desktop grid, how many nodes of dedicated cluster, \( M \), with comparable CPU clock rates, are required such that the two platforms have equal utility?" They defined \( \frac{M}{N} \) as the cluster equivalence ratio of the desktop grid. They calculated the ratio for (a) different job sizes; (b) for two host performance classes and (c) separately for weekdays and weekends. Their finding is that during weekdays for >10 minutes job sizes the ratio is between 0.65 and 0.75 and for weekends it is even higher between 0.9 and 0.95. They argue that DGs are mainly used for embarrassingly parallel applications and more advanced usage is impeded by the lack of understanding of resource availability characteristics and lack of quantitative models and metrics of utilization of resources and the entire platform.

Choi et al. [4, 5] state that DGs have received increased attention for executing high throughput workloads as resources are becoming cheaper. They argue that DGs are different from service grids in many aspects, but there is no taxonomy or survey on DGs. They categorize DGs based on organization (centralized or distributed), platform, scale (Internet or LAN) and resource providers (volunteer or enterprise) characteristics. They also compare VC (they refer it as volunteer desktop grids) and DGs (referred as enterprise desktop grids by the authors) to service grids on an informal per attribute basis, and provide no insight what the relation between the DGs and service grids could be.

Wang et al. [6] uses a formal method inspired by Mobile Ambients to build a formal model for VC by identifying the different roles for hosts in VC and describing their relation and interaction. The model is derived mainly based on the characteristics of XtremWeb(-HEP) [48]. They state that their model can help to lay a strong foundation for further research on formalisms of VC. However their model does not distinguish between DGs and VC and seems generic in an extent that most DG systems could fit it as well. Also it seems their generic model is derived from a single specific middleware (XtremWeb-HEP). What if their basic assumption is not true, i.e., their demonstrated middleware is not suited for being a VC middleware? For example in [4] Choi et al. state: “Lack of trust: In Desktop Grid, anonymous nodes can participate as a resource provider. Some malicious resource providers tamper with the computation and then return corrupted results. A scheduler should guarantee the correctness of results.”. In their comparison of volunteer and enterprise DGs result certification (see Section 2.3 for more details on the subject) is listed for VC. However based on the XtremWeb-HEP documentation\(^1\): “Result

certification: The XWHEP middleware does not propose anything on this field. It is the end user responsibility to verify the results of his/her jobs.”

Zhao et al. [49] survey DGs from the aspects of peer-to-peer (P2P) paradigms. They argue that separately both DG and P2P paradigms were investigated is several works, but the convergence of the two paradigms was not investigated in a systematic way.

### 2.2 Modelling desktop grids

A formal model for desktop grids (DGs) is presented in this section. The modelling method used is Abstract State Machines (ASM). A brief introduction to ASM in general can be found in Appendix A.3. The model presented here is based on an ASM model for grids by Nemeth et al. [1] and Kertesz et al. [2]. A summary of their model can be found in Appendix A.4. I use formal methods to represent DG/VC systems: contrary to informal methods, that compare usually based on a selected characteristic, with formal methods, I seize the semantics of the systems.

Based on M\(_{GRID}\) I developed a formal model named M\(_{GROUND−DG}\) for modelling desktop grids. I proved that systems represented by M\(_{GROUND−DG}\) satisfy the requirements of M\(_{GRID}\):

\begin{center}
\begin{boxedminipage}{0.95\textwidth}
\textbf{Theorem I.1.}: M\(_{GROUND−DG}\) is an extension of M\(_{GRID}\) and is a formal reference model for the architecture and application-management of desktop grids thus, enables the differentiation of desktop grids from the traditional grids represented by M\(_{GRID}\).
\end{boxedminipage}
\end{center}

Based on this theorem, systems represented by model M\(_{GROUND−DG}\) are also considered as grids based on M\(_{GRID}\).

Related publications: [13], [14], [15] and [12].

### 2.2.1 Assumptions

The following assumptions are made for DGs for the ground model (M\(_{GROUND−DG}\)) based on the environment described for DGs in Section 2.1:

1. \textit{Resources for a desktop grid are provided usually by a single entity.}
   Desktop Grids are deployed within a organization unit of a university or company thus, collect the compute (or storage) resources of computer labs or office computers within those organizational units. For more details refer to \textit{Main and Resources aspects} in Section 2.1. These resources are usually maintained by a single organization (e.g., university faculty or department; or the IT department of a company). The entities providing (i.e., managing or owning) the resources are referred as \textit{providers}.

2. \textit{Hosts are required to deploy a small piece of software to connect to the desktop grid.}
   This software (referred as \textit{worker software}) acts as a handler of the desktop grid:
2.2. Modelling desktop grids

manages jobs on the host, performs network communication, suspends and resumes activities as needed, etc. However this software acts on behalf of the provider. It is hidden from the person (referred as owner) sitting in front of and directly interacting with the host without any possibility of direct influence.

The worker software can either be a standalone application that need to be installed on the host, like Condor [50] or Entropia [47], or a web based one like Javelin [51] and the GridBee framework [52]. In the latter case the web browser runs the software (e.g., JavaScript code, Java applets, etc.) downloaded from a web page. However, both cases imply that a handler software must be present on the host as a prerequisite. In both cases middleware refers to the DG software components at whole and worker refers to the deployed standalone or browser based component.

3. Hosts are volatile.

The hosts providing (computational) resources for the desktop grid are not dedicated and are not always accepting jobs. Desktop grids are primarily cycle-scavenging platforms. The resources are usually lab or office computers that are used by their owners through office hours. Depending on policies set by the provider the hosts may be used exclusively by their owners when interacting with the host (e.g., pressing a key on the keyboard) any time resulting in suspended desktop grid activities for that host. Owners have absolute priority over the activities of the desktop grid and the desktop grid must honor this. The worker software is usually configured either to run desktop grid activities in the background using the unused resources of the host or wait until the computer is idle. For more details refer to Resources aspect in Section 2.1.

4. Jobs can get delayed and even lost.

Contrary to traditional service grids where the resources are mostly available for the grid, in desktop grids hosts are not dedicated and owners are prioritized. It is possible that a host that is running a job is interrupted and won’t be available for the desktop grid for a longer period of time, perhaps even never again. This means that the completion of the job might be delayed or never happen. The desktop grid must be able to identify these late or “lost” jobs and handle them in some manner.

5. Small number of users.

Usually the desktop grid is used within an institute, meaning there will be only a limited number of users submitting jobs and as a consequence the number applications used for the jobs is usually limited as well.

6. All processes of a job assumed to be running on the same host.

Hosts in a desktop grid are not dedicated. There is no guarantee that multiple hosts will be available for the desktop grid at the same time and for the same time period. Executing parallel applications is not feasible this way. However they can always run on a single host.

The model presented here is a multi-agent ASM model for desktop grids where agents are jobs (i.e., elements from the JOB universe). The nullary self function \( j \in JOB \) allows an
agent to identify itself among other agents. The different agents interpret it differently. The rules presented here form a module, i.e., a single-agent model that is executed by each agent. Agents have the same initial states as described later in Section 2.2.3.

2.2.2 Universes and signatures

Table 2.7 shows the basic components (universes) of the ASM model and their counterpart from the base Grid ASM model $M_{GRID}$ (see Section A.4). HOST denotes in both models the entities that provide resources for the system. NODE is introduced to have universal denotation for the system components (UI, MANAGER, HOST). With UI and MANAGER I want to give a more comprehensive picture of the mechanisms and concepts of the systems via my models. These can be extracted freely from my models, but can be included in $M_{GRID}$ without any issues. I am also omitting PROCESS. I assume that TASK includes all its processes on the resource (HOST), it is a simplification helping to reduce lower-level details. The following universes are available in the $M_{GROUND-DG}$ model (see Figure 2.1):

<table>
<thead>
<tr>
<th>$M_{GRID}$</th>
<th>$M_{GROUND-DG}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER</td>
<td>USER</td>
</tr>
<tr>
<td>JOB</td>
<td>JOB</td>
</tr>
<tr>
<td>PROCESS</td>
<td>-</td>
</tr>
<tr>
<td>ARESOURCE</td>
<td>ARESOURCE</td>
</tr>
<tr>
<td>-</td>
<td>NODE</td>
</tr>
<tr>
<td>-</td>
<td>UI</td>
</tr>
<tr>
<td>-</td>
<td>MANAGER</td>
</tr>
<tr>
<td>HOST</td>
<td>HOST</td>
</tr>
<tr>
<td>TASK</td>
<td>TASK</td>
</tr>
<tr>
<td>PRESOURCE</td>
<td>PRESOURCE</td>
</tr>
</tbody>
</table>

Table 2.7: Basic elements of the ASM model: on the left side the base Grid ASM model $M_{GRID}$, while the $M_{GROUND-DG}$ model for desktop grids on the right side.

1. USER: A user of the desktop grid is the entity that submits jobs and retrieves results. There might be multiple users for a single desktop grid each responsible for own jobs.

2. JOB: The job submitted by a user to be executed on a host. Usually contains binaries of an application and its libraries, input and output files and resource requirement descriptions.

3. ARESOURCE: Abstract resources represent the resource requirements of a JOB (e.g., operating system, CPU architecture, memory and disk requirements). These resource requirements can be satisfied by the physical resources offered by the hosts (see PRESOURCE below).

4. NODE: Umbrella term for user interfaces, managers and hosts (see below) in the desktop grid. Typically a virtual or physical machine where the required software components are deployed.
5. UI: user interface (UI). A type of node where jobs can be submitted to connected desktop grids by users. It acts as a gateway and the pool of resources of the desktop grid can be accessed through it.

6. MANAGER: The main component of a desktop grid. It is a type of node that manages resources and allocates Jobs to hosts. Jobs are submitted through UIs to managers. In a centralized DG there is usually a single manager, but for example in a peer-to-peer system each connected host can be a manager as well that communicates with other managers.

7. HOST: provides physical resources (PRESOURCE) for jobs and thus, executes them. Hosts are computers of a lab, office, etc. resources for the desktop grid. The worker component of the DG is installed on the hosts. This worker acts as a handler on behalf of the host for the DG. Since all hosts have workers installed these won’t be distinguished in the model rather, only the host referenced with different context.

8. TASK: The physical representation of a job installed on a host. All processes of a specific job executing on a host are represented by a task.

9. PRESOURCE: physical resources provided by a host. These resources are compared to the abstract resources required by jobs. If they are compatible the job can be executed on the host. The physical resources provided for the job can be anytime revoked temporarily or for longer periods of time by the owner.
The following functions are modified compared to their counterparts in $M_{GRID}$:

\[
\text{attr} : \{\text{ARESOURCE},\text{PRESOURCE},\text{UI}, \\
\text{MANAGER},\text{HOST},\text{TASK}\} \rightarrow \text{ATTR} \tag{2.1}
\]

\[
\text{canLogin} : \text{USER} \times \text{NODE} \rightarrow \{\text{true}, \text{false}\} \tag{2.2}
\]

\[
\text{canUse} : \text{USER} \times \{\text{NODE},\text{PRESOURCE}\} \rightarrow \{\text{true}, \text{false}\} \tag{2.3}
\]

\[
\text{mappedResource} : \text{JOB} \times \text{ARESOURCE} \rightarrow \text{PRESOURCE} \tag{2.4}
\]

\[
\text{uses} : \text{JOB} \times \text{PRESOURCE} \rightarrow \{\text{true}, \text{false}\} \tag{2.5}
\]

\[
\text{mapped} : \text{JOB} \rightarrow \text{HOST} \tag{2.6}
\]

\[
\text{location} : \text{PRESOURCE} \rightarrow \text{HOST} \tag{2.7}
\]

\[
\text{handler} : \text{PRESOURCE} \rightarrow \text{HOST} \tag{2.8}
\]

\[
\text{localuser} : \text{HOST} \rightarrow \text{USER} \tag{2.9}
\]

\[
\text{globaluser} : \text{JOB} \rightarrow \text{USER} \tag{2.10}
\]

The $\text{canUse}$ shared function denotes if a user is granted to use the resources provided by the host, while $\text{canLogin}$ specifies if the user can login to a host. In a DG a user has access to the pool of resources ($\text{canUse}(u,n) = \text{true}$), but cannot access the hosts individually ($\forall h \in \text{HOST} : \text{canLogin}(u,n) = \text{false}$). Jobs are mapped to certain hosts for execution. This is denoted by the $\text{mapped}$ function. $\text{mappedResource}$ function is handled by the resource mapping functionality (see Functionality F1.2) and describes the abstract to physical resource mapping. It is assumed that all processes of a job are mapped onto the same host and thus, the mapping between the job and the physical resources must be provided by a single host and not between individual processes of the job and physical resources of multiple hosts. The $\text{location}$ function determines which host provides the given physical resource. The $\text{handler}$ function represents the auxiliary "handler" component on a physical resource that provides the execution platform (i.e., the worker) as defined by $M_{GRID}$. However since processes are not part of this model the function is modified to represent the host providing the physical resources (rather than a process).

The following functions are adopted in unmodified form from the $M_{GRID}$ model:

\[
\text{compatible} : \text{ATTR} \times \text{ATTR} \rightarrow \{\text{true}, \text{false}\} \tag{2.11}
\]

\[
\text{belongsTo} : \text{HOST} \times \text{PRESOURCE} \rightarrow \{\text{true}, \text{false}\} \tag{2.12}
\]

\[
\text{installed} : \text{TASK} \times \text{HOST} \rightarrow \{\text{true}, \text{false}\} \tag{2.13}
\]

\[
\text{usermapping} : \text{USER} \times \text{USER} \rightarrow \{\text{true}, \text{false}\} \tag{2.14}
\]

The $\text{compatible}$ function determines if two attributes (e.g., an element of PRESOURCE and an element of ARESOURCE) are compatible or equal in some sense. For example the abstract resources can be compared to physical resources to see if a host has the resources for running a specific job. The $\text{installed}$ function tells if a task is installed on a given host or not. The $\text{usermapping}$ function is used by the user mapping functionality (see Functionality F1.2) to map the user of the job to a local user of the host where the job will be executed. The $\text{belongsTo}$ function determines the host where the physical resource belongs to.
The following new functions are introduced by the DG model:

\[
\text{available} : \text{HOST} \rightarrow \{\text{true, false}\} \quad (2.15)
\]
\[
\text{acceptsJobs} : \text{HOST} \rightarrow \{\text{true, false}\} \quad (2.16)
\]
\[
\text{node} : \{\text{NODE, UI, MANAGER}\} \rightarrow \text{HOST} \quad (2.17)
\]
\[
\text{provides} : \text{HOST} \times \text{PRESOURCE} \rightarrow \{\text{true, false}\} \quad (2.18)
\]
\[
\text{resourceRequest} : \text{JOB} \times \text{ARESOURCE} \rightarrow \{\text{true, false}\} \quad (2.19)
\]
\[
\text{submitted} : \text{JOB} \times \text{USER} \times \text{UI} \times \text{MANAGER} \rightarrow \{\text{true, false}\} \quad (2.20)
\]
\[
\text{readyForDelete} : \text{JOB} \rightarrow \{\text{true, false}\} \quad (2.21)
\]
\[
\text{managerOf} : \text{HOST} \rightarrow \text{MANAGER} \quad (2.22)
\]
\[
\text{jobPhase} : \text{JOB} \rightarrow \{\text{init, readytorun, running, finished}\} \quad (2.23)
\]
\[
\text{jobOutcome} : \text{JOB} \rightarrow \{\text{success, failure}\} \quad (2.24)
\]
\[
\text{uiOf} : \text{MANAGER} \rightarrow \text{UI} \quad (2.25)
\]
\[
\text{capability} : \text{JOB} \times \text{CAPABILITY} \rightarrow \{\text{true, false}\} \quad (2.26)
\]

\[
\text{createCheckpoint},
\]
\[
\text{resumeCheckpoint},
\]
\[
\text{checkpoint},
\]
\[
\text{readytoCheckpoint},
\]
\[
\text{suspended} : \text{JOB} \rightarrow \{\text{true, false}\} \quad (2.27)
\]

A peculiarity of desktop grids used in the model is that hosts are not always connected and available. The \text{available} function tells whether a host is available for the desktop grid at a given time. The \text{acceptsJobs} monitored function determines whether the host is willing to accept jobs. These complement each other: the former tells whether there is a connection between the desktop grid and the host, while the latter represents an internal state of the host. The \text{provides} function determines if the host currently provides the physical resource. The \text{submitted} function denotes if an job was submitted using the given ui, to a given manager by a given user. In this model it is assumed that every job submitted is submitted successfully.

After a job finishes (regardless of successfully or not) the users retrieve the results and the job is discarded. Whether the job is ready for deletion is determined by the monitored function \text{readyForDelete}. The \text{jobPhase} represents the different phases of the life-cycle of the job (was state in \text{MGRID}). \text{jobOutcome} represents the outcome of a finished job. It can be either successful or failed.

Events represent various external and internal occurrences affecting the desktop grid. Events either originate from external sources (e.g., from the user who submitted the job) or are generated internally by the worker (e.g., configured policies such as activity suspension of activities by the worker: the owner wants to use her host). Events are generated by corresponding functionalities using the \text{event} function and are interpreted by the different rules in the model. Event functionalities are detailed in Section 2.2.4.
Chapter 2. Formal models for desktop grids and volunteer computing

The \textit{EVENT} set describes the events that are considered for jobs in the model:

\[ \text{EVENT} = \{ \text{start, stop, suspend, resume, abort, terminate, ckpt} \} \quad (2.28) \]

\[ \text{event} : \text{JOB} \rightarrow \text{EVENT} \quad (2.29) \]

\[ \text{eventMasked} : \text{JOB} \times \text{EVENT} \rightarrow \{ \text{true, false} \} \quad (2.30) \]

\[ \text{maskEvents} : \text{JOB} \times 2^{\text{EVENT}} \rightarrow \{ \text{true, false} \} \quad (2.31) \]

The \textit{event} function sends an event to a job, while the \textit{eventMasked} allows masking the specified event from a job until it is re-enabled by setting the function with the same parameters to false. This means that while an event is masked for a job, the job will not receive those events under any circumstances. This masking functionality is an internal one of the event function. The \textit{start} event signals the start of the execution of the job. The \textit{stop} event means that the execution is interrupted (e.g., the DG middleware is shutting down), however the job is not finished and can be restarted (or resumed from a checkpoint) later. \textit{Suspend} and \textit{resume} events handle shorter interruptions in the execution (e.g., internal scheduling policies or CPU usage throttling). The \textit{abort} event represents either an error in the execution or some external occurrences abort the job. Finally, \textit{ckpt} represents a request for the job to save its state by whatever method available. The \textit{maskEvents} is a convenience function that uses \textit{eventMasked} to mask/ unmask events for a job.

2.2.3 Initial state

In the following we define the initial states of the model \( M_{\text{GROUND-DG}} \):

1. Each user can use at least a single \textit{UI} type of host (submit and execute jobs):

\[ \forall \text{us} \in \text{USER} : \exists u \in \text{UI}, \exists h \in \text{HOST} : \]

\[ \text{canUse}(\text{us}, \text{node}(u)) = \text{true} \land \text{canUse}(\text{us}, \text{node}(h)) = \text{true} \quad (2.32) \]

2. \textit{Hosts} are connected to managers, \textit{managers} are connected to \textit{ui-s} and \textit{managers} have both \textit{ui-s} and \textit{hosts}:

\[ \forall h \in \text{HOST} : \exists \text{mgr} \in \text{MANAGER} : \text{managerOf}(h) = \text{mgr} \quad (2.33) \]

\[ \forall \text{mgr} \in \text{MANAGER} : \exists u \in \text{UI} : \text{uiOf}(\text{mgr}) = u \quad (2.34) \]

\[ \forall \text{mgr} \in \text{MANAGER} : \exists h \in \text{HOST} : \]

\[ \text{uiOf}(\text{managerOf}(h)) \neq \text{undef} \quad (2.35) \]

3. No job is suspended and no events are masked:

\[ \forall \text{job} \in \text{JOB}, \forall e \in \text{EVENT} : \]

\[ \text{suspended}(\text{job}) = \text{false} \land \text{maskedEvent}(\text{job}, e) = \text{false} \quad (2.36) \]
4. Miscellaneous relations and functions of initial state based on $M_{GRID}$:

\[
\forall job \in JOB, \exists us \in USER : user(job) = us \tag{2.37}
\]

\[
\forall job, \exists ar \in ARESOURCE : resourceRequest(job, ar) = true \tag{2.38}
\]

\[
\forall job, \forall pr \in PRESOURCE : uses(job, pr) = false \tag{2.39}
\]

\[
\forall job : task(job) = undef \tag{2.40}
\]

\[
\forall job : mapped(job) = undef \tag{2.41}
\]

\[
\forall job : jobPhase(job) = undef \tag{2.42}
\]

\[
\forall u \in USER, \exists pr \in PRESOURCE : canUse(u, node(handler(pr))) = true \tag{2.43}
\]

2.2.4 Functionalities

Resource and user mapping functionalities These functionalities must be provided by every distributed system in order to qualify as a grid. It is a requirement of $M_{GRID}$.

**Functionality F1.1: Resource mapping**

The resource mapping functionality (see Listing 2.1) allocates the abstract resource requests of a job to physical resources provided by available hosts. This is denoted by the following rule:

```plaintext
if (\exists ar \in ARESOURCE, \exists h \in HOST, \exists job \in JOB):
    mappedResource(job, ar) = undef \land resourceRequest(job, ar) = true

choose pr in PRESOURCE satisfying
    compatible(attr(ar), attr(pr)) = true \land belongsTo(pr, h) = true \land
    handler(pr) = h \land available(h) = true \land acceptsJobs(h) = true

    mappedResource(job, ar) := pr

endif
```

Listing 2.1: Functionality F1.1: Resource mapping

This is a refined version of the module proposed in $M_{GRID}$ model for service grids. The following changes are introduced in $M_{GRID}$ for functions: the $mappedResource$ and $resourceRequest$ (was request in $M_{GRID}$) functions use jobs instead of processes. This version of the module takes into account the following characteristics of desktop grids: Hosts in the pool of resources ($pr \in PRESOURCE$) are not dedicated, thus it cannot be assumed that they are always connected to the desktop grid ($available(h) = false$).

**Functionality F1.2: User abstraction**

User abstraction is shown in Listing 2.2. One of the essential characteristics of grid systems is user abstraction that realizes a functionality by which trusted users granted access to a resource are mapped to local users temporarily and can act on behalf of the local users. The fact that a user granted access to a resource does not mean she can log in to the host where the resource is and e.g., can start a process: $canUse(u, n) = true \neq canLogin(u, node(r)) = true$. This may be solved by mapping a ‘global user’ to a local user’s account (e.g., as it was done in Globus [53]) or allowing the ‘global user’ to use a service that is executed on behalf of the local user. Since desktop grids assume that a client
program – the worker – is installed on the host and provides the handler functionality
serves exactly this purpose. The worker acts on behalf of the host as a handler (the handler
function) for the physical resources and also runs as localuser and based on the actual
authorization mechanism implemented by the DG middleware provides the local part of
the user mapping. Thus, taking the corresponding definition from [1] for user abstraction
and omitting the unnecessary parts, we got that the system must provide a functionality
equivalent to:

\[
\text{let } pr = \text{mappedResource}(\text{job}, ar) \\
\text{if } (\exists ar \in ARESOURCE, \exists \text{job} \in JOB): \text{resourceRequest}(\text{job}, ar) = \text{true} \land \\
pr \neq \text{undef} \land \text{canUse}((\text{user}(\text{job}), pr) = \text{true then} \\
\text{usermapping}((\text{globaluser}(\text{job}), pr) := \text{localuser}(\text{handler}(pr)) \\
\text{endif}
\]

Listing 2.2: Functionality F1.2: User mapping

Event functionalities Based on the assumptions made at the beginning of the section
the following events are included in the model:

Functionality F1.3: Delayed jobs
Desktop grids may utilize non-dedicated, heterogeneous resources and the case when a
job is delayed for any reason (or even lost) must be taken in account. The cause for such
delay can be numerous, e.g., resource requirements cannot be satisfied, the host the job is
mapped is claimed by its owner or is permanently shut down. The exact specifics of a job
considered as delayed is up to the desktop grid middleware providing the functionality. This
is represented by the high-level \(\text{jobDelayed} : \text{JOB} \rightarrow \{\text{true, false}\}\) monitored function in
this model. Also the concrete handling of such event is the responsibility of the middleware
(e.g., abort or migrate the job). The only restriction is that it must be ensured that the
job is completed or aborted in finite time. A simplistic description of the functionality can
be the following:

\[
\text{if } (\exists \text{job} \in JOB): \text{jobDelayed}(\text{job}) = \text{true then} \\
\text{maskEvents}(\text{job}, \{\text{start, resume, stop, suspend, ckpt}\}) := \text{true}; \\
\text{event}(\text{job}) := \text{abort} \\
\text{endif}
\]

Listing 2.3: Functionality F1.3: Delayed job handling

Functionality F1.4: Mapped resources become unavailable
In DGs owners of the resources are prioritized and resources mapped and assigned to
jobs can get unavailable for shorter or longer periods of time. For example such policies
are set that prioritize the user logged in to the host compared to the DG. Even at such
extent that the DG must suspend all activities during any local user activity. The exact
implementation is up to the desktop grid, but it must provide this functionality. Here the
\(\text{presource} : \text{JOB} \rightarrow \text{PRESOURCE} \) function denotes the mapped physical resources of
the job. The \(\text{provides}(\text{pr}, h) = \text{false}\) denotes that the host \(h\) (through the worker software)
can not provide (some or all of) the mapped physical resources for the job any more. In this
case the job must be suspended using the \text{suspend} event. The actual revocation (\(\text{uses}(j, \text{presource}(j)) := \text{false}\)) happens in the state transition rule for suspend (see Rule R1.6). The
task(j) = t implies that resources are mapped and the task is installed. In this realization it is assumed that the unavailability is a temporal state and not all of the resources are needed to be released, thus the job only needs to be suspended while so. The elseif statement is evaluated true if the resources are available again (provides(pr, h) = true) and the job is not using the resources (uses(j, pr) = false). In this case the resources are granted and the events are unmasked. The job is actually not restarted here, allowed to start only:

\[
\text{if } (\exists h \in \text{HOST}, \exists \text{job} \in \text{JOB}, \exists t \in \text{TASK}) : \text{task}(j) = t \land \text{installed}(t, h) = \text{true} \land \text{jobDelayed}(\text{job}) = \text{false} \text{ then}
\]

\[
\text{if } (\text{provides} (\text{presource} (\text{job}), h) = \text{false}) \text{ then}
\]

\[
\text{maskEvents(job, \{start, resume\}) := true}
\]

\[
\text{event(job) := suspend}
\]

\[
\text{else}
\]

\[
\text{maskEvents(job, \{start, resume\}) := false}
\]

\[
\text{endif}
\]

\[
\text{endif}
\]

Listing 2.4: Functionality F1.4: Mapped resources become unavailable

2.2.5 Job phase transition rules

In the following we detail the phase transitions rules for jobs (Rules R1.2 – R1.12). Figure 2.2 shows the phase transitions for a job. It is kept as simple as possible and can be extended with further rules and states. As an addition, check-pointing functionality is included with Rules R1.10 and R1.12.

![Diagram of job phase transition](image)

Figure 2.2: Job phase transition diagram with corresponding rules for the $M_{GROUND-DG}$ model.

**Rule R1.1: Job staging**
Jobs can be staged (mapped) onto hosts that are available for the desktop grid and provide the required resources (see Listing 2.5). The difference between mapped and mappedResource is that the former assigns a host to a job once using the latter resource requests of the job can be satisfied. A job might require just a fraction of resources a host provides.

```plaintext
if (∃ar ∈ ARESOURCE): resourceRequest(j, ar) = true ∧
mappedResource(j, ar) ≠ undef then
    let pr := mappedResource(j, ar)
    if (userMapping(globalUser(j), pr) ≠ undef) then
        mapped(j) := location(pr)
        resourceRequest(j, ar) := false
    endif
endif
```

Listing 2.5: Rule R1.1

**Rule R1.2: Job submitted**

A job is always submitted by a user, to a manager and by using a ui. Its initial state is init once the DG accepted the submission. This is denoted by the following rule:

```plaintext
if (∃mgr ∈ MANAGER, ∃u ∈ UI, ∃us ∈ USER): submitted(j, us, u, mgr) = true ∧
    mapped(j) = undef then
    jobPhase(j) := init
endif
```

Listing 2.6: Rule R1.2: Job submitted

**Rule R1.3: Task installed and physical resources allocated**

The job is ready to run after the required resources are selected (see Rule R1.1) and the job is mapped and installed on a host. This is described by the following rule:

```plaintext
if (∃pr ∈ PRESOURCE): mapped(j) ≠ undef ∧ uses(j, pr) = true ∧
    jobPhase(j) = init then
    extend TASK by t with
    task(j) := t
    installed(t, location(pr)) := true
endextend
checkpoint(j) := false
suspended(j) := false
jobPhase(j) := readytorun
```

Listing 2.7: Rule R1.3: Task installed and physical resources allocated

**Rule R1.4: Start job**

Jobs that are in the readytorun phase are allowed to enter the running phase. The start is denoted by the event start. The job will start to use the physical resources in the running phase. This is described by the following rule:

```plaintext
if (jobPhase(j) = readytorun ∧ event(j) = start ∧
    checkpoint(j) = false) then
    uses(j, presource(j)) := true
    jobPhase(j) := running
endif
```

Listing 2.8: Rule R1.4: Start task

**Rule R1.5: Stop job**


2.2. Modelling desktop grids

A job can be stopped by the *stop* event. A stopped job will return to the *readytostart* phase. It represents the case when the task of the job and its processes are stopped (terminated) but the job should not be considered failed, rather restarted when possible. For example, this is the case if the host running the job is shutting down, however once it becomes available it can restart the computation. This is described by the following rule:

if (jobPhase(j) = running ∧ event(j) = stop) then
    jobPhase(j) := readytorun
    uses(j, presource(j)) := false
endif

Listing 2.9: Rule R1.5: Stop job

**Rule** R1.6: Suspend job

Running jobs can be suspended by the middleware itself not just by external modules. It is denoted by the event *suspend*. This represents the case when the execution of a task must be interrupted for a short period, but can continue afterwards without restarting from beginning or from a checkpoint (e.g., a task and its processes are suspended). This is described by the following rule:

if (jobPhase(j) = {running | readytorun} ∧ event(j) = suspend ∧ suspended(j) = false) then
    suspended(j) := true
    uses(j, presource(j)) := false
endif

Listing 2.10: Rule R1.6: Suspend job

**Rule** R1.7: Resume job

Suspended jobs can be resumed by the *resume* event:

if (jobPhase(j) = {running | readytorun} ∧ event(j) = resume ∧ suspended(j) = true) then
    suspended(j) := false
    uses(j, presource(j)) := true
endif

Listing 2.11: Rule R1.7: Resume job

**Rule** R1.8: Job termination

The *terminate* event denotes job completion without failure. It is described by the following rule:

if (jobPhase(j) = running ∧ event(j) = terminate) then
    jobPhase(j) := finished
    jobOutcome(j) := success
    uses(j, presource(j)) := false
    TASK(task(j)) := undef
endif

Listing 2.12: Rule R1.8: Job finished

**Rule** R1.9: Job fails

A job receiving the *abort* event represents a failure or abort request by the user. This might be caused by erroneous application, hardware issues or by other causes. However, it shall remain abstract in this model. In case of any error the task is aborted and its job is considered failed. This is described by the following rule:
Chapter 2. Formal models for desktop grids and volunteer computing

if \( \text{event}(j) = \text{abort} \land \text{jobPhase}(j) \neq \text{undef} \) then
    \text{jobPhase}(j) := \text{finished}
    \text{jobOutcome}(j) := \text{failed}
    \text{uses}(j, \text{presource}(j)) := \text{false}
    \text{TASK(task}(j)) := \text{undef}
endif

Listing 2.13: Rule R1.9: Abort task

Rule R1.10: Perform checkpointing

Checkpoint requests are signaled by the event \( \text{ckpt} \). The capability of the job to perform the checkpoint operation is represented by the \text{capability} function with the \text{ckpt} parameter. A job can only be checkpointed if it is running and a suspended job should never receive the checkpoint event. This model does not detail the specifics of the checkpointing procedure. The rule is the following:

if \( \text{event}(j) = \text{ckpt} \land \text{capability}(j, \text{ckpt}) = \text{true} \land \\ \text{readyToCheckpoint}(j) = \text{true} \land \text{createCheckpoint}(j) = \text{true} \land \\ \text{checkpoint}(j) = \text{true} \land \text{suspended}(j) = \text{false} \) then
    \text{checkpoint}(j) := \text{true}
endif

Listing 2.14: Rule R1.10: Perform checkpointing

Rule R1.11: Job cleanup

The following rule ensures if the job has finished that it is removed from the JOB universe after its results were retrieved by the submitting user:

if \( \text{jobPhase}(j) = \text{finished} \land \text{readyForDelete}(j) = \text{true} \) then
    \text{JOB}(j) := \text{undef}
endif

Listing 2.15: Rule R1.11: Cleanup after finished job

The \text{readyForDelete} abstract function represents whether the user finished all interaction with the job after it finished (e.g., retrieved its outputs and logs) and the job can be deleted from the desktop grid. It is assumed that the middleware provides a mechanism for the user to signal this.

Rule R1.12: Resume from checkpoint

A resume happens if a job is in \text{readytorun} phase, it receives the \text{start} event and there is an existing checkpoint for the job.

if \( \text{jobPhase}(j) = \text{readytorun} \land \text{event}(j) = \text{start} \land \text{checkpoint}(j) = \text{true} \) then
    \text{resumeCheckpoint}(j) := \text{true}
    \text{checkpoint}(j) := \text{false}
    \text{uses}(j, \text{presource}(j)) := \text{true}
    \text{jobPhase}(j) := \text{running}
endif

Listing 2.16: Rule R1.12: Resume from checkpoint
2.2. Correspondence of M\textsubscript{GRID} and M\textsubscript{GROUND−DG}

Nemeth et al. [1] give the formal definition of a Grid in M\textsubscript{GRID} as: "A system is said to be a Grid if it can provide a service equivalent to Π\textsubscript{resource\_mapping} and Π\textsubscript{user\_mapping} according to some reasonable definition of equivalence[...]"] where the 'reasonable definition of equivalence' is used in a sense as in [54]. In the following, I am going to argue that M\textsubscript{GROUND−DG} models a valid grid based on this definition.

2.2.6.1 Equivalence of user mapping service in M\textsubscript{GRID} and M\textsubscript{GROUND−DG}

One of the essential characteristics of grid systems as defined by M\textsubscript{GRID} is user abstraction that realizes a functionality by which trusted users granted access to a resource are mapped to local users temporarily and can act on behalf of the local users. As stated: canUse(u, n) = true \n⇒\ canLogin(u, node(r)) = true, i.e., a user granted access to a resource cannot log in to the host where the resource is found and cannot start a process on his behalf on (desktop) grids. This is solved by mapping a 'global user' to a local user’s account. Thus there is a need for globaluser \mapsto localuser mapping functionality. The actual mapping is provided by the authorization mechanism of the DG middleware. The localuser is provided by the worker program thus, the direct resource access from the corresponding definition from [1] was omitted (it always evaluates to false). These result in Functionality F1.2.

If there is a request for a (abstract) resource and a (physical) resource is found and the user is allowed to use the found resource (however, cannot necessarily log in to it) then if and only if there exists a process that is able to handle the resource (i.e., provide services as a resource) the user is granted access to the resource mapped to the user. In the current scenario, user of the desktop grid is allowed to use resource and the worker installed by the resource owned acts as the handler: allows the user to execute a job on behalf of the resource owner. In such a way user abstraction (mapping global users to local accounts) is implicit in desktop grids.

2.2.6.2 Equivalence of resource mapping service in M\textsubscript{GRID} and M\textsubscript{GROUND−DG}

For desktop grids the hosts in the pool of resources (pr ∈ PRESOURCE) are not dedicated and it cannot be assumed that they are always connected to the desktop grid (available(h) = false). For example the owner uses exclusively the host and the worker is suspended. Nemeth et al. [1] argues somewhat similarly for service grids: "In Grids, the virtual pool of resources is dynamic and diverse, since the resources can be added and withdrawal at any time according to their owner’s discretion [..]". However the dynamism of the resources in [1] is assumed less frequent than for desktop grids where the volatility of resources is a fundamental characteristic of the system. On the other hand if the worker is active and the worker is connected to the desktop grid it still might not accept jobs (acceptJobs(h) = false). This can be for example due to the scheduling policies of the worker or set resource constraints by the worker. In M\textsubscript{GROUND−DG} jobs are entities to map resources to instead of processes, this introduced basic changes in the mappedResource and resourceRequest (was request in M\textsubscript{GRID}) functions. Additionally, resource mapping...
in desktop grids takes into account (a) the non-dedicated nature; and (b) the prioritization of the owner of the host providing resources over the desktop grid. Thus, in desktop grids physical resource mapping follows the scheme of $M_{GRID}$ with the refinement that physical resources can only be mapped to compatible and available hosts that have no local policies that prohibit accepting jobs (as described by Functionality F1.1).

### 2.2.7 Definition for Desktop Grid

Based on the arguments in Section 2.2.6.1 and 2.2.6.2 the relationship between $M_{GRID}$ and $M_{GROUND-DG}$ can be formulated as follows:

**Statement 1** The functionality F1.1 in $M_{GROUND-DG}$ is equivalent to $\Pi_{resource\_mapping}$ and functionality F1.2 is equivalent to $\Pi_{user\_mapping}$ module of $M_{GRID}$ thus, $M_{GROUND-DG}$ models a valid grid based on the definition of grid by Nemeth et al. [1] and, any system conforming to $M_{GROUND-DG}$ is a grid based on the same definition.

As a consequence, a definition for Desktop Grid should incorporate all aspects of a definition for Grid, but also emphasize the refinements identified in Section 2.2. These refinements are not necessarily restricted to desktop grids and can be applied to the base – $M_{GRID}$ – model as well. They are the following. Initially each user must have access to some uis and hosts so she can submit and run jobs (as described by initial state rule 2.32). Desktop grid has a specific topology comprising nodes with different roles: hosts, managers and uis are connected to form a desktop grid (as described by initial state rules 2.33 - 2.35). Events – as discussed before – represent external or internal occurrences that affect the DG. Most likely in every distributed systems there are events that affect the system, in case of desktop grids the volatility and non-dedicated nature of resources is dominant that a definition must incorporate. In any case it can be safely assumed that some resources are initially available for a desktop grid (as described by initial state rule 2.35). A DG should allow the interaction of any user with a host (e.g., using a office or lab computer for daily work) to take higher priority than the tasks the DG is running there. A DG must be prepared and provide the needed functionality (as described by Functionality F1.4). An idiosyncrasy of the non-dedicated and owner prioritized characteristics of DGs is that a submitted job is expected to "get lost". In traditional grids there is some dynamic behavior in the availability of resources (as discussed in Section 2.2.6) however in DGs this is a defining attribute and jobs are expected to be delayed or even to get lost caused by longer period of resource unavailability. Thus a desktop grid must be prepared to handle this and provide the needed functionality (as described by Functionality F1.3).

Additionally, there are five assumptions made at the beginning of Section 2.2 for DGs. In a DG resources are from a single provider thus, there is no requirement for administrative domains, resources are more homogeneous in terms of software and hardware components. This enables the implicit user mapping and assigning physical resources from
a single pool of resources provided by the resource and user abstraction functionalities. This is described by assumption 1. A DG host must provide the handler function for resource mapping and the localuser function for user abstraction. This can be achieved by deploying a component of the DG on the host (as required by assumption 2.). As stated before event functionalities handle the volatility of hosts and manage lost or delayed jobs (required by assumption 3. and 4.). User mapping ensures that a user can use the resources of a host, but cannot access the host directly (required by assumption 5.). In a DG there is no guarantee that hosts are available simultaneously thus, running parallel jobs (e.g., MPI) is mostly infeasible. Conforming to this in $M_{GROUND-DG}$ there is no notion of process, rather jobs consist of a single process and are mapped to physical resources provided by a single host (required by assumption 6.).

The self function is represented by $j \in JOB$ in $M_{GROUND-DG}$ thus, the model must describe the life-cycle of jobs that behave in compliance with the previous statements in this section. The life-cycle is represented by job phases (via the jobPhase function), and described by Rules R1.1 - R1.12.

Based on these a definition for Desktop Grid can be given as follows:

**Statement 2** Rules R1.1-R1.12 together with functionalities F1.1 (resource mapping), F1.2 (user abstraction), F1.3 (delayed jobs) and F1.4 (mapped resources become unavailable) constitute a reference model ($M_{GROUND-DG}$) for distributed applications under the assumptions made for desktop grids in Section 2.2. A desktop grid must provide a service equivalent to functionalities F1.1 (resource mapping), F1.2 (user abstraction), F1.3 (delayed jobs) and F1.4 (mapped resources become unavailable) specified by some reasonable notion of equivalence (e.g., as described in [54]).

### 2.2.7.1 Discussion of existing middleware

In this section we demonstrate the details of definitions by discussing three representative middlewares and their derivatives, six in total: SETI@HOME [55, 56] together with BOINC [57] and SZTAKI Desktop Grid (SZDG) [15]; HTCondor [50]; and XtremWeb-HEP (XW-HEP) [58] together with XtremWeb [59, 48]. Of course, there are many other systems (e.g., OurGrid [60], Charlotte [61], CCOF [62], Paradropper [63], Entropia [47], Javelin [64], Organic Grid [65], Popcorn [66], Saleve [67], GridBee [52], etc.) that are categorized as desktop grids by different surveys (e.g., [4, 5, 49, 42] discussed in Section 2.1.1).

BOINC is a framework for DG and VC and is the generalization of SETI@HOME. SZDG is a fork and extension of BOINC. However, the internal concepts remain unchanged thus, I am going to discuss the three together using BOINC as reference. One of the main results of [1] was that SETI@HOME, despite not aiming to be a grid, is indeed a grid based on the definition of grid by Nemeth et al. I am going to argue that this is still true based on Statement 1. In BOINC there is typically a single user. She has no a priori knowledge about the resources, but still there is a mapping where abstract resource requirements of
the jobs are mapped to the physical resources. The job assignment process is initiated by
the (owners of the) resources. Based on these BOINC provides Functionality F1.1 (resource
mapping). The user has no direct (login) access to the resources, but by downloading the
job of the user the resource owners authorize her to use the resources provided thus, there
is an implicit user abstraction provided realizing Functionality F1.2. Based on these BOINC
conforms Statement 1 and thus, it is a grid. BOINC sets a deadline (report_ deadline) for
each job during resource mapping. When this deadline is passed the job is considered lost
and a new instance is created. There are certain configurable limits (e.g., maximum error
results, maximum total results, etc.) that limit the count of new instance creation for a
job. This realizes Functionality F1.3 (delayed jobs). The BOINC Client (worker) always
prioritizes the owner of the resource, all jobs are executed with below normal process
priorities. The owner of the resource has the possibility to configure the amount of resources
a job can use, and also the condition when the worker is allowed to run, e.g., only when
the computer is idle, in case of laptops only when the computer is plugged in to a power
source, etc. The main idea is that BOINC should be non intrusive, the resource owner
should not notice anything “unusual”. These combined realize Functionality F1.4 (mapped
resources become unavailable) in a very effective way. Based on these both SETI@HOME,
BOINC and SZDG conform Statement 2 thus, they are all desktop grids. They are suited
for deployment in environments characterized in Section 2.1 for desktop grids.

In HTCondor – as discussed in [1] – user abstraction is semantically realized by its
flocking mechanism. This still applies for MGROUND DG as there is an implicit user abstraction
here thus, HTCondor realizes Functionality F1.1. Further condor Starter, condor Startd
and condor_kbd provide matchmaking mechanism for jobs and resources realize resource mapping (Functionality F1.2). Based on these HTCon-
dor conforms Statement 1 thus, it is a grid. HTCondor uses the Classified Advertisements
(ClassAd) mechanism that allows the extensive configuration of the behaviour and char-
acteristics of jobs on different hosts. Each ClassAd is a mapping from an attribute name
(e.g., START, SUSPEND, etc.) to a logic expression. This ClassAd mechanism provides
an “owner state” that any worker can enter if the defined conditions evaluate true (e.g.,
HTCondor registers keyboard activity). In turn this effectively realizes Functionality F1.4.
The extensive configuration possibilities through ClassAd allow HTCondor to handle the
volatility of desktop computers2 and vacate jobs if they are suspended extensively. When
this configuration is set properly it can realize Functionality F1.3. In addition HTCondor
can run a BOINC client as a backfill job. In conclusion HTCondor can be regarded as a
“swiss army knife”: with deep knowledge and the proper configuration it is suited for many
purposes including providing Functionalities F1.3-F1.4 of MGROUND DG. If configured
properly it can be considered as a desktop grid and thus, it is suited for deployment in
environments characterized in Section 2.1 for desktop grids.

XtremWeb is a global computing (GC) platform. Its authors defined GC as an ex-
tension of cycle stealing beyond the frontier of administrative domains. XW-HEP started
as a specialized version of the original XtremWeb. Its developers categorize it as a mem-

\footnote{http://research.cs.wisc.edu/htcondor/manual/current/3_5Policy_Configuration.html# 
SECTION00459400000000000000000000000000, accessed at 2014-10-30}
2.3. Modelling volunteer computing

Models $M_{VC-VOTE}$ and $M_{VC-SPOT}$ are formal models I developed for representing volunteer computing systems: $M_{VC-VOTE}$ represents systems with voting based result certification; and $M_{VC-SPOT}$ represents systems with spot-checking result certification. The models make it possible to evaluate existing systems whether they can be considered as desktop grids and/or volunteer computing systems:

**Theorem I.2.:** $M_{VC-VOTE}$ and $M_{VC-SPOT}$ model variants derived from $M_{GROUND-DG}$ represent formal models narrowed for representing volunteer computing systems that make possible the evaluation of computing infrastructures against

---

volunteer computing systems.

\( M_{\text{VC-VOTE}} \) satisfies the requirements of model \( M_{\text{GRID}} \) since, it corresponds to the desktop grid definition by \( M_{\text{GROUND-DG}} \). As a consequence systems modelled by \( M_{\text{VC-VOTE}} \) are grids as per \( M_{\text{GRID}} \), and desktop grids as per \( M_{\text{GROUND-DG}} \), and correspond to the definition of desktop grids as per Theorem I.1. Similarly it can be shown for \( M_{\text{VC-SPOT}} \) that it also satisfies the same requirements.

### 2.3.1 Assumptions

In the following formal models for volunteer computing (VC) are presented. The following assumptions are made for the models \( (M_{\text{VC-VOTE}} \) and \( M_{\text{VC-SPOT}} \)) based on the environment described in Section 2.1:

1. **Hosts are provided by a large number of individuals rather than by a single organization.**
   These individuals are referred as *donors*. They are usually motivated by the possibility to contribute to science by donating their otherwise idle resources of their computer(s) to Volunteer Computing. The large number of independent donors also increases the heterogeneity of the computing capabilities of the hosts (e.g., varying CPU performances and capabilities, amounts of storage and memory, network bandwidths or operating systems).

2. **Donors should retain full control over the hosts they provide for Volunteer Computing.**
   Although idle resources are used by VC, donors should be able to retain full control over their hosts: they can determine when the worker is allowed to run, to what extent it is allowed to use the resources. Donors should have the possibility to manually intervene in the jobs running on their hosts and suspend and resume the activities of the worker completely.

3. **Donors should be rewarded proportionally for their contributed resources.**
   VC should reward donors for their contributed resources (e.g., processor time). These rewards are usually virtual (e.g., badges, virtual credits or just a summary of the contributed resources) and serves as further motivation for donation of resources and provides a means for donors to compare their achievements to the others’. For example BOINC has several 3rd party sites like BOINCStats/BAM! [68] that track the total earned rewards for different users in all BOINC based Volunteer Computing deployments and provides rankings and different top list charts for donors.

4. **Donated private desktop computers: increased volatility and restricted accessibility compared to desktop grids.**
   Resources are donated by individuals. Although the volatility stems from the cycle-scavenging use case, compared to desktop grids the volatility is assumed to be increased: the donated resources fluctuate rapidly between available and unavailable
states depending on the settings by the donor and her computer usage pattern. Also, home computers are usually behind firewalls and/or routers that are using typically NA(P)T (Network Address [and Port] Translation) for sharing the internet connection between multiple devices making the computer inaccessible from the public internet. The inaccessibility and unavailability characteristics make running traditional HPC workloads like tightly-coupled MPI (Message Passing Interface) and HTC (High Throughput Computing) applications on Volunteer Computing unfeasible, however there are works like [42] that aim to overcome this. The EGI Glossary\(^4\) defines HTC as “A computing paradigm that focuses on the efficient execution of a large number of loosely-coupled tasks. Given the minimal parallel communication requirements, the tasks can be executed on clusters or physically distributed resources using grid technologies.”. Contrary to MTC (Many Task Computing), especially embarrassingly parallel or parameter study problems [7, Section 1.4.4] where there is no dependency between jobs are well suited. In this case the same computation is performed using different input parameters that are usually achieved by partitioning a large parameter space. In these cases for each job a non-changing static part is present: the application that performs the computation, and there is a per job changing part: the input parameters.

5. *The reliability of hosts is lower compared to desktop grids.*

In a volunteer environment the reliability of the hosts vary more than in an institutional one. Some may return erroneous results on purpose or unintentionally due to some hardware or software error. Result certification is used to mitigate this where the final or intermediate results of finished jobs are checked. There are two major approaches as identified in [69] and categorized in [70] as depicted in Figure 2.3. First in *voting* copies of the same job are sent to different hosts (or hosts of different donors) and the results of these (either the final or an intermediate checkpoint) are compared and a final result is selected. Second in *spot-checking* jobs with already known result are sent to workers. The result returned by the worker is compared to the already known one (“oracle”). The first approach is more easy to implement and works well in systems with low failure rate, while the second provides better results with systems that have few idle nodes (for the redundant computing) or higher failure rate [69, pp. 113-115].

The most dominant attribute of Volunteer Computing is the requirement to certify the returned results of the volunteer workers. I present two models for Volunteer Computing each one based on the two major result certification methods. I assume that the certification is based on the final result, but the models can be adapted to incorporate comparison of intermediate or special tasks as well (see Figure 2.3). The first (\(\text{M}_{\text{VC-VOTE}}\)) models systems that use the voting based result certification, while the second one (\(\text{M}_{\text{VC-SPOT}}\)) is intended as a model of VC with the spot-checking based result certification.

---

\(^4\)https://wiki.egi.eu/wiki/Glossary_V2#High_Throughput_Computing
2.3.2 \textit{M}_{VC-VOTE}: model for voting based result certification

2.3.2.1 Universes and signatures

In traditional grids jobs may include different metadata (e.g., input and output filenames, command line parameters, environment variables, list and location of application binaries, etc.) and different binaries like input files and the application binaries. A separation can be introduced between strictly application related components (e.g., application binaries and metadata) and job related components (e.g., input files, command line parameters). This has many benefits, e.g., in case of parameter sweep jobs – where the application remains the same – the application related components can be identified and possibly processed only once by the middleware and not over and over again for each different job.

\[
\begin{array}{c|c}
\text{\textit{M}}_{GROUND-DG} & \text{\textit{M}}_{VC-VOTE} \\
\hline
\text{USER} & \text{USER} \\
\text{JOB} & \text{JOB} \\
\text{APPLICATION} & \text{APPLICATION} \\
\text{UNITOFWORK} & \text{UNITOFWORK} \\
\text{ARESOURCE} & \text{ARESOURCE} \\
\text{NODE} & \text{NODE} \\
\text{UI} & \text{UI} \\
\text{MANAGER} & \text{MANAGER} \\
\text{HOST} & \text{HOST} \\
\text{TASK} & \text{TASK} \\
\text{PRESOURCE} & \text{PRESOURCE} \\
\end{array}
\]

Table 2.8: Basic elements of the ASM model: on the left side the \textit{M}_{GROUND-DG} model, while the \textit{M}_{VC-VOTE} model for Volunteer Computing on the right side.

This separation can be done by introducing an indirection for \textit{JOB}: (a) a job has an application represented by the \textit{application} : \{\text{\textit{TASK}, \textit{JOB}, \textit{UNITOFWORK}}\} \rightarrow \text{\textit{APPLICATION}} monitored function and (b) a unit of work represented by
2.3. Modelling volunteer computing

Figure 2.4: Basic elements of the $M_{VC\text{-}VOTE}$ ASM model.

unitOfWork : $\{\text{TASK, JOB}\} \rightarrow \text{UNITOFWORK}$ monitored function. Abstract resource requirements are split between units of work and applications and are not tied to jobs directly. However, physical resources are still mapped to jobs. An application represented by the APPLICATION universe contains all application related components of a job, excluding all data and metadata that may change in jobs belonging to the application (e.g., command line parameters, input files and additional hardware requirements). Every job must belong to an application. A unit of work represented by the UNITOFWORK universe incorporates all data and metadata that can be specific for a job (e.g., command line parameters or input data). This indirection for JOB is induced by Assumption 4. The TASK universe is expanded: now it embodies the installed job with all its processes and also its application. The DONOR universe is introduced to represent the owners of the hosts donating to the VC project. Each host has a donor and a donor may have multiple hosts. Table 2.8 and Figure 2.4 summarizes the introduced changes in the new model.

Functions modified and introduced This paragraph is intended to collect all modified and newly introduced functions. The more generic ones are detailed here, and some at their places of usage. The following functions are carried over in a modified fashion compared to their corresponding ones in model $M_{GROUND\text{-}DG}$.
The following functions are newly introduced by the model:

\[
\text{application} : \{\text{TASK, JOB, UNITOFWORK}\} \rightarrow \text{APPLICATION} \quad (2.46)
\]
\[
\text{unitOfWork} : \{\text{TASK, JOB}\} \rightarrow \text{UNITOFWORK} \quad (2.47)
\]
\[
\text{initialInstances} : \text{JOB} \rightarrow \text{NUMBER} \quad (2.48)
\]
\[
\text{role} : \text{JOB} \times \{\text{abstract, instance}\} \rightarrow \{\text{true, false}\} \quad (2.49)
\]
\[
\text{resourceRequest} : \{\text{APPLICATION, UNITOFWORK}\} \times \text{ARESOURCE} \rightarrow \{\text{true, false}\} \quad (2.50)
\]
\[
\text{parent} : \text{JOB} \times \text{JOB} \rightarrow \{\text{true, false}\} \quad (2.51)
\]
\[
\text{consensus} : \text{JOB} \rightarrow \{\text{init, none, success, failure}\} \quad (2.52)
\]
\[
\text{reward} : \text{DONOR} \times \text{JOB} \rightarrow \{\text{true, false}\} \quad (2.53)
\]
\[
\text{readyForConsensus} : \text{JOB} \rightarrow \{\text{true, false}\} \quad (2.54)
\]

Jobs have dual roles in the M\textsubscript{VC−VOTE} model, represented by the role dynamic function. First abstract jobs \(\text{role}(j, \text{abstract}) = \text{true}\), see Equation 2.49) represent the jobs users submit to the volunteer computing project similarly to the jobs in previous (desktop) grid models. However these jobs do not get executed directly on hosts rather they serve as templates and their instances are mapped to hosts. Such job instances are represented by jobs that have \(\text{role}(j, \text{instance}) = \text{true}\) set. A job has either the role of instance or abstract it cannot have both. The abstract/instance job roles allow the replication of jobs in case a volunteer host returns an erroneous result. They also allow the introduction of a census mechanism that is able to form a consensus based on the results of multiple instances of the same job, resulting in either success or failure final phase for the abstract job the instances belong to. initialInstances (see Equation 2.48) is a static function it determines how many instances for the abstract job will be created upon submission. The parent (see Equation 2.51) function describes the relationship between abstract and instance jobs. It returns the abstract job the instance belongs to. The resourceRequest (see Equation 2.50) function determines the abstract resources requested by the application and unit of work part of the job separately. However these requests must be satisfied still from the same host. Also the physical resource usage \(\text{uses} : \text{JOB} \times \text{PRESOURCE} \rightarrow \{\text{true, false}\}\) function does not change since the resources are used together by the application and unit of work. Neither of them can use the resources (i.e., run) alone. consensus (see Equation 2.52) is a dynamic function. For each job instance it is set to init upon creation, none when the job finishes and failure if it fails. Consensus is only evaluated for jobs with
2.3. Modelling volunteer computing

*consensus none* and the outcome of it is either *success* or *failure*. It is possible that a job finishes successfully however it fails the evaluation. It is a job specific task (or application specific one if it is coming from a repository) and thus it is handled for each abstract job independently (see Functionality F2.3).

The EVENT (see Equation 2.56) set is extended with events generated by the donor, the *eventMasked* (see Equation 2.57) function now also includes the entity that masked the event. The *isEventMasked* (see Equation 2.58) convenience monitored function is introduced that tells whether the given signal for the given job is masked by any means.

\[
\text{MASKENTITY} = \{\text{donor, host}\} \quad (2.55)
\]

\[
\text{EVENT} = \{\text{start, stop, suspend, resume, abort, terminate, ckpt, donorSuspend, donorResume, donorStart, donorStop, donorAbort}\} \quad (2.56)
\]

\[
\text{eventMasked} : \text{JOB} \times \text{EVENT} \times \text{MASKENTITY} \to \{\text{true, false}\} \quad (2.57)
\]

\[
\text{isEventMasked} : \text{JOB} \times \text{EVENT} \to \{\text{true, false}\} \quad (2.58)
\]

\[
\text{maskEvents} : \text{JOB} \times 2^{\text{EVENT}} \times \text{MASKENTITY} \to \{\text{true, false}\} \quad (2.59)
\]

The event *donorSuspend* represents the event when a donor wants to suspend the execution of a single job instance, and *donorResume* lifts this. The event *donorStop* signals that the donor wants to stop all activities of the VC worker. While *donorSuspend* targets a single job instance, *donorStop* targets all job instances on a host owned by the donor. Also *donorStop* causes that all jobs are stopped and all other activities are suspended that affect the host. *donorAbort* signals that the donor wants to abort the execution of a given job instance running on her host.

The *eventMasked* and *maskEvents* (see Equations 2.57 and 2.58) functions are modified that they include the entity that masks the event. These entities are represented by the set *MASKENTITY* (see Equation 2.55). It is either the donor or the host the job is running on. This latter represents the case when the used resources of the host become unavailable (see Functionality F2.4), while the donor entity represents donor interaction through *donor* events. An event remains masked from a job (i.e., the job will not receive that event) as long as a single entity masks it. Meaning if an event is masked by both *donor* and *host*, the job won’t receive the event until it is unmasked by both entities.

2.3.2.2 Initial state

In the following we define the initial states of the model \(M_{VC-\text{VOTE}}\):

1. Initial state rules of \(M_{\text{GROUND-DG}}\): 2.32-2.43.

2. All jobs must have an application:

\[
\forall j \in \text{JOB}, \exists \text{app} \in \text{APPLICATION} : \quad \text{application}(j) = \text{app} \quad (2.60)
\]
2.3.2.3 Functionalities

The Resource and user mapping functionalities (see Functionalities F2.1-F2.2) are refinements of the resource mapping and user abstraction of $M_{GROUND-DG}$ (see Functionalities F1.1-F1.2). Job certification and donor rewards functionalities are provided by the accordingly named functionality in the model (see Functionality F2.3). The event functionalities introduced in $M_{GROUND-DG}$ are used here in a modified manner (see Functionalities F2.4-F2.5), and also a new one is introduced that handles the interaction of the donor (see Functionality F2.6) and creates events that are interpreted by the job phase transition rules.

**Functionality F2.1: Resource mapping**

The following represents the resource mapping functionality:

```plaintext
if (\exists a_1, a_2 \in \text{AResource}, \exists h \in \text{Host},
\exists j \in \text{Job}, \exists uow \in \text{UnitOfWork}, \exists \text{app} \in \text{Application}) :
    \text{role} (j, \text{instance}) = \text{true} \land \text{application} (j) = \text{app} \land
    \text{unitOfWork} (j) = \text{uow} \land \text{mappedResource} (\text{uow}, a_1) = \text{undef} \land
    \text{mappedResource} (\text{app}, a_2) = \text{undef} \land \text{resourceRequest} (\text{uow}, a_1) = \text{true} \land
    \text{resourceRequest} (\text{app}, a_2) = \text{true} \land \text{event} (j) \neq \text{abort}
then
    \text{choose} \ p_1, p_2 \text{ in PResource satisfying}
    \text{compatible} (\text{attr} (a_1), \text{attr} (p_1)) \land \text{belongsTo} (p_1, h) = \text{true} \land
    \text{compatible} (\text{attr} (a_2), \text{attr} (p_2)) \land \text{belongsTo} (p_2, h) = \text{true} \land
    \text{available} (h) = \text{true} \land \text{acceptJobs} (h) = \text{true}
    \text{mappedResource} (\text{uow}, a_1) := p_1
    \text{mappedResource} (\text{app}, a_2) := p_2
endchoose
endif
```

Listing 2.17: Resource mapping functionality

In the $M_{GROUND-DG}$ model the job has an abstract resource requirement (represented by the universe ARESOURCE). The job is split into an application and a unit of work part and the resource requirements are also split between them. A host must be available and must be able to provide physical resources both for the application and unit of work part of the job. However physical resources for the job and unit of work part are still mapped to the job itself. This is for simplifying the mapping for this model.

**Functionality F2.2: User abstraction**

This functionality is equivalent to F1.2 of $M_{GROUND-DG}$. There is no difference in user mapping. All users should have similar access to the resources provided by the desktop grid:

\[ F2.2 \equiv F1.2 \]

**Functionality F2.3: Result certification**

To eliminate the effect caused by unnoticed erroneous results returned by volunteer resources a check is required that is able to identify and filter out these. This is the responsibility of the consensus functionality. The guard executes the consensus forming for all successful finished job instances that have enough ‘siblings’ (instances belonging to the same abstract job). The consensus is formed for the given instance based on all available instances. The readyForConsensus (see Equation 2.54) abstract function determines if
the consensus forming procedure can start (e.g., there are enough siblings available to start the process, either by reaching a threshold for the number of non-checked jobs or having one already checked considered as a reference). The consensus is formed by the consensus function (see Equation 2.52) abstract function. If the job is considered successful the donor who completed the job should be rewarded by some means. This is represented by the reward: DONOR × JOB → \{true, false\} abstract function. The guard $jobOutcome(job) = success$ guarantees that only final results are checked. For intermediate or special results this guard needs to be modified.

The exact specifics of forming a consensus and the means of rewarding donors are the internals of the given volunteer computing middleware (i.e., the exact details of the $formConsensus$ function). The requirement is that each VC middleware should provide this functionality with reasonable refinement of the abstract functions.

**Functionality F2.4:** Mapped resources become unavailable

This functionality is basically the same as F2.4 of $M_{GROUP-DG}$. The only difference is that in $M_{VC-VOTE}$ only job instances are affected by the unavailability of mapped resources. However the $task(job) = t$ guard guarantees this, since only job instances may have tasks. The only modification required is caused by the modified eventMasked function: eventMasked(job,<EVENT>) is replaced by eventMasked(job,<EVENT>,host), thus I will omit the listing.

**Functionality F2.5:** Delayed jobs

This functionality is adapted from $M_{GROUP-DG}$. Only job instances can be delayed, thus the functionality must be modified to reflect this:

Listing 2.18: Functionality F2.5: Example for delayed job handling

**Functionality F2.6:** Donor interaction

This functionality handles events originating from the donor and generates events that are interpreted by the job phase transition rules of the model. Resume/ start types of donor events do not generate their normal pairs, just the restriction is lifted (e.g., it is up to the scheduling policy of the VC middleware when to start the jobs).
let \( MEV\) \( ENTS \) := \{start, resume, ckpt\}
if \( (\exists \) job \( \in \) \( JOB \) ) : role(job, instance) = true \( \land \)
\( event(j\) ob) = \{donorSuspend|donorResume|donorStart|donorStop|donorAbort\} then
if \( (event(j\) ob) = \) donorSuspend then
maskEvents(job, \( MEV\) \( ENTS \), donor) := true
\( event(j\) ob) := suspend
endif
if \( (event(j\) ob) = \) donorResume then
maskEvents(job, \( MEV\) \( ENTS \), donor) := false
endif
if \( (event(j\) ob) = \) donorStop then
maskEvents(job, \( MEV\) \( ENTS \cup \{\) donorResume\} , donor) := true
\( event(j\) ob) := stop
endif
if \( (event(j\) ob) = \) donorStart then
maskEvents(job, \( MEV\) \( ENTS \cup \{\) donorResume\} , donor) := false
endif
if \( (event(j\) ob) = \) donorAbort then
maskEvents(job, \( MEV\) \( ENTS \), donor) := true
\( event(j\) ob) := abort
endif
endif
Listing 2.19: Functionality F2.6: Donor events handling

Figure 2.5: Simplified job phase transition diagram for the \( MV\) \( C-VOTE \) model.
2.3. Modelling volunteer computing

2.3.2.4 Job phase transition rules

Figure 2.5 shows the rules and corresponding job phases in $M_{VC-VOTE}$. Job phases are distinguished by job roles (i.e., abstract and instance). This is for better presentation only as the two sub parts (separated by a dashed line) can be mapped onto each other. However this would make the figure a bit convoluted. Rules R2.1-R2.6 are for abstract jobs and Rules R2.8-R2.17 detail the behavior of job instances in $M_{VC-VOTE}$.

**Rule R2.1: Job submitted**
An abstract job is always submitted (by a user) using an user interface connected to the manager of the VC project (represented by the submitted function) and not created internally by the system. This user/ system origin characteristics is represented by the role function:

\[
\text{if} \left( \exists \text{mgr} \in \text{MANAGER}, \exists u \in \text{UI}, \exists us \in \text{USER} : \text{submitted}(j, us, u, \text{mgr}) = \text{true} \right) \land \text{jobPhase}(j) = \text{undef} \text{ then } \text{jobPhase}(j) := \text{init} \text{ endif}
\]

Listing 2.20: Rule R2.1: Job submitted

The model does not handle the case when a job fails at submission for some reason (it is assumed that a submission never fails, thus submitted function is always true).

**Rule R2.2: Abstract job ready to run**
An abstract job is ready to run if all its instances are created and initialized.

\[
\text{if} \left( \text{jobPhase}(j) = \text{init} \land \text{role}(j, \text{abstract}) = \text{true} \right) \text{ then } \forall 1 \leq k \leq \text{initialInstances}(j) \text{ do extend JOB by child with } \begin{align*}
\text{parent}(\text{child}, j) & := \text{true} \\
\text{role}(\text{child}, \text{instance}) & := \text{true} \\
\text{application}(\text{child}) & := \text{application}(j) \\
\text{unitOfWork}(\text{child}) & := \text{workunit}(j) \\
\text{jobPhase}(\text{child}) & := \text{init}
\end{align*} \text{ endextend} \text{ endif}
\]

Listing 2.21: Rule R2.2: Abstract job ready to run

**Rule R2.3: Start abstract job**
Enter running phase if one of the instances enters running phase. It will remain running even if its instance is stopped.

\[
\text{if} \left( \text{jobPhase}(j) = \text{readytorun} \land \text{role}(j, \text{abstract}) = \text{true} \land \text{event}(j) \neq \text{abort} \right) : \exists \text{child} \in \text{JOB}: \text{parent}(\text{child}, j) := \text{true} \land \text{role}(\text{child}, \text{instance}) = \text{true} \\
\land \text{jobPhase}(\text{child}) = \text{running} \text{ then } \text{jobPhase}(j) := \text{running}
\]

Listing 2.22: Rule R2.3: Start abstract job

**Rule R2.4: Abstract job successful**
If there is a successful and valid job instance the abstract job is considered successful.
Chapter 2. Formal models for desktop grids and volunteer computing

if (jobPhase(j) = running ∧ role(j, abstract) = true):
    ∃jchild ∈ JOB: parent(jchild, j) = true ∧ role(jchild, instance) = true ∧
    jobOutcome(jchild) = success ∧ consensus(jchild) = success then
        jobOutcome(j) := success
        jobPhase(j) := finished
endif

Listing 2.23: Rule R2.4: Abstract job successful

Rule R2.5: Abstract job failed
An abstract job has failed if (i) it has no more non-finished instances; and (ii) none of them are valid and (iii) there are no more instances required.

if (jobPhase(j) = running ∧ role(j, abstract) = true):
    ¬∃jj ∈ JOB: parent(jj, j) = true ∧ role(jj, instance) = true ∧
    (consensus(jj) ≠ failed ∨ jobPhase(jj) ≠ finished) then
        jobOutcome(j) := failed
        jobPhase(j) := finished
endif

Listing 2.24: Rule R2.5: Abstract job failed

Rule R2.6: Abort abstract job
On abort event abort all instances and the abstract job itself:

if event(j) = abort ∧ role(j, abstract) = true ∧ jobPhase(j) ≠ {undef | init}
    then
        jobPhase(j) := finished
        jobOutcome(j) := failed
        forall jj in JOB: parent(jj, j) = true ∧ role(jj, instance) = true do
            event(jj) := abort
        endforall
endif

Listing 2.25: Rule R2.6: Abort abstract job

Rule R2.7: Clean up after abstract job
An abstract job can be cleaned after it is finished, and also all its instances have finished and the submitting user allows the deletion. This user approval/ request is represented by the readyForDelete monitored function introduced in MGROUND-DG.

if (jobPhase(j) = finished ∧ role(j, abstract) = true ∧
    readyForDelete(j) = true) ∧ ¬∃jj ∈ JOB: parent(jj, j) = true ∧
    role(jj, instance) = true ∧ jobPhase(jj) ≠ finished then
    forall jj ∈ JOB: parent(jj, j) = true ∧ role(jj, instance) = true do
        JOB(jj) := undef
    endforall
    JOB(j) := undef
endif

Listing 2.26: Rule R2.7: Clean up after abstract job

Rule R2.8: Job staging
This rule is a refined version of R1.1. Only job instances can be mapped to physical resources and abstract resource requests belong to the application and workunit part of the job instead of itself:
2.3. Modelling volunteer computing

let \( p_1 = \text{mappedResource}(wu, ar_1) \)
let \( p_2 = \text{mappedResource}(app, ar_2) \)
if \( (\exists ar_1, ar_2 \in ARESOURCE, \exists wu \in UNITOFWORK, \exists app \in APPLICATION, \exists jj \in JOB) : \)
\( \text{role}(j, \text{instance}) = \text{true} \land \text{application}(j) = app \land \)
\( \text{workunit}(j) = wu \land \text{parent}(j, jj) = \text{true} \land \text{jobPhase}(jj) = \text{readytorun} \land \)
\( \text{wuResourceRequest}(wu, ar_1) = \text{true} \land \text{appResourceRequest}(app, ar_2) = \text{true} \land \)
\( p_1 \neq \text{undef} \land p_2 \neq \text{undef} \land \text{usermapping}(\text{globaluser}(j), p_1) \neq \text{undef} \) then
\( \text{mapped}(j) := \text{location}(p_1) \)
\( \text{installed}(\text{task}(j), \text{location}(p_1)) := \text{true} \)
\( \text{resourceRequest}(wu, ar_1) := \text{false} \)
\( \text{resourceRequest}(app, ar_2) := \text{false} \)
endif

Listing 2.27: Rule R2.8: Resource selection

The \( \text{jobPhase}(jj) = \text{readytorun} \) guard ensures that no job instance is started before all instances are created.

Rule R2.9: Task installed and physical resources allocated

if \( (\exists h \in HOST, \exists pr \in PRESOURCE, \exists wu \in UNITOFWORK, \exists app \in APPLICATION, \exists jj \in JOB) : \)
\( \text{mapped}(j) \neq \text{undef} \land \text{uses}(j, p_1) = \text{true} \land \text{jobPhase}(j) = \text{init} \land \)
\( \text{role}(j, \text{instance}) = \text{true} \land \text{parent}(j, jj) = \text{true} \land \text{jobPhase}(jj) = \text{readytorun} \) then
extend \text{TASK} by \( t \) with
\( \text{task}(j) := t \)
\( \text{checkpoint}(j) := \text{false} \)
\( \text{installed}(t, \text{location}(p_1)) := \text{true} \)
endextend
\( \text{jobPhase}(j) := \text{readytorun} \)

Listing 2.28: Rule R2.9: Task installed and physical resources allocated

This rule is a refined version of R1.3. The job instance is ready to run after the required resources are selected (see Rule R2.8) and the job is mapped and installed on a host.

Rule R2.10: Start job instance

Job instances that are in the \( \text{readytorun} \) phase can be started by an internal event or by the donor. The start is denoted by the task (internal) or job (external) event \( \text{start} \). This is described by the following rule:

if \( (\exists t \in \text{TASK}) : \text{task}(j) = t \land \text{role}(j, \text{instance}) = \text{true} \land \text{jobPhase}(j) = \text{readytorun} \land \)
\( \text{jobPhase}(j) = \text{readytorun} \land \text{event}(j) = \text{start} \land \text{checkpoint}(t) = \text{false} \) then
\( \text{uses}(j, \text{presource}(j)) := \text{true} \)
\( \text{jobPhase}(j) := \text{running} \)
endif

Listing 2.29: Rule R2.10: Start job instance

Rule R2.11: Stop job instance

A job instance can be stopped by the \( \text{stop} \) event. This rule is a refined version of R1.5.

if \( (\exists t \in \text{TASK}) : \text{task}(j) = t \land \text{role}(j, \text{instance}) = \text{true} \land \)
\( \text{jobPhase}(j) = \text{running} \land \text{event}(j) = \text{stop} \land \text{checkpoint}(t) = \text{false} \) then
\( \text{uses}(j, \text{presource}(j)) := \text{false} \)
\( \text{jobPhase}(j) := \text{readytorun} \)
endif

Listing 2.30: Rule R2.11: Stop job instance
Rule R2.12: Suspend job instance
Running job instances can be suspended by the event `suspend`. This rule is a refined version of R1.6.

\[
\text{if } \left( \text{jobPhase}(j) = \{ \text{running}, \text{readytorun} \} \land \text{role}(j, \text{instance}) = \text{true} \land \text{suspended}(j) = \text{false} \land \text{event}(j) = \text{suspend} \right) \text{ then}
\]
\[
\text{uses}(j, \text{presource}(j)) := \text{false}
\text{ suspended}(j) := \text{true}
\]
\[
\text{endif}
\]

Listing 2.31: Rule R2.12: Suspend job instance

Rule R2.13: Resume job instance
Suspended job instances can be resumed by the event `resume`. This rule is a refined version of R1.7.

\[
\text{if } \left( \text{role}(j, \text{instance}) = \text{true} \land \text{suspended}(j) = \text{true} \land \text{event}(j) = \text{resume} \right) \text{ then}
\]
\[
\text{uses}(j, \text{presource}(j)) := \text{true}
\text{ suspended}(j) := \text{false}
\]
\[
\text{endif}
\]

Listing 2.32: Rule R2.13: Resume job instance

Rule R2.14: Job fails or abort
A job instance receiving the `abort` event represents a failure or an abort request. This rule is a refined version of R1.9.

\[
\text{if } \left( \text{role}(j, \text{instance}) = \text{true} \land \text{event}(j) = \text{abort} \right) \text{ then}
\]
\[
\text{uses}(j, \text{presource}(j)) := \text{false}
\text{ jobPhase}(j) := \text{finished}
\text{ jobOutcome}(j) := \text{failed}
\text{ if } \left( \exists t \in \text{TASK} : \text{task}(j) = t \right) \text{ then}
\]
\[
\text{TASK}(t) := \text{undef}
\]
\[
\text{endif}
\text{ endif}
\]

Listing 2.33: Rule R2.14: Job fails

Rule R2.15: Perform checkpointing
Checkpoint requests are signaled by the event `ckpt`. This rule is a modified version of R1.10.

\[
\text{if } \left( \exists t \in \text{TASK} : \text{task}(j) = t \land \text{event}(t) = \text{ckpt} \land \text{capability}(j, \text{ckpt}) = \text{true} \land \text{readyToCheckpoint}(j) = \text{true} \land \text{createCheckpoint}(j) = \text{true} \land \text{role}(j, \text{instance}) = \text{true} \right) \text{ then}
\]
\[
\text{checkpoint}(j) := \text{true}
\]
\[
\text{endif}
\]

Listing 2.34: Rule R2.15: Perform checkpointing

Rule R2.16: Job termination
The internal event `terminate` denotes job completion without failure. It is described by the following rule:
2.3. Modelling volunteer computing

\[
\text{if } (\text{jobPhase}(j) = \text{running } \land \text{event}(j) = \text{terminate } \land \\
\text{role}(j, \text{instance}) = \text{true } \land \text{event}(j) = \text{terminate} \\
\text{role}(j, \text{instance}) = \text{true} ) \text{ then} \\
\begin{align*}
\text{uses}(j, \text{presource}(j)) & := \text{false} \\
\text{jobPhase}(j) & := \text{finished} \\
\text{jobOutcome}(j) & := \text{success} \\
\text{TASK}(\text{task}(j)) & := \text{undef}
\end{align*}
\]

Listing 2.35: Rule R2.16: Job termination

**Rule R2.17: Resume from checkpoint**

A resume from checkpoint happens if a job is in \textit{readytorun} phase, it receives the \textit{start} event and there is an existing checkpoint for the job. If the resume from checkpoint fails the task should be restarted. This rule is a modified version of R1.12.

\[
\text{if } (\text{role}(j, \text{instance}) = \text{true } \land \text{jobPhase}(j) = \text{readytorun} \land \text{event}(j) = \text{start} \\
\text{checkpoint}(j) = \text{true} ) \text{ then} \\
\begin{align*}
\text{uses}(j, \text{presource}(j)) & := \text{true} \\
\text{resumeCheckpoint}(j) & := \text{true} \\
\text{checkpoint}(j) & := \text{false} \\
\text{jobPhase}(j) & := \text{running}
\end{align*}
\]

Listing 2.36: Rule R2.17: Resume from checkpoint

2.3.3 Correspondence of \(M_{\text{GROUND-DG}}\) and \(M_{\text{VC-VOTE}}\) ASM models

2.3.3.1 Notion of completeness and equivalence

ASM allows defining the precise meaning of equivalence of models [54]. There are two main approaches at one’s disposal. In the first case equivalence is based on inputs and outputs and time complexity. In the second case equivalence is expanded to (some) internal states of the system. Here no observer of the execution should be able to detect any difference between them. There can be different interpretations on what parts of the execution the observer is allowed to witness.

The basic idea behind the proof of equivalence is to decompose the diagram into sub diagrams or computational segments. The edges connecting states represent an arbitrary relation characterized by the coupling invariant [71]. This allows reducing the correspondence between two ASM runs to correspondence of given computational segments. The notion of refinement in general can be defined as follows:

**Definition 1** Each ASM model \(M = (S, I, \rho)\) consists of a set \(S\) of possible states, a set of \(I \subseteq S\) initial states and a state transition relation \(\rho : S \times S\).

**Definition 2** A refinement of a ASM model \(M^A = (S^A, I^A, \rho^A)\) to another ASM model \(M^B = (S^B, I^B, \rho^B)\) is given by the relation \(\text{INPUT} : I^A \times I^B\) on initial states and a relation \(\text{OUTPUT} : F^A \times F^B\) on final states.

A general scheme for refinement is shown in Figure 2.6. Refinement based on this scheme establishes the standard notion of simulation used for verifying data refinements. For a refinement of ASM \(M^A\) to ASM \(M^B\) the following can (and has to) be defined based on [72]:
- a notion of refined state;
- a notion of states of interest and correspondence between ASM $M^A$-states $st$ and ASM $M^B$ states $st^*$ of interest, the pairs of states one wants to relate through the refinement, including usually the initial and final states;
- notion of abstract computation segments $\tau_1, ..., \tau_m$ where $\tau_i$ represents a single $M^A$ step, and of corresponding refined computing segments $\sigma_1, ..., \sigma_n$, of single $M^B$ steps $\sigma_j$, which in turn lead from corresponding states of interest to the next corresponding states of interest, resulting in $(m, n)$-diagrams and the refinements $(m, n)$-refinements;
- and notion of equivalence ($\equiv$) of corresponding states of interest.

**Figure 2.6**: ASM refinement scheme, where $\equiv$ denotes equivalence of states of interest in corresponding states.

Based on these the notion of correct and complete refinement are defined as follows [72]:

**Definition 3** An ASM $M^B$ is called a correct refinement of an ASM $M^A$ if and only if for each $M^B$ run $S_0^*, S_1^*, ...$ there is an $M^A$ run $S_0^*, S_1^*, ...$ and sequences $i_0 < i_1, ..., j_0 < j_1 < ...$ such that $i_0 = j_0 = 0$ and $S_k^* \equiv S_{i_k}^*$ for each $k$, and either
- both runs terminate and their final states are the last pair of equivalent states, or
- both runs and both sequences $i_0 < i_1, ..., j_0 < j_1 < ...$ are infinite.

Let $M^A \Rightarrow M^B$ denote a correct refinement.

**Definition 4** $M^B$ is called a complete refinement of $M^A$, in short $M^A \triangleleft M^B$, if and only if $M^A$ is a correct refinement of $M^B$.

A correct and complete refinement can be referred as bisimulation.

Further the commuting diagram in Figure 2.6 can be decomposed into more simplistic diagrams satisfying an invariant $\approx$. These are the following and are shown on Figure 2.7 based on [72]:
- (m,0)-triangles: computation run where only the abstract run make progress performing a positive number \( m \) of steps to reach an \( st' \approx st^* \),

- (0,n)-triangles: computation run where only the concrete run make progress performing a positive number \( n \) of steps to reach an \( st'' \approx st \),

- (m,n)-trapezoids: representing a computation segment which leads \( m > 0 \) steps to an \( st' \) and in \( n > 0 \) steps to an \( st'' \) such that \( st' \approx st'' \).

Figure 2.7: ASM refinement component diagrams: (m,n)-trapeziod; (0,n)-triangle; and (m,0)-triangle

Based on the definition of the previous items, a comparison can be performed and the equivalence can be proven by applying Definition 3.

### 2.3.3.2 Proof of correspondence

Correspondence of \( M_{\text{GROUND-DG}} \) and \( M_{\text{VOTE}} \) is shown next. First the visible states for the observer need to be defined. \( M_{\text{GROUND-DG}} \) has several job related functions (e.g., mapped, canUse, etc.), but obviously the most characteristic function is \( \text{jobPhase} \). Every operation of each job is tied to specific \( \text{jobPhase} \), so the state of the job agent in \( M_{\text{GROUND-DG}} \) is expressed by its \( \text{jobPhase} \):

**Definition 5** The state \((S)\) of the job agent \( j \) in \( M_{\text{GROUND-DG}} \) is expressed by the value of \( \text{jobPhase} \) of \( j \) with no additional constraints as follows:

\[
M_{\text{GROUND-DG}} : S^A(j) := \text{jobPhase}(j) \tag{2.61}
\]

In \( M_{\text{VOTE}} \) roles are introduced. Users submit abstract jobs. From these the VCS creates instance(s) and dispatches them to hosts for processing. For that reason semantically a job in \( M_{\text{VOTE}} \) consists of an abstract part and at least of a single instance part. Multiple instances are used for result certification, however based on \( M_{\text{GROUND-DG}} \) desktop grids do not have the notion of result certification, so a single instance (with its abstract part) looks like the most similar to a job in \( M_{\text{GROUND-DG}} \) thus:

\[
\text{initialInstances}(j) = 1 \tag{2.62}
\]

basically the abstract part is for the user who submitted the job, and the instance is for the donor processing it in \( M_{\text{VOTE}} \). Based on these the state of the job agent in \( M_{\text{VOTE}} \) is expressed as follows:
Definition 6 Let \( j \) denote an abstract job and \( j_i \) denote its instance whenever it exists. The state \( (S) \) of the job agent \( j \) in \( M_{\text{GROUND-DG}} \) is expressed by the value of \( \text{jobPhase} \) of job \( j \) as follows:

\[
M_{\text{VC-VOTE}} : S^B(j) := \begin{cases} 
\text{jobPhase}(j_i) & \text{if } \exists j_i \in \text{JOB} : \text{parent}(j_i) = j \\
\text{jobPhase}(j) & \text{otherwise} 
\end{cases}
\]  

(2.63)

Applying Definition 6 to \( M_{\text{GROUND-DG}} \) yields Definition 5. The \textit{role} and \textit{initialInstances} functions (see 2.48 and 2.49) in Definition 6 are parameters of the agent in \( M_{\text{VC-VOTE}} \). In \( M_{\text{GROUND-DG}} \) these functions are not defined, thus there is no need for such constraints in Definition 5. We assume that the result certification functionality of \( M_{\text{VC-VOTE}} \) (see F2.3) accepts successful finished jobs (\( \text{jobOutcome}(job) = \text{success} \)) as there is no such functionality in \( M_{\text{GROUND-DG}} \):

\[
\text{formConsensus} : \text{JOB} \rightarrow \{\text{success}\}
\]  

(2.64)

Event functionalities (e.g., Donor interaction, Delayed jobs or Mapped resources become unavailable) interact indirectly through events (e.g., \text{abort}) with the model and thus, have no direct influence on the job states. Nevertheless, these are assumed to be disabled.

Next the states of interests need to be defined:

Definition 7 The following states denote the states of interest for models \( M_{\text{GROUND-DG}} \) and \( M_{\text{VC-VOTE}} \):

\[
S^{\{A|B\}}_{\text{INTEREST}} := \{\text{SIN}, \text{SI}, \text{SPE}, \text{SE}, \text{SF}, \text{ST}\}
\]  

(2.65)

The states defined in Definition 7 represent the following:

- \text{SIN}: Initial state of the model. All visible jobs (per Definition 5-6) are \( \text{JOB}(j) = \text{undef} \) or \( \text{jobPhase}(j) = \text{undef} \). (Refer to Section 2.3.2.2 for more details.)
- \text{SI}: All visible jobs (per Definition 5-6) \( j \) are in init state: \( S^{\{A|B\}}(j) = \text{init} \).
- \text{SPE}: Pre-execution state. All visible jobs (per Definition 5-6) \( j \) are ready to run: \( S^{\{A|B\}}(j) = \text{readytorun} \).
- \text{SE}: Execution state. All visible jobs (per Definition 5-6) \( j \) are running: \( S^{\{A|B\}}(j) = \text{running} \).
- \text{SF}: Finished state. All visible jobs (per Definition 5-6) \( j \) have finished: \( S^{\{A|B\}}(j) = \text{finished} \).
- \text{ST}: Terminated state. All visible jobs (per Definition 5-6) \( j \) have terminated and were discarded: \( \text{JOB}(j) = \text{undef} \).
2.3. Modelling volunteer computing

When a state (e.g., $SI$ in Figures 2.8-2.14) of a model is in relation with a state of another model, they are referred as corresponding states. Computation segments represent a sequence of steps that lead from a corresponding state of interest (states of interests are represented as shaded nodes in Figures 2.8-2.14) to a next corresponding state of interest. The following computational segments are defined:

- $SEG_{INITIALS}$ segment represents the path $SIN \rightarrow SI$ for the related states.
- $SEG_{INIT}$ segment represents the path $SI \rightarrow SPE$.
- $SEG_{START}$ segment represents the path $SPE \rightarrow SE$.
- $SEG_{STOP}$ segment represents the path $SE \rightarrow SPE$.
- $SEG_{EXEC}$ segment represents the path $SE \rightarrow SF$.
- $SEG_{TERM}$ segment represents the path $SF \rightarrow ST$.
- $SEG_{ABORT}$ segment represents the path $\{SI, SPE, SE\} \rightarrow SF$ caused by an abort event.

Trace of a normal execution is shown in Figures 2.8-2.12, while Figure 2.13 shows the state transition for the abort sequence of an abstract job in $M_{VC-VOTE}$. Figure 2.14 depicts an abort event (job abort or failure) for a job instance. Corresponding states in the two models are connected with dashed lines and are represented as shaded nodes. The asterisk sign (*) denotes that a state transition or segment can occur $\geq 0$ times in a trace.

State transition diagrams (see Figures 2.8-2.14) include functionalities for User abstraction, Resource abstraction and Result certification (for $M_{VC-VOTE}$), as these affect directly the state of the jobs. However, event functionalities (e.g., Donor interaction, Delayed jobs or Mapped resources become unavailable) generate events (e.g., abort) that the job phase transition rules interpret and do not affect the states directly thus, they are not needed to be included in the diagrams.

Figure 2.8 describes the job submission ($SEG_{INITIALS}$) and pre-execution ($SEG_{INIT}$) phases of a normal execution. On top the steps in $M_{GROUND-DG}$ and on the bottom the steps and corresponding states of $M_{VC-VOTE}$ are shown. For each transition the involved rule or functionality is shown. For $SEG_{INITIALS}$ (see Figure 2.8/a) each model a job starts as undefined. In $M_{GROUND-DG}$ Rule R1.2 represents the job submission that transitions a job from $SIN$ to $SI$. In $M_{VC-VOTE}$ this involves two steps, first the abstract job is created (Rule R2.1) and then an instance for the abstract job is created (Rule R2.2). In both cases the end state is $SI$. For $SEG_{INIT}$ (see Figure 2.8/b) in $M_{GROUND-DG}$ four steps are required to get to the pre-execution phase. First the resource mapping functionality (Functionality F1.1) allocates resources for the job. Next the user mapping functionality (Functionality F1.2) provides access to the resources. The third step involves job staging (Rule R1.1): if a resource a host provides is actually available for the desktop grid, the job is mapped. Finally Rule R1.3 installs the physical representation of the job on the host and the job enters $SPE$ phase. Similarly in $M_{VC-VOTE}$ for $SEG_{INIT}$: $SI \xrightarrow{F2.1} SV_{I2} \xrightarrow{F2.2} SV_{I3} \xrightarrow{R2.8} SV_{I4} \xrightarrow{R2.9} SPE$. 
Figure 2.8: Computation segments SEG\text{INITIALS} (a.) and SEG\text{INIT} (b.) for normal executions in M_{\text{GROUND-DG}} and in M_{\text{VC-VOTE}} with corresponding states.

Figure 2.9 details the SEG\text{START} segment. SEG\text{START} represents the job start sequence (SPE $\rightarrow$ SE). For M_{\text{GROUND-DG}} Rules R1.6 and R1.7 are suspend and resume events that any job can receive that is either in SPE or SE states (similarly Rules R2.12 and R2.13 in M_{\text{VC-VOTE}} for job instances). Rule R1.4 represents the job start (Rule R2.10 in M_{\text{VC-VOTE}} respectively) and Rule R1.12 the resume from checkpoint (Rule R2.17 in M_{\text{VC-VOTE}} respectively) both leading to SE state.

Figure 2.9: SEG\text{START} computation segment for normal executions in M_{\text{GROUND-DG}} and in M_{\text{VC-VOTE}} with corresponding states.

In Figure 2.10 the SEG\text{STOP} segment represents a stop sequence (which after the job can be started again via SEG\text{START}). Similarly to SEG\text{START} (see Figure 2.9) jobs in SE state can receive suspend and resume events (Rules R1.6 and R1.7; and Rules R2.12 and R2.13 respectively), but also perform checkpoint events (Rules R1.10 and R2.15 respectively). A stop event (Rule R1.5 and Rule R2.11 respectively) will result in SPE state for both models. In M_{\text{VC-VOTE}} Rule R2.3 starts the abstract job once one of its instances entered SE state. It remains running regardless if the instance is stopped later:
since the instances can stopped/restared by the donors, the abstract job has no knowledge about this so it is not affected. So Rule R2.3 does not affect the abstract job and thus, it is not discussed in detail here.

Figure 2.11 details the $SEG_{EXEC}$ segment. This represents the execution of a job (instance). It starts in $SE$ and leads to $SF$. This is similar to $SEG_{STOP}$ with the only difference of the job termination event (Rules R1.8 and R2.16 respectively).

Figure 2.10: $SEG_{STOP}$ computation segment for normal executions in $M_{GROUND-DG}$ and in $M_{VC-VOTE}$ with corresponding states.

Figure 2.11: Computation segment $SEG_{EXEC}$ for normal executions in $M_{GROUND-DG}$ and in $M_{VC-VOTE}$ with corresponding states.

Figure 2.12 details the job termination ($SEG_{TERM}$) segment that leads from the finished stat ($SF$) to the terminated state ($ST$) where the job does not exist anymore. In $M_{GROUND-DG}$ Rule R1.11 is responsible for this cleanup. In $M_{VC-VOTE}$ this involves three steps. First the result of the job instance is certified via Functionality F2.3 leading
to state $SV_{T_1}$. Next the abstract job is finished successfully (the failure event is detailed in Figure 2.13) leading to state $SV_{T_2}$. Finally the job can be removed via cleanup (Rule R2.7) leading to state $ST$.

Figure 2.12: Computation segment $SEG_{TERM}$ for normal executions in $M_{GROUND-DG}$ and in $M_{VC- VOTE}$ with corresponding states.

Figure 2.13 and 2.14 detail the two cases for the abort sequence. The second case is not needed strictly as for the proof a single run of $M_{VC-VOTE}$ is enough for each run of $M_{GROUND-DG}$, but it is included for completeness. In the first case an abstract job in $M_{VC-VOTE}$ is aborted. Figure 2.13 shows the $SI \rightarrow SF$ path. The $SEG_{INITIALS}$ segment is the same as shown in Figure 2.8. Similarly $SPE, SE$ states could be shown instead here.

Rule R1.9 represents a job abort or failure in $M_{GROUND-DG}$, while Rule R2.14 represents a job instance abort or failure in $M_{VC-VOTE}$. Both lead to $SF$. This is since $SF$ does not differentiate an erroneous or successful job as it is an attribute of the job. However the termination segment ($SEG_{TERM}$) is different for aborted jobs. First it involves a different intermediary state ($SV_{A1}$) in $M_{VC-VOTE}$ by Rule R2.5 compared to the normal termination detailed by Figure 2.12. This rule is responsible handling job failure, however there is no result certification in this case (by Functionality F2.5). The next step involves Rule R2.7 to clean up after the job similarly to $SEG_{TERM}$ presented in Figure 2.12. For $M_{GROUND-DG}$ there is no difference in the $SEG_{TERM}$ segments as there the notion of abstract job is not defined.

In the second case (see Figure 2.14) a different run is shown for the abort sequence in $M_{GROUND-DG}$. Here an abstract job is aborted in $M_{VC-VOTE}$. In $M_{VC-VOTE}$ the abort is handled by Rule R2.6 leading to $SV_{X1}$. The job instance of the abstract job is aborted by Rule R2.14 and finally Rule R2.7 cleans up resulting in $ST$.

As a summary, the model $M_{VC-VOTE}$ is considered to be a correct refinement of $M_{GROUND-DG}$ (based on Definitions 1-7), since for every run of $M_{GROUND-DG}$ model there is a corresponding run of $M_{VC-VOTE}$. Every possible run of the models has been defined by

- showing that the model may only have the following computation segments: $SEG_{INITIALS}, SEG_{INIT}, SEG_{START}, SEG_{STOP}, SEG_{EXEC}, SEG_{TERM}$ and $SEG_{ABORT}$; and
2.3. Modelling volunteer computing

Based on these the following statement can be formulated:

**Statement 3** The refinement of $M_{GROUND-DG}$ to $M_{VC-VOTE}$ is a correct refinement: $M_{GROUND-DG} \triangleright M_{VC-VOTE}$.

The notion of refinement and correct refinement are used as defined by Definitions 1-7.
2.3.4 **M\textsubscript{VC–SPOT}: model for spot-checking based result certification**

Spot-checking relies on sending out jobs with already known results to workers. The model presented here models volunteer computing systems with spot-checking result certification mechanism. It extends the model for desktop grids (M\textsubscript{GROUND–DG}) and also integrates most functionalities from the M\textsubscript{VC–VOTE} volunteer computing model thus, only the differences compared to M\textsubscript{VC–VOTE} are going to be detailed. This is intended to demonstrate that based on the previous models a new model with extended and modified functionality can be developed with relative ease.

2.3.4.1 **Universes and signatures**

The basic elements of the model are the same (see Table 2.8), however there are no instance related functions in the model since certification is solved by introducing oracle jobs (see Assumption 5 in Section 2.3). The following functions are changed and introduced:

\[ initial\text{Instances} : JOB \rightarrow 1 \] \hspace{1cm} (2.66)

\[ role : JOB \times \{abstract, instance\} \rightarrow true \] \hspace{1cm} (2.67)

\[ parent : JOB \times JOB \rightarrow true \] \hspace{1cm} (2.68)

\[ isOracle : JOB \rightarrow \{true, false\} \] \hspace{1cm} (2.69)

\[ checksWithKnownResult : JOB \rightarrow \{true, false\} \] \hspace{1cm} (2.70)

Oracle jobs are jobs with known results and are intended for verifying the workers. The \textit{isOracle} function (see Function 2.69) is set for each such oracle job. There is no parent-child relationship defined thus, the \textit{parent} relation is always true. The \textit{checksWithKnownResult} function compares the result of a job with a previous known valid result and represents the outcome of the comparison. The exact details of when two results are considered corresponding – similarly to \textit{consensus} function of M\textsubscript{VC–VOTE} – is a characteristic of the job.

2.3.4.2 **Initial state**

Initial state rules of M\textsubscript{VC–VOTE} apply for the model.

2.3.4.3 **Functionalities**

All functionalities defined in M\textsubscript{VC–VOTE} (see F2.1 - F2.6) are included in M\textsubscript{VC–SPOT} as well. The main difference is in result certification (Functionality F2.3 in M\textsubscript{VC–VOTE}), which is provided as follows:

**Functionality F3.3: Result certification**

Result (outputs) of the oracle job is compared with the known result. If it matches then all jobs belonging to the host can be considered as correct and possibly the donor rewarded. If the result of the oracle job does not match with the known result all previous results of the host (not yet checked) should be considered invalid. This is show in Listing 2.37.
2.3.4.4 Job phase transition rules

Since there are no separate instance and abstract job roles (denoted by 2.67) there is no instance creation required. Jobs should be regarded as jobs with single instances and without abstract jobs. Oracle jobs have the isOracle relation set to true. This can be done e.g., during submission. However this does not affect the job phase transition rules as they should not be handled differently by the system before certification. Thus Rules R2.8-R2.17 can be reused with minor modifications as Rules 3.2-3.12, while Rule 3.1 is an adaptation of Rule R2.1 for jobs submitted. The phase transition diagram can be adopted from the role(j) = instance part of Figure 2.5.

\[
\exists j \in JOB: \text{if } \text{isOracle}(j) = \text{true} \land \text{jobOutcome}(j) = \text{success} \land \\
\text{consensus}(j) = \text{true} \land \text{readyForConsensus}(j) = \text{true} \text{then} \\
\text{if } \text{checksWithKnownResult}(j) = \text{false} \text{then} \\
\text{consensus}(j) := \text{failure} \\
\text{reward}(\text{donor}(j), j) := \text{false} \\
\text{else} \\
\text{consensus}(j) := \text{success} \\
\text{reward}(\text{donor}(j), j) := \text{true} \\
\text{endif} \\
\text{endif} \\
\forall pj \in JOB: \text{if } \text{host}(pj) = \text{host}(j) \land \text{jobOutcome}(pj) = \text{success} \land \\
\text{isOracle}(pj) = \text{false} \land \text{consensus}(pj) = \{\text{init, none}\} \text{then} \\
\text{consensus}(pj) := \text{consensus}(j) \\
\text{if } \text{consensus}(pj) = \text{success} \text{then} \\
\text{reward}(\text{donor}(j), j) := \text{true} \\
\text{else} \\
\text{reward}(\text{donor}(j), j) := \text{false} \\
\text{endif} \\
\text{endif} \\
\text{endif}
\]

Listing 2.37: Functionality F3.3: Result certification

2.3.4.5 Correspondence of $M_{\text{GROUND-DG}}$ and $M_{\text{VC-SPOT}}$

Similarly to the method used for proving correspondence shown between $M_{\text{GROUND-DG}}$ and $M_{\text{VC-VOTE}}$ in Section 2.3.3.2 the correspondence can be shown between $M_{\text{GROUND-DG}}$ and $M_{\text{VC-SPOT}}$. Using the same segments as defined in Section 2.3.3.2 without the restrictions for job instances the equivalence can be shown rather trivially. Based on these the following statement can be formulated:

**Statement 4** The refinement of $M_{\text{GROUND-DG}}$ to $M_{\text{VC-SPOT}}$ is a correct refinement: $M_{\text{GROUND-DG}} \triangleright M_{\text{VC-SPOT}}$.

The notion of refinement and correct refinement are adopted from Definitions 1-7.
2.3.5 Definition for volunteer computing

Based on Sections 2.3, 2.3.2 and 2.3.3.2 a definition for Volunteer Computing using voting based result certification can be given as follows:

**Statement 5** Rules R2.1-R2.17 together with functionalities F2.1 (resource mapping), F2.2 (user abstraction), F2.3 (result certification), F2.4 (mapped resources become unavailable), F2.5 (delayed jobs) and F2.6 (donor interaction) constitute a reference model \( M_{VC-\text{VOTE}} \) for distributed applications under the assumptions made for volunteer computing systems using voting based result certification in Section 2.3. A volunteer computing system using voting based result certification provides a service equivalent to functionalities F2.1 (resource mapping), F2.2 (user abstraction), F2.3 (result certification), F2.4 (mapped resources become unavailable), F2.5 (delayed jobs) and F2.6 (donor interaction) specified by some reasonable notion of equivalence.

Similarly to Statement 5 a definition for Volunteer Computing using spot-checking based result certification can be given based on the functionalities, job phase transition rules and correspondence described in Section 2.3.4 as follows:

**Statement 6** Rules 3.1-3.12 together with functionalities F3.1 (resource mapping), F3.2 (user abstraction), F3.3 (result certification), F3.4 (mapped resources become unavailable), F3.5 (delayed jobs) and F3.6 (donor interaction) constitute a reference model \( M_{VC-\text{SPOT}} \) for distributed applications under the assumptions made for volunteer computing systems using spot-checking based result certification in Section 2.3. A volunteer computing system using spot-checking based result certification provides a service equivalent to functionalities F3.1 (resource mapping), F3.2 (user abstraction), F3.3 (result certification), F3.4 (mapped resources become unavailable), F3.5 (delayed jobs) and F3.6 (donor interaction) specified by some reasonable notion of equivalence.

2.3.5.1 Discussion of existing middleware

As a continuation of Section 2.2.7.1 BOINC, HTCondor and XtremWeb are discussed further, with regards to their fitness for Volunteer Computing.

BOINC and its derivatives like SZTAKI Desktop Grid – as discussed – provide service equivalents for all functionalities defined in Statement 2 and thus, they are Desktop Grids based on that definition. BOINC uses voting based result certification that satisfies Functionality F2.3: job instances are called "results" – not to be confused with outputs of finished tasks – and user provided components (called "validators") are used to determine if (i) two or more outputs are matching; (ii) select a "canonical result" that is
used later to compare outputs of additional instances. BOINC provides some predefined validators for e.g., comparing outputs bit-wise, but the real power is in user defined comparison functions. As there is no difference in user abstraction (see Functionality F2.2) between $M_{GROUND-DG}$ and $M_{VC-VOTE}$ BOINC satisfies it. Similarly for mapped resources become unavailable functionality (see Functionality F2.4). For resource mapping (see Functionality F2.1) BOINC uses the notion of applications and workunits (jobs). The former are pre-deployed binaries, while latter are inputs to those. BOINC also uses "platforms" that is a combination of operating system and architecture. Each application is tied to one or more platforms (abstract resource requirement for applications) while jobs have memory, disk and CPU requirements. BOINC uses the notion of "deadline" that is used to identify belated or lost jobs (see Functionality F2.5). In BOINC the donor has full control over her host, she might suspend BOINC itself and/or running jobs; or abort any job her host is running at any time (see Functionality F2.6). As a conclusion BOINC is considered a volunteer computing system based on Statement 6 and it is suited for deployments in environments characterized in Section 2.1 for Volunteer Computing.

HTCondor – as discussed in Section 2.2.7.1 – provides service equivalents for all functionalities defined in Statement 2 and thus, it is a Desktop Grid based on that definition. Functionalities user abstraction (see Functionality F2.2), resource mapping (see Functionality F2.1) and mapped resources become unavailable (see Functionality F2.4) functionalities used as-is or in an extended manner in $M_{VC-VOTE}$ and the corresponding ones in $M_{VC-SPOT}$ thus HTCondor satisfies these. However HTCondor provides no result certification (see Functionality F2.3 and F3.3) and donor interaction functionality (see Functionality F2.6 and F3.6). As a conclusion HTCondor cannot be considered a volunteer computing system based on Statement 6 and it is not suited for deployments in environments characterized in Section 2.1 for Volunteer Computing.

XtremWeb – as discussed in Section 2.2.7.1 – provides service equivalents for all functionalities defined in Statement 2 and thus, it is a Desktop Grid based on that definition. However XW-HEP does not provide Functionality F2.4 thus, it is not a Desktop Grid based on Statement 2. Functionalities user abstraction (see Functionality F2.2), resource mapping (see Functionality F2.1) and mapped resources become unavailable (see Functionality F2.4) functionalities used as-is or in an extended manner in $M_{VC-VOTE}$ and the corresponding ones in $M_{VC-SPOT}$. As a result XW-HEP cannot be considered a volunteer computing system based on Statement 6 and it is not suited for deployments in environments characterized in Section 2.1 for Volunteer Computing. Additionally both XtremWeb and XW-HEP lack any result certification functionalities (see Functionality F2.3 and F3.3): “Result certification: The XWHEP middleware does not propose anything on this field. It is the end user responsibility to verify the results of his/her jobs.”

---

2.4 A formal model of BOINC

The Berkeley Open Infrastructure for Network Computing (BOINC) [57] is a widely used volunteer computing framework with more than 80 public deployments around the world [68]. A deployment of BOINC is referred as a project. In this section I present a formal model of BOINC, namely $M_{BOINC}$, based on the previous volunteer computing model $M_{VC-VOTE}$.

For this theorem I created a criteria system that incorporates the most important aspects of BOINC. I demonstrate that the model $M_{BOINC}$ I developed and is a formal model for BOINC corresponds to these aspects. I show that $M_{BOINC}$ is a correct refinement of $M_{VC-VOTE}$ thus, it represents a volunteer computing system, a desktop grid system and a grid system. I demonstrate that $M_{BOINC}$ satisfies the requirements of Theorem I.2.:

\begin{center}
\textbf{Theorem I.3.:} Model $M_{BOINC}$ is a correct refinement of model $M_{VC-VOTE}$, further $M_{BOINC}$ correctly represents the internal architecture and operation of BOINC thus, BOINC qualifies as a volunteer computing system, a desktop grid system and a grid system.
\end{center}

Related publications: [16], [13], [14], [15] and [12].

2.4.1 Assumptions

The model takes into account the following assumptions for BOINC based on the previous assumptions made for $M_{VC-VOTE}$, and the discussion in Section 2.2.7.1 and 2.3.5.1:

1. \textit{BOINC follows a centralized client-server architecture.} 
   A deployment of BOINC is referred as a project. Volunteer hosts have the worker deployed that is in regular contact with the project. A comparison of the different node types and their connection is shown in Figure 2.15. Although BOINC is centralized, several of the low level components of the BOINC server may be replicated to or deployed at different nodes for load balancing and failure tolerance. These components are described in Table 2.9. These components affect the life-cycle of jobs. Regardless of the distributed nature of BOINC, the life-cycle of jobs can be modeled with a finite-state machine. BOINC provides a preliminary API for remote job submission, however it is also using the UI of the project, so the $M_{BOINC}$ model assumes that the job submission is at the UI and all components of the project are deployed on the same node.

2. \textit{Applications cannot be submitted as part of jobs. Jobs must refer a previously at the project deployed application.} 
   An administrator must register applications by hand at the project application
2.4. A formal model of BOINC

<table>
<thead>
<tr>
<th>Server Side</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduler</td>
<td>Communicates with clients including dispatching jobs and updating status of jobs.</td>
</tr>
<tr>
<td>Feeder</td>
<td>Creates a shared-memory segment used to pass database records to the Scheduler. The data includes applications, their versions, and “work items” (an untended result and its corresponding workunit).</td>
</tr>
<tr>
<td>Transitioner</td>
<td>Handles state transitions of workunits and results. Generates initial results for workunits, and generates more results when timeouts or errors occur.</td>
</tr>
<tr>
<td>Validator*</td>
<td>Compares returned results of a workunit and selects correct ones. Involves custom application specific code.</td>
</tr>
<tr>
<td>Assimilator*</td>
<td>Processes outputs of successfully finished jobs, e.g., parse the output files and store the data in a database. Involves application specific code.</td>
</tr>
<tr>
<td>File deleter</td>
<td>Deletes files after the assimilation process has finished.</td>
</tr>
</tbody>
</table>

Table 2.9: Components of BOINC on the server side. Components denoted with “*” are application dependent.

repository. First the application name must be added (i.e., its name must be entered into the database of the project). Next concrete implementations can be deployed. This involves signing the binaries of the application with a project specific key (BOINC uses public key authentication), copying files to specific locations, adding a new database entry and finally notifying the project components of the new application version. The presence of a repository in BOINC stems from multiple causes. First BOINC targets parameter sweep or parameter study type of applications. There the application does not change, rather a large parameter space is divided into smaller pieces and formulated as input files. This allows for example “caching” the application on workers, with each job only the input files and parameters (i.e., the unit of work) needs to be downloaded to the hosts. Second reason is the question of trust. Donors provide their hosts voluntarily for a BOINC project. The project promotes itself with the grand scientific problem it aims to solve. This problem is represented by the application it runs. The deployment of the application at the repository – that involves the signing of all its binaries using a public key mechanism – aims to guarantee that only those application(s) will be sent by the project that it wants to support and no user can submit (malicious) other ones.

3. The resource requirements of applications are consolidated into resource groups.
In MVC-VOTE the resource requirements of jobs were split into the requirements of the application and the unit of work. In BOINC the resource requirements of applications are restricted mainly to operating system and system architecture, while the resource requirements of unit of works contain other ones (e.g., disk and memory requirements). The combination of these operating system and architecture (e.g., Linux and AMD64, or Windows and x86) of an application is referred as platform\(^6\). Platforms are predefined in BOINC (e.g., i686-pc-linux-gnu), however different platforms can be added manually. Applications can support any platform. There is no such requirement that they must support all platforms.

4. BOINC implements a result certification mechanism based on comparing returned

\(^6\)http://boinc.berkeley.edu/trac/wiki/BoincPlatforms, accessed on 2014-01-01
finished job instances.
This can be regarded as a refinement of the result certification concept from \textit{M\textsubscript{VC-VOTE}}. The validators must be supplied on a per application basis. It is an application specific task to determine whether to job instances can be considered as matching or not.

5. \textit{BOINC may create additional job instances during the running phase of an abstract job.}
The requirement introduced by \textit{M\textsubscript{VC-VOTE}} only states that there must be a job replication method available that is able to create a desired number of instances for each job submitted. Every time a job instance fails or is considered lost another one is created by BOINC to supplement it until a certain threshold is reached.

6. \textit{Donors can restrict access to their hosts’ on a per application base.}
Each donor is able to filter the applications of the different BOINC project she contributes to. Only jobs that use these applications are allowed to run on her donated host(s).

7. \textit{Donors are granted virtual credits for their contributed resources.}
As reward and incentive donors are awarded virtual credits for each job instance they successfully complete based on the amount of contributed processor time. BOINC estimates the peak FLOPS (floating operations per second) for each processor the host has. For CPUs BOINC runs the Whetstone benchmark, for GPUs the manufacturer provided values are used. The basis of the system is the “Cobblestone”, from which 200 would be claimed by a host for one day of work with the performance of 1000 double precision MIPS (million instructions per second) based on the Whetstone benchmark. Each host completing a job instance reports the amount of credit it requests (claimed credit) and the server decides what amount will be given, e.g., the highest and lowest values are dropped and the mean is given (granted credit). The server also uses different normalization approaches that tries to normalize across hosts, different devices, application versions, etc \footnote{http://boinc.berkeley.edu/trac/wiki/CreditNew, accessed on 2014-01-01}, but also new approaches for a generalized credit mechanism is being investigated\footnote{http://boinc.berkeley.edu/trac/wiki/CreditGeneralized, accessed on 2014-01-01}.

2.4.2 Universes and signatures
Table 2.10 and Figure 2.16 summarizes the introduced universes for the new model. The \textit{PLATFORM} universe is introduced to represent the different preset combination of operating system and system architecture requirements of applications in BOINC. The \textit{REPOSITORY} universe represents all application repositories where applications are deployed. Each BOINC project has a single repository. This is since BOINC uses private/public key based signing of applications binaries. The signing is always performed by the project using its own private key, thus it is not possible to use applications from another projects repository.
2.4. A formal model of BOINC

Figure 2.15: On the right side the different nodes (host, manager, ui) used in M_{VC-VOTE}. On the left side their counterparts in M_{BOINC}.

<table>
<thead>
<tr>
<th>M_{VC-VOTE}</th>
<th>M_{BOINC}</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER</td>
<td>USER</td>
</tr>
<tr>
<td>JOB</td>
<td>JOB</td>
</tr>
<tr>
<td>APPLICATION</td>
<td>APPLICATION</td>
</tr>
<tr>
<td>-</td>
<td>PLATFORM</td>
</tr>
<tr>
<td>UNIT OF WORK</td>
<td>UNIT OF WORK</td>
</tr>
<tr>
<td>ARESOURCE</td>
<td>ARESOURCE</td>
</tr>
<tr>
<td>NODE</td>
<td>NODE</td>
</tr>
<tr>
<td>-</td>
<td>REPOSITORY</td>
</tr>
<tr>
<td>UI</td>
<td>UI</td>
</tr>
<tr>
<td>MANAGER</td>
<td>MANAGER</td>
</tr>
<tr>
<td>DONOR</td>
<td>DONOR</td>
</tr>
<tr>
<td>HOST</td>
<td>HOST</td>
</tr>
<tr>
<td>TASK</td>
<td>TASK</td>
</tr>
<tr>
<td>PRESOURCE</td>
<td>PRESOURCE</td>
</tr>
</tbody>
</table>

Table 2.10: Basic elements of the ASM model: on the left side the M_{VC-VOTE} model for volunteer computing, and M_{BOINC} model of BOINC on the right side.

**Functions modified and introduced** This paragraph is intended to collect all modified and newly introduced functions. The more generic ones are detailed here, and the more specialised are discussed at their places of usage. The following functions are carried over in a modified fashion compared to their corresponding ones in M_{VC-VOTE} model:

\[
\text{managerOf} : \{\text{NODE, REPOSITORY}\} \rightarrow \text{MANAGER} \tag{2.71}
\]
\[
\text{role} : \text{JOB} \rightarrow \{\text{workunit, result}\} \tag{2.72}
\]
\[
\text{parent} : \text{JOB} \rightarrow \text{JOB} \tag{2.73}
\]

The \text{managerOf} function (see Equation 2.71) is modified to incorporate the newly introduced repositories. Abstract jobs are referred as \textit{workunits} and job instances as \textit{results} in BOINC thus, the \text{role} function (see Equation 2.72) is changed. This model will refer to jobs with \textit{workunit} roles as workunits and similarly to jobs with \textit{result} roles as results. The \text{parent} function (see Equation 2.73) has only a stylistic change.

The following platform and repository related functions are newly introduced by the
model:

\[
\text{repositoryOf : MANAGER} \rightarrow \text{REPOSITORY}\quad (2.74)
\]

\[
\text{isDeployed : REPOSITORY} \times \text{APPLICATION} \\
\rightarrow \{\text{true, false}\}\quad (2.75)
\]

\[
\text{supportsPlatform : \{HOST, APPLICATION\} \times PLATFORM} \\
\rightarrow \{\text{true, false}\}\quad (2.76)
\]

\[
\text{platform : \{APPLICATION, HOST\}} \rightarrow 2^{\text{PLATFORM}}\quad (2.77)
\]

\[
\text{appAllowed : DONOR} \times \text{APPLICATION} \\
\rightarrow \{\text{true, false}\}\quad (2.78)
\]

\[
\text{mappedPlatform : JOB} \rightarrow \text{PLATFORM}\quad (2.79)
\]

\[
\text{count : } 2^{\text{ITEM}} \rightarrow \text{NUMBER}\quad (2.80)
\]

The repositoryOf function (see Equation 2.76) determines the repository connected to the given manager and the isDeployed relation (see Equation 2.75) tells if an application is deployed in a repository. The supportsPlatform function (see Equation 2.76) denotes if a host or application supports a given platform. Similarly the platform function (see Equation 2.77) returns all platforms supported by given application or host. The appAllowed function (see Equation 2.78) determines if a donor allows a given application. The mappedPlatform function (see Equation 2.79) denotes the actual platform mapped to the job. These functions are more detailed at Functionality F4.1. Finally the count macro (see Equation 2.80) determines the count of items in the set used as parameter (or 1 if a single item is provided), where ITEM represents a set containing elements from all universes in the model.

The following functions are introduced for solely role\((j) = \text{workunit}\) jobs with the exception of fileDeleteState:

\[
\text{fileDeleteState : JOB} \rightarrow \{\text{init, ready, done}\}\quad (2.81)
\]

\[
\text{canonicalResult : JOB} \rightarrow \text{JOB}\quad (2.82)
\]

\[
\text{needValidate : JOB} \rightarrow \{\text{false, true}\}\quad (2.83)
\]

\[
\text{assimilateState : JOB} \rightarrow \{\text{init, ready, done}\}\quad (2.84)
\]

\[
\text{ERRORMASK : \{could\_not\_send, too\_many, too\_many\_success, too\_many\_total\}}\quad (2.85)
\]

\[
\text{errorMask : JOB} \rightarrow 2^{\text{ERRORMASK}}\quad (2.86)
\]

For workunits (role\((j) = \text{workunit}\)) fileDeleteState represents the state of the input files, while for results the state of their output files. The validator component of BOINC selects one from the valid and successful results that acts as a reference for later completed results to compare against. This result is represented by canonicalResult (see Equation 2.82). The needValidate (see Equation 2.83) function indicates that the workunit has a result
that needs validation. Refer to Functionality F4.3 for more details about validation. The output of a successful workunit needs to be retrieved (sometimes also further processed) before the workunit can be discarded. This retrieval and post-processing step is referred as assimilation. The \( \text{assimilateState} \) function (see Equation 2.84) denotes this. Initially workunits have init assimilateState \( (\text{assimilateState}(j) = \text{init}) \). Once they are finished they are ready to be assimilated \( (\text{assimilateState}(j) = \text{ready}) \), and once their output is retrieved and they can be deleted their assimilateState is set to \textit{done}.

The following functions are introduced for \( \text{role}(j) = \text{result} \) type of jobs:

\[
\text{serverState} : \text{JOB} \rightarrow \{\text{unsent}, \text{in\_progress}, \text{over}\} \tag{2.87}
\]

\[
\text{outcome} : \text{JOB} \rightarrow \{\text{success}, \text{couldnt\_send}, \text{client\_error}, \\
\text{no\_reply}, \text{didnt\_need}, \text{validate\_error}, \\
\text{client\_detached}\} \tag{2.88}
\]

\[
\text{clientState} : \text{JOB} \rightarrow \{\text{downloading}, \text{downloaded}, \\
\text{compute\_error}, \text{uploading}, \text{uploaded}, \text{aborted}\} \tag{2.89}
\]

\[
\text{validateState} : \text{JOB} \rightarrow \{\text{init}, \text{valid}, \text{invalid}, \text{no\_check}, \text{error}, \\
\text{inconclusive}, \text{too\_late}\} \tag{2.90}
\]

\[
\text{successResults}, \text{runningResults}, \\
\text{unsentResults}, \text{failedResults} : \text{JOB} \rightarrow 2^{\text{JOB}} \tag{2.91}
\]

\[
\text{minQuorum}, \text{targetNResults}, \text{maxErrorResults}, \\
\text{maxTotalResults}, \text{maxSuccessResults}, \text{delayBound}, \\
\text{sentTime}, \text{serverTime} : \text{JOB} \rightarrow \text{NUMBER} \tag{2.92}
\]

\( \text{serverState} \) (see Equation 2.87) denotes the status of the workunit from the perspective of the server: it can be \textit{unsent}; \textit{in\_progress}, meaning a client started processing at least one of its results; or \textit{over} if the workunit is considered finished.

The \textit{successResults} macro returns the set of successful finished results for the workunit. Similarly the \textit{runningResults}, \textit{unsentResults} and \textit{failedResults} macros (see Equation 2.91) return the set of running, not yet started and failed results. \textit{minQuorum} denotes the number of successful results for a workunit to start validation. \textit{targetNResults} (see Equation 2.91 and Rule R4.1) denotes the number of initial results thus, replaces the \textit{initialInstances} function of \textit{M\_VOTE}. \textit{maxTotalResults} (see Equation 2.92) denotes the maximum number of results (with any state or outcome) before the workunit is considered failed. Similarly \textit{maxErrorResults}, \textit{maxSuccessResults} denote the maximum number of erroneous and successful results. \textit{delayBound}, \textit{sentTime} and \textit{serverTime} are discussed with Functionality F4.5. Finally the result certification functionality is provided by the different validators thus \textit{consensus} function of \textit{M\_VOTE} (see Equation 2.52) is replaced by the function \textit{validate}. See Functionality F4.3 for more details.

The matching workunit (\textit{fileDeleteState}, \textit{assimilateState}, etc.) and result states (\textit{serverState}, \textit{outcome}, etc.) are described on Tables 2.11 and 2.12 with references to the job phase transition diagram (see Figure 2.17).
Chapter 2. Formal models for desktop grids and volunteer computing

Figure 2.16: Basic elements of the ASM model $M_{BOINC}$.

<table>
<thead>
<tr>
<th>fileDelete</th>
<th>assimilate</th>
<th>need</th>
<th>validate</th>
<th>error</th>
<th>jobPhase</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P^{A}_{wu}$</td>
<td>undef</td>
<td>undef</td>
<td>undef</td>
<td>undef</td>
<td>undef</td>
<td>job submitted</td>
</tr>
<tr>
<td>$P^{A#1}_{wu}$</td>
<td>init</td>
<td>init</td>
<td>false</td>
<td>0</td>
<td>(init)</td>
<td>results created</td>
</tr>
<tr>
<td>$P^{A#2}_{wu}$</td>
<td>init</td>
<td>init</td>
<td>false</td>
<td>0</td>
<td>improgress (readytorun)</td>
<td></td>
</tr>
<tr>
<td>$P^{A#3}_{wu}$</td>
<td>init</td>
<td>init</td>
<td>false</td>
<td>0</td>
<td>running</td>
<td>one result is running</td>
</tr>
<tr>
<td>$P^{W}_{wu}$</td>
<td>init</td>
<td>ready</td>
<td>false</td>
<td>0</td>
<td>/ -0</td>
<td>results validated</td>
</tr>
<tr>
<td>$P^{W#1}_{wu}$</td>
<td>init</td>
<td>done</td>
<td>false</td>
<td>0</td>
<td>/ -0</td>
<td>awaited for assimilation</td>
</tr>
<tr>
<td>$P^{W#2}_{wu}$</td>
<td>ready</td>
<td>done</td>
<td>false</td>
<td>0</td>
<td>/ -0</td>
<td>assimilated</td>
</tr>
<tr>
<td>$P^{W#3}_{wu}$</td>
<td>done</td>
<td>done</td>
<td>false</td>
<td>0</td>
<td>/ -0</td>
<td>assimilated</td>
</tr>
<tr>
<td>$P^{W#4}_{wu}$</td>
<td>done</td>
<td>done</td>
<td>false</td>
<td>0</td>
<td>/ -0</td>
<td>files deleted</td>
</tr>
<tr>
<td>$P^{Z}_{wu}$</td>
<td>discarded</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.11: Values of different state variables of a workunit for different job phases.

2.4.3 Initial state

In the following we define the initial state of the model $M_{BOINC}$:
### 2.4. A formal model of BOINC

#### Table 2.12: Values of different state variables of a result for different job phases.

<table>
<thead>
<tr>
<th>P</th>
<th>report</th>
<th>serverState</th>
<th>outcome</th>
<th>clientState</th>
<th>fileDelete</th>
<th>validateState</th>
<th>jobPhase</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{ab}$</td>
<td>undef</td>
<td>undef</td>
<td>undef</td>
<td>undef</td>
<td>undef</td>
<td>undef</td>
<td>job submitted</td>
</tr>
<tr>
<td>$P_{bc}$</td>
<td>undef</td>
<td>unsent</td>
<td>undef</td>
<td>undef</td>
<td>init</td>
<td>init</td>
<td>init</td>
</tr>
<tr>
<td>$P_{cd}$</td>
<td>~undef</td>
<td>in_progress</td>
<td>undef</td>
<td>undef</td>
<td>init</td>
<td>init</td>
<td>readytorun</td>
</tr>
<tr>
<td>$P_{de}$</td>
<td>~undef</td>
<td>in_progress</td>
<td>undef</td>
<td>undef</td>
<td>init</td>
<td>init</td>
<td>running</td>
</tr>
<tr>
<td>$P_{ef}$</td>
<td>~undef</td>
<td>over</td>
<td>~undef</td>
<td>(client_error)</td>
<td>init</td>
<td>init</td>
<td>finished</td>
</tr>
</tbody>
</table>

1. The following initial state rules of $M_{\text{GROUND-DG}}$ and $M_{\text{VC-VOTE}}$:

   $M_{\text{GROUND-DG}} : 2.32 - 2.43$

   $M_{\text{VC-VOTE}} : 2.60$  (2.93)

2. All applications must be deployed at a repository:

   \[
   \forall app \in \text{APPLICATION}, \exists repo \in \text{REPOSITORY}:
   \]

   isDeployed(repo, app) = true  (2.94)

3. All jobs must have an application and a unit of work:

   \[
   \forall j \in \text{JOB}, \exists app \in \text{APPLICATION}, \exists uow \in \text{UNITOFWORK}:
   \]

   application(j) = app ∧ unitofwork(j) = uow  (2.95)

4. There is only a single repository, ui and manager in a BOINC project located on the same node:

   \[
   \forall mgr \in \text{MANAGER} : \exists rep_1, \neg \exists rep_2 \in \text{REPOSITORY}:
   \]

   rep_1 ≠ rep_2 ∧ repositoryOf(mgr) = rep_1∧

   repositoryOf(mgr) = rep_2  (2.96)

   \[
   \forall rep \in \text{REPOSITORY} : \exists mgr_1, \neg \exists mgr_2 \in \text{MANAGER}:
   \]

   mgr_1 ≠ mgr_2 ∧ managerOf(rep) = mgr_1∧

   managerOf(rep) = mgr_2  (2.97)

   \[
   \forall ui \in \text{UI} : \exists mgr_1, \neg \exists mgr_2 \in \text{MANAGER}:
   \]

   mgr_1 ≠ mgr_2 ∧ managerOf(ui) = mgr_1∧

   managerOf(ui) = mgr_2  (2.98)
∀mgr ∈ MANAGER, ∃ui ∈ UI, rep ∈ REPOSITORY :
    managerOf(repositoryOf(ui)) = mgr ∧
    nodeOf(mgr) = nodeOf(ui) ∧
    nodeOf(rep) = nodeOf(mgr)

(2.99)

It is assumed that this topology does not change during the life-cycle of any project.

5. Each application must have a validator that provides the checkset and checkpair functions specific for the application.

2.4.4 Functionalities

Functionality \textit{F4.1}: Resource mapping

\begin{verbatim}
if (∃ar ∈ ARESOURCE, ∃h ∈ HOST, ∃job ∈ JOB, ∃uow ∈ UNITOFWORK, ∃app ∈ APPLICATION, ∃plat ∈ PLATFORM, ∃d ∈ DONOR):
    role(job) = result ∧ application(job) = app ∧
    unitOfWork(job) = uow ∧ mappedResource(uow, ar) = undef ∧
    resourceRequest(uow, ar) = true ∧ event(job) ≠ abort then
    choose pr in PRESOURCE satisfying
        compatible(attr(ar), attr(pr)) = true ∧
        belongsTo(pr, h) = true ∧ available(h) = true ∧
        acceptsJobs(h) = true ∧ supportsPlatform(app, plat) = true ∧
        donor(h) = d ∧ appAllowed(d, app) = true ∧
        supportsPlatform(h, plat) = true
    mappedResource(uow, ar) := pr
    mappedPlatform(job) := plat
endif
\end{verbatim}

Listing 2.38: Resource mapping functionality

The functionality is shown in Listing 2.38. Resource requests of applications are consolidated into platforms in BOINC. During resource mapping it must be ensured that the host supports one of the platforms the application has available. Applications can have multiple implementations, each for a different platform, thus the selected platform is rather mapped to the job instead of the application. Still \textit{unitofworks} have resource requests and mapped resources unchanged. The \textit{supportsPlatform} : \textit{APPLICATION} × \textit{PLATFORM} → \{true, false\} function tells if an application has an instance for the given platform. This must platform must match the platform the host is reporting. The mapped platform of the job is represented by the \textit{mappedPlatform} : \textit{JOB} → \textit{PLATFORM} function. The \textit{platform} : \{HOST, JOB\} → \textit{PLATFORM} returns a platform the entity supports (there is no restriction that only a single one can be supported). The donor has the privilege to select allowed applications for her hosts as she wishes. BOINC projects usually run a single application aimed at solving some (grand) scientific challenge. However there are \textit{umbrella} projects that host different applications. In this case the donor is given the freedom to disable application that she does not wish to support by accepting their jobs. This is represented by the \textit{appAllowed} : \textit{DONOR} × \textit{APPLICATION} → \{true, false\} relation.
2.4. A formal model of BOINC

Functionality F4.2: User abstraction
In BOINC the globaluser → localuser mapping is straightforward since all users of BOINC have access to all hosts. One restriction that can be set for accessing the hosts is imposed by the allowed applications list of donors discussed in Functionality F4.1. However this does not affect user abstraction. So the functionality is equivalent to F1.2 of MVC-VOTE:

\[ F4.2 \equiv F2.2 \]

Functionality F4.3: Result certification
Comparing returned results is an application specific task in BOINC since, for every application it is different what can be considered matching (e.g., small difference resulting from floating point implementation differences by different operating systems, or different end of line markers, etc.). In BOINC validators implement this functionality. Therefore a validator is required for every application. A generic validator framework is provided by BOINC and every validator must supply two functions for the framework: checkSet and checkPair. The abstract representation of the framework and the provided functions are shown on listing 2.40.

```plaintext
local rule checkSet(success_results) :=
  ...

local rule checkPair(result, canonical_result) :=
  ...

resultCertification(app1)
```

Listing 2.39: Validator module for ‘app1’ application: ΠValidator_app1

Each of the validators provides an implementation for these functions that check the successful results of a single application. BOINC also provides validators for some common cases that can be reused by applications, e.g., a bitwise validator that compares results bit-by-bit. In this model these are represented as modules and are executed by external agents. These validators in the model use the following naming convention: ΠValidator_<application> where <application> represents the application (usually by a name it can be referred) the validator belongs to. An example for the application ‘app1’ is shown on Listing 2.39. Ultimately the rules determine if the successful finished results indeed produced correct outputs. This is represented by the validate function of the results: valid, invalid or inconclusive if no decision could be made. Validation is achieved in two ways depending on if a representative result ("canonical" result denoted by the canonicalresult : JOB → JOB function) is already found for the workunit.

If a canonical result was already found (see canonicalResult(j) ≠ undef on Listing 2.39) then all new results are compared against it (see checkPair). The outcome of this comparison can be that either the results match thus the new one is valid (validate(r) = valid) or they mismatch and the result is invalid. If there is no canonical result available yet then first it is checked whether there are enough successful results available to form a quorum and select one (the number is determined by the minQuorum func-
tion). Next a check is run on the set of results with checkSet. This function compares all results, decides whether they are valid or invalid and selects a canonical result from the valid ones. It is still a possibility that no canonical result is found (validate(r) = inconclusive for all results). In this case the validation procedure is rerun later when a new successful result is returned. However if the limit for successful results is reached and still there is no consensus on the validation the workunit is considered failed. If there is no consensus but the limit is not reached then the targetNResults is increased for the workunit and a new result (job instance) will be created (by rule R4.2). If a canonical result is found by checkset then there is no need to send the unsent results to clients, thus abort event is generated for them. However already in progress results should be accepted (and validated) when they are returned.

```plaintext
rule resultCertification (app) :=
  if ( exists job ∈ JOB : needValidate (job) = true ∧ role (job) = workunit ∧ application (job) = app ) then
    if ( canonicalResult (job) ≠ undefined ) then
      forall ( r ∈ JOB : parent (r) = job ∧ jobOutcome (r) = success ∧
        validate (r) ∈ {init, inconclusive} ) do
        validate (r) = checkPair (r, canonicalResult (job))
        if ( validate (r) = valid ) then
          grantCredit (r)
        endif
      endforall
    else
      if ( count (successResults (job)) ≥ minQuorum (job) ) then
        canonicalResult (job) := checkSet (successResults (job))
      endif
      forall ( r ∈ successResults (job) : (validate (r) = valid) ) do
        grantCredit (r)
      endforall
      if ( canonicalResult (job) ≠ undefined ) then
        forall ( k ∈ unsentResults (job) ) do
          event (k) := abort
        endforall
      else
        if ( count (successResults (job)) ≥ maxSuccessResults (job) ) then
          event (job) := abort
        endif
        if ( count (successResults (job)) ≥ targetNResults (job) ) then
          targetNResults (job) := targetNResults (job) + 1
        endif
      endif
    endif
  endif
endif
```

Listing 2.40: The result certification functionality

For valid results credit is granted by the validator. This is represented by the grantCredit function and will be detailed later. All donors who produced a valid result for the workunit are granted the same amount of credit. The amount is either fixed preset for the workunit or based on the amount of work BOINC client reports (although normalized in various ways), referred as claimed credit. There is a maximum set for how much credit a client can claim for the result and for example the mean (or median) of these is granted.
2.4. A formal model of BOINC

**Functionality F4.4: Mapped resources become unavailable**
The model does not need any refinements in this aspect compared to $M_{VC-VOTE}$ thus, the functionality becomes:

\[
F_{4.4} \equiv F_{2.4}
\]

**Functionality F4.5: Job is delayed**
Only job instances can be delayed, the functionality must be modified to incorporate this:

```plaintext
if (\exists j \in JOB) : role(j) = result \land jobDelayed(j) = true then
    maskEvents(j, \{start, resume, stop, suspend, ckpt\}, host) := true
    event(j) := abort
endif
```

Listing 2.41: Example for delayed job handling

BOINC assigns a delay bound to workunits (represented by the $delayBound : JOB \rightarrow NUMBER$ function). This is set in seconds and represents how much time a result of the workunit has before it is considered delayed and discarded. The rule representing this characteristic is shown on listing 2.42. The timer is started when the task is installed on the volunteer host (see rule R4.12). The sent time is represented by the $sentTime : JOB \rightarrow NUMBER$ function, while the $serverTime : JOB \rightarrow TIME$ function represents the time of the server where the project is from which the job is originating.

```plaintext
rule jobDelayed(j) =
    return := false
    if (role(j) = result \land serverTime(j) > sentTime(j) + delayBound(parent(j)) ) then
        return := true
    endif
```

Listing 2.42: $jobDelayed$ function of Functionality F2.5

**Functionality F4.6: Donor interaction**
The model does not contain any refinements compared to Functionality F2.6 of $M_{VC-VOTE}$ only the changes to the role functions must be incorporated: donor events target results. I am going to omit the listing.

2.4.5 Job phase transition rules

The following rules describe how job phases represented by the function $jobPhase$ are evolving in the model. An overview is shown in Figure 2.17. Rules R4.1-R4.6 detail the life-cycle of work units, while rules R4.11-R4.24 detail results.

**Rule R4.1: Job submitted**
The rule is shown in Listing 2.44 and the complementary submitted rule is detailed in Listing 2.43.
Figure 2.17: Job phase transition diagram for the M_{BOINC} model. Dashed arrows and job phases with dashed frames represent the three states and their transitions contracted into inprogress job phase via (m,0)-trapezoid refinement (see Section 2.3.3.1 and Figure 2.7 for more details).

Listing 2.43: the submitted : JOB × USER × UI × MANAGER → {true, false} function, part of Rule R4.1.

In case of a submitted job does not satisfy the guards of the rule, the job is considered as an erroneous one that the model does not aim to handle since in this case the job is not able to enter BOINC. Once the job enters BOINC it is assigned the running status and the role of workunit. In BOINC there is no distinguished phase of init, readytorun and running for workunits. Rather after submit the job enters a phase that is a consolidation
of these three. It is referred in the model as inprogress.

The default values for the job replication are set here for the missing values and the job is checked if it can enter BOINC. The minQuorum : JOB → NUMBER function defines the required number of successfully finished results to start the validation process. The targetNResults : JOB → NUMBER function determines how many initial instances (results) should be created, it is basically the initialInstances function of M\textsubscript{VC→VOTE}. The maxErrorResults : JOB → NUMBER function sets how many failed (erroneous) results a workunit may have before it is considered failed as well. The maxSuccessResults : JOB → NUMBER determines the maximum number of successful results. If this number is reached and the validator still cannot validate the workunit it is considered failed. Finally the delayBound determines the maximum turnaround time for a result. This means the maximum time the donors have to return the finished result after it is installed on their host. The job can only enter BOINC if it refers to a valid application and was submitted using a UI connected to the manager of BOINC.

Listing 2.44: Rule R4.1: job submitted

Rule R4.2: Create results
BOINC can create results for the submitted job as needed. It will create targetNResults number of results at the beginnig. However later it can create additional ones, e.g., if a result is late or failed. The conditions is covered by the needResults : JOB → \{true, false\} function that is detailed on Listing 2.46. The guard in the needResults function (see Listing 2.46) is broken into three statements for better readability. Additional instances (results) are needed if the maximum number is not reached (set by
maxSuccessResults, maxErrorResults and maxTotalResults) and the number of target results (targetNResults) is greater than the number of unsent, running and successful results. The number of target results that is used as the number of initial results required is actually increased when needed by the validator, that’s why it is included in the condition.

if canonicalResult(j) = undef ∧ errorMask(j) = 0 ∧ role(j) = workunit ∧ needResults(j) = true ∧ jobPhase(j) = inprogress then
extend JOB by child with
parent(child) := j
role(child) := result
application(child) := application(j)
unitofwork(child) := unitofwork(j)
jobPhase(child) := init
serverState(child) := unsent
jobOutcome(child) := undef
endextend
endif
Listing 2.45: Rule R4.2: Create results

rule needResults(job) :=
return := false
if (role(job) = workunit) then
if (count(successResults(job)) < maxSuccessResults(job) ∧
count(failedResults(job)) < maxErrorResults(job) ∧
count(totalResults(job)) < maxTotalResults(job)) then
if (targetNResults(job) − count(unsentResults(job)) −
count(runningResults(job)) − count(successResults(job))) > 0) then
return := true
endif
endif
endif
Listing 2.46: The needResults : JOB → {true, false} function, part of Rule R4.2.

Rule R4.3: Workunit successful
A workunit is considered successful if there is a canonical result (and the error mask is empty). It is represented by the canonicalResult function, which is set by the validator.

if (canonicalResult(j) ≠ undef ∧ errorMask(j) = 0 ∧ role(j) = workunit ∧
event(j) = {}) then
jobPhase(j) := finished
jobOutcome(j) := success
endif
Listing 2.47: Rule R4.3: Workunit successful

Rule R4.4: Workunit failed
A workunit is considered failed if its error mask is not empty. It is set by the validator or by the transitioner.

if (jobPhase(j) = running ∧ role(j) = workunit ∧ event(j) ≠ abort ∧
errorMask(j) ≠ 0) then
jobPhase(j) := finished
jobOutcome(j) := failure
endif
Listing 2.48: Rule R4.4: Workunit failed

Rule R4.5: Abort workunit
2.4. A formal model of BOINC

If an abort event is received from the user abort all results of the workunit and the workunit itself.

```plaintext
if (event(j) = abort ∧ role(j) = workunit ∧ jobPhase(j) = running) then
    jobPhase(j) := failure
    forall jj in JOB: parent(jj) = j ∧ role(jj) = result ∧ jobPhase(jj) ≠ finished do
        event(jj) := abort
    endforall
endif
```

Listing 2.49: Rule R4.5: Abort workunit

**Rule R4.6: Clean up after workunit**

The workunit can be cleaned up if it is either failed or successfully terminated, and all its results have finished and the submitting users allows the deletion. This user approval is represented by the readyForDelete monitored function introduced in $M_{GROUND-DG}$.

```plaintext
if (jobPhase(j) = filesDeleted ∧ role(j) = workunit ∧ readyForDelete(j) = true): ¬∃jj ∈ JOB: parent(jj) = j ∧ role(jj) = result ∧ jobPhase(jj) ≠ finished then
    forall jj ∈ JOB: parent(jj) = j ∧ role(jj) = result do
        JOB(jj) := undef
    endforall
    JOB(j) := undef
endif
```

Listing 2.50: Rule R4.6: Clean up after workunit

**Rule R4.7: Workunit ready for assimilation**

Workunit is ready for assimilation. Set to ready by transitioner if assimilateState(j) = init and workunit has error condition. Set to ready by validator when it finds a canonical result and assimilateState(j) = init.

```plaintext
if (role(j) = workunit ∧ assimilateState(j) = init ∧ (errorMask(j) ≠ undef ∨ canonicalResult(j) ≠ undef)): assimilateState(j) := ready
    jobPhase(j) := readyForAssimilation
endif
```

Listing 2.51: Rule R4.7: Ready for assimilation

**Rule R4.8: Workunit assimilated**

Workunit was assimilated. Set to done by assimilator when assimilation has finished.

```plaintext
if (assimilateState(j) = ready): assimilateWorkunit(j)
    assimilateState(j) := done
    jobPhase(j) := assimilated
endif
```

Listing 2.52: Rule R4.8: Workunit assimilated

**Rule R4.9: Workunit files ready for delete**

Workunit inputs are ready to be deleted. Set to ready by transitioner when all results have serverState(jj) = over and the workunit has assimilateState(j) = done. From the BOINC wiki "Note: db_purge purges a WU and all its results when file_delete_state = DONE; therefore it is critical that it only be set to DONE if all results have server_state="
OVER.

if (role(j) = workunit ∧ testAllResults(j, serverState, over) = true ∧ assimilateState(j) = done):
    fileDeleteState(j) := ready
    jobPhase(j) := readyForFileDelete
endif

Listing 2.53: Rule R4.9: Workunit files ready for delete

rule testAllResults(job, function, value) :=
    return := false
    if role(job) = workunit ∧ ∃res ∈ JOB: result(job)=res:
        return := true
        forall res ∈ JOB: result(job) = res:
            if function(res) ≠ value:
                return := false
                break
        endforall
    endif
endif

Listing 2.54: Helper rule for testing all results of a workunit with a given function

Rule R4.10: Workunit files deleted
Workunit files were deleted. fileDeleteState is set to done by file_deleter when it has attempted to delete files.

if (fileDeleteState(j) = ready):
    deleteInputFiles(j)
    fileDeleteState(j) := done
    jobPhase(j) := filesDeleted
endif

Listing 2.55: Rule R4.10: Workunit files deleted

Rule R4.11: Job staging
This rule is a refined version of R4.11. Only results can be mapped to physical resources and abstract resource requests belong to the application and workunit part of the job instead of itself. The resource request of the unit of work and application are mapped together, where the first is an abstract resource request and the request of the latter is consolidated into a platform. If the host can provide the needed resources the job is mapped to the host. This also marks the start of the timer of the job (denoted by the sentTime function) that counts towards delayBound and determines when is a job considered delayed (see Functionality F2.5).

if (∃ar ∈ ARESOURCE, uow ∈ UNITOFWORK):
    uowofwork(j) = uow ∧ resourceRequest(uow, ar) = true ∧ mappedResource(uow, ar) ≠ undef then
        let pr := mappedResource(j, ar)
        if (usermapping(globaluser(j), pr) ≠ undef) then
            mapped(j) := location(pr)
            resourceRequest(uow, ar) := false
            sentTime(j) := serverTime(j)
        endif
    endif
endif

Listing 2.56: Rule R4.11: Resource selection
Rule R4.12: Task installed and physical resources allocated
This rule is a refined version of R2.9. The result is ready to run after the required resources are selected (see Rule R2.8) and the job is mapped and installed on a host. This is described by the following:

```plaintext
if (∃h ∈ HOST, ∃pr ∈ PRESOURCE, ∃wu ∈ UNITOFWORK, ∃app ∈ APPLICATION):
    mapped(j) ≠ undef ∧ uses(j, pr) = true ∧
    jobPhase(j) = init ∧ role(j) = result ∧ jobPhase(parent(j)) = readytorun then
        extend TASK by t with
            task(j) := t
            installed(t, location(pr)) := true
        endextend
        checkpoint(j) := false
        jobPhase(j) := readytorun
endif
```

Listing 2.57: Rule R4.12: Task installed and physical resources allocated

Rule R4.13: Start result
Rule R4.14: Stop result
Rule R4.15: Suspend result
Rule R4.16: Resume result
Rule R4.18: Result fails
Rule R4.24: Abort result
The rules for starting, stopping, suspending, resuming and aborting results; result failure and resuming from checkpoint are the same as their corresponding ones in M_{VC-VOTE}.

Rule R4.21: Result termination
Result termination is realized by the following:

```plaintext
if (jobPhase(j) = running ∧ event(j) = terminate ∧
    role(j, instance) = true ) then
    uses(j, presource(j)) := false
    jobPhase(j) := finished
    jobOutcome(j) := success
    needValidate(j) := true
    TASK(task(j)) := undef
endif
```

Listing 2.58: Rule R4.21: Result termination

Rule R4.19: Perform checkpointing
Rule R4.23: Resume from checkpoint
These rules are the same as their corresponding parts in M_{VC-VOTE} (see Rule R2.15 and R2.17) is the same as BOINC does not provide a central checkpointing facility rather the applications must implement their own method for saving their states periodically. The restart from a saved state (checkpoint) must be also implemented by the application. The only support BOINC provides is a signal for the application that it should checkpoint if possible. In this sense the capability(j,ckpt) function (see Rule R1.10 of M_{GROUND-DG}) only depends from the application part of the job. If the application instance supports checkpointing then the job supports it. Also the createCheckpoint(j) function is provided by the application instance. Finally the readyToCheckpoint(j) function is a signal (i.e.
a boolean variable in a structure stored a shared memory segment) sent by the BOINC Client to the running application.

2.4.6 Correspondence of $M_{VC-VOTE}$ and $M_{BOINC}$ ASM models

Correspondence of $M_{VC-VOTE}$ and $M_{BOINC}$ can be shown similarly as the correspondence of $M_{GROUND-DG}$ and $M_{VC-VOTE}$(see Section 2.3.3.2). First the states of interests need to be defined:

**Definition 8** The following states denote the states of interest for models $M_{VC-VOTE}$ and $M_{BOINC}$:

$$S_{INTEREST}^{\{A|B\}} := \{SIN, SI, SPE, SE, SF, ST\}$$

(2.100)

The states of interests defined in Definition 8 are used as coupling invariant for both models and represent the following:

- **SIN**: Initial state of the model. All visible jobs (per Definition 5-6) are $JOB(j) = undef$. (Refer to Section 2.3.2.2 for more details.)

- **SI**: All visible jobs (per Definition 5-6) $j$ are in init state: $S_{\{A|B\}}^{\{A|B\}}(j) = init$.

- **SPE**: Pre-execution state. All visible jobs (per Definition 5-6) $j$ are ready to run: $S_{\{A|B\}}^{\{A|B\}}(j) = readytorun$.

- **SE**: Execution state. All visible jobs (per Definition 5-6) $j$ are running: $S_{\{A|B\}}^{\{A|B\}}(j) = running$.

- **SF**: Finished state. All visible jobs (per Definition 5-6) $j$ have finished: $S_{\{A|B\}}^{\{A|B\}}(j) = finished$.

- **ST**: Terminated state. All visible jobs (per Definition 5-6) $j$ have terminated and were discarded: $JOB(j) = undef$.

The same computation segments are defined as in Section 2.3.3.2 ($SEG_{INITIALS}$, $SEG_{INIT}$, $SEG_{START}$, $SEG_{STOP}$, $SEG_{EXEC}$, $SEG_{TERM}$ and $SEG_{ABORT}$). Similarly, using the approach presented in Section 2.3.3.2 their correspondence can be shown thus, I omit detailing it again here. The correspondence is as follows:

**Statement 7** There is a correct refinement of $M_{VC-VOTE}$ to $M_{BOINC}$:

$$M_{VC-VOTE} \triangleright M_{BOINC}.$$
2.4.7 Discussion

The presented rules and functionalities constitute a formal model of BOINC:

**Statement 8** Rules R4.1 - R4.24 and functionalities F4.1 (Resource mapping), F4.2 (User abstraction), F4.3 (Result certification), F4.4 (Mapped resources become unavailable), F4.5 (Job is delayed), and F4.6 (Donor interaction) constitute a reference model of BOINC (M\textsubscript{BOINC}) under the assumptions made in Section 2.4.

The following characteristics were identified at the beginning of the section (see Section 2.4) and used in this model (M\textsubscript{BOINC}):

1. **BOINC follows a centralized client-server architecture.**
   
   The connection between different nodes is defined by the initial state rules 2.33, 2.34 and 2.35 of M\textsubscript{GROUND–DG} that inherits M\textsubscript{BOINC} through the initial state rule 2.93 from M\textsubscript{VC–VOTE}. The initial state rules 2.96, 2.97, 2.98 and 2.99 ensure that all components (except the workers) are located on the same node and there is a single ui, manager and repository in the project.

2. **Applications cannot be submitted as part of jobs. Jobs must refer a previously at the project deployed application.**

   That all jobs must have an application is enforced by the initial state rule 2.60 of M\textsubscript{VC–VOTE} which is in initial state rule of M\textsubscript{BOINC}. Also this property is enforced during job submission by the \textit{submitted} : JOB × USER × UI × MANAGER → {true, false} function of Rule R4.1 where the isDeployed function checks if the application the job refers is deployed at a repository connected to the manager where the job was submitted to (uiOf(mgr) = u).

   The rule actually does not enforce that no application binary is submitted with the job, it only makes sure that the job refers to an application that is deployed in the repository of the project. This is actually how BOINC works. Namely any file (executable and library) can be submitted with the job, however all submitted files will be treated as inputs for the referred application and not as an application.

3. **The resource requirements of applications are consolidated into resource groups.**

   These resource groups are referred as platforms, represented by the \textit{PLATFORM} universe. Each application can support multiple platform (with a different implementation for each) and also hosts can support multiple ones (e.g., a Windows 64bit host can run both 32 and 64 bit Windows applications). This is represented by the \textit{supportsPlatform} : \{HOST, APPLICATION\} × PLATFORM → {true, false} function. The platform is enforced with other resource requirements during resource mapping in the Π\textsubscript{resource_mapping} functionality (see Functionality F4.1). There both the application and host must support the same platform (∃plat ∈ PLATFORM : supportsPlatform(app, plat) = true ∧ supportsPlatform(h, plat) = true) in order
to map the job to the host. The physical resources are mapped to the unit of work part of the job \( (\text{mappedResource}(\text{uow}, \text{ar}) := \text{pr}) \) while the platform is mapped directly to the job \( (\text{mappedPlatform}(j) := \text{plat}) \). This is since applications have many platforms (denoted by the \( \text{platform} : \{\text{APPLICATION, HOST}\} \to 2^{\text{PLATFORM}} \) function) and the mapped platform is a property of the job.

4. **BOINC implements a result certification mechanism based on comparing returned finished job instances.**

   The validation mechanism is provided through a core functionality (see \text{doValidate} macro on Listing 2.40 at Functionality F4.3). Using this core functionality each application must provide a validator that is responsible for the jobs that use the application. In these validators the \text{checkSet} and \text{checkPair} rules must be provided that set the \text{validate} of the results either to \text{valid}, \text{invalid}, or \text{inconclusive}.

5. **BOINC may create additional job instances during the running phase of an abstract job.**

   BOINC does not separate initial and additional result creation. All results are created by Rule R4.2. The job creation condition is provided by the \text{needResults} function. The number of required results is determined by the \text{targetNResults} function of the workunit. Totally there should be \text{targetNResults} number of running (represented by \text{runningResults}), unsent (\text{unsentResults}) and successful (\text{successResults}) results combined. However the \text{doValidation} macro of the validator (see Functionality F4.3 and Listing 2.40) belonging to the application of the job will increase this number if there are enough successful results to form a quorum (\( \text{count} (\text{successResults}(j)) > \text{minQuorum}(j) \)), but still no quorum can be formed by the \text{checkSet} function. This increase will evaluate \text{needResults} to true and thus additional results are going to be created by Rule R4.2 for the workunit.

6. **Access to a host can be restricted by its owner based on a per application basis.**

   Whether the application is allowed or not on a certain host of a donor is determined by the function \( \text{appAllowed} : \text{DONOR} \times \text{APPLICATION} \to \{\text{true, false}\} \). This preference is taken into account during the resource mapping phase of a job, which is provided by Functionality F4.1: Jobs can be mapped only to those available hosts that provide the required resources for the unit of work part of the job, have a compatible platform for the application of the job and their donor preferences allow running the application on the host.

7. **Donors are granted virtual credits for their contributed resources.**

   This is ensured by the validator of each application (see Functionality F4.3). The credit granting is performed by the \text{grantCredit} macro used in the \text{doValidation} macro (see Listing 2.40). The guards ensure the macro is only used for granting credits for successful and valid results provided by the donors. The actual amount and specifics of the credit granting are usually project specific.
This chapter discusses possible future directions and challenges of VC. In the first part (see Section 3.2) I make an attempt – starting from a definition for Cloud Computing – to identify the required steps and formulate a definition for what can be considered as the next evolutionary stage for Volunteer Computing: Volunteer Cloud Computing (VCC). While Volunteer Computing uses volatile, heterogeneous and unreliable resources Cloud Computing is built using dedicated and reliable resources and provides uniform seemingly unlimited capacities. There are many idiosyncrasies of VC to overcome (e.g., volatility, heterogeneity, reliability, responsiveness, scalability, etc.). In this section I am investigating heterogeneity as it is not just crucial as a future direction, but also has a large impact on VC in its current state. Heterogeneity exists at multiple levels. For example the donated resources are heterogeneous. First they have a diverse set of hardware (e.g., CPU type, features and speed; available memory, bandwidth and disk, etc.); and second they run a diverse set of software components (e.g., operating system, installed libraries, etc.). Middleware can have their own set of requirements for applications in order to be compatible with them. Applications on their own have requirements (e.g., depend on specific software libraries and packages). Contrary to these the vision of Cloud Computing promises to provide a homogeneous environment. My goal in the first part of the chapter is
to identify methods and propose solutions that tackle the different aspects of heterogeneity and thus, bring the vision of Volunteer Computing Clouds closer.

In the second part (see Section 3.3) I propose methods to improve responsiveness and implicitly performance of VC projects. As mentioned previously VC is suited mainly for embarrassingly parallel or parameter study applications. These are usually run in form of batches where each batch represents an experiment. The total execution time of the batch from the submission until the finish of the last job is referred as $makespan$. The $makespan$ of a batch is determined by the slowest (i.e., last finishing) job. The volatility and unreliability of the resources of VC can lead to prolonged makespan by some jobs finishing slowly; sometimes even invoking the \textit{delayed jobs} functionality of VC (see Functionality F4.5 in $M_{BOINC}$) thus, delaying the completion of the batch. This phenomena is referred as the “tail-effect” \cite{11} or “long tail” in general. I propose two methods derived from real-world use cases involving different BOINC projects. The first method identifies belated jobs and increases their instance count as needed (see \textit{targetNResults} in $M_{BOINC}$). The second method uses on-demand reliable resources from different academic and public clouds to compute delayed jobs. The two real-world scenarios are the following:

1. **EDGeS@home**: EDGeS@home \cite{73} is a “umbrella” project. It hosts multiple applications and serves multiple user communities. Its aim is to provide “Quality of Service” (QoS) to its communities by identifying and “accelerating” belated jobs in a non-interactive manner. Acceleration is achieved by redirecting problematic jobs to reliable on-demand resources provided by 5 academic partners and a single industrial provider. Validation is performed using the PublicAutodock \cite{74} application (DG/VC implementation of the well-known AutoDock \cite{75} tool).

2. **KOPI Cross-Language Plagiarism Search**: The KOPI Portal \cite{76} uses SZTAKI Desktop Grid (SZDG) VC project \cite{77} for pre-processing the articles from different language Wikipedia. KOPI and SZDG are operated by separate departments. KOPI has no influence on the internals and job scheduling of SZDG. The first proposed algorithm identifies belated jobs based on characteristics of the batch and increases their replication as needed to ensure makespan reduction. This approach treats the VC project as a black box with no knowledge of its internal workings.
3.1 Towards federating volunteer computing

In this section an abstract model is introduced for federating desktop grids and volunteer computing systems (VCSs). In this case the fourth model $M_{BOINC}$ is used as a basis. The correspondence of $M_{GROUND-DG}$ and $M_{VC-VOTE}$ was shown in Section 2.3 and the correspondence of $M_{VC-VOTE}$ and $M_{BOINC}$ in Section 2.4. The extensions shown here can be applied to $M_{GROUND-DG}$ as well. In this sense use cases will be presented both for volunteer computing and desktop grids however the federation model will be based on $M_{BOINC}$. However the federation should not need to be homogeneous in terms of middleware used. The model is referred as $M_{FED-BOINC}$.

Desktop grids and volunteer computing systems operate isolated and independently from each other. Additionally they provide jobs periodically thus, they are not able to utilize fully all their donated resources. These resources (if they are connected to multiple projects) will download tasks from others making the resource potentially unavailable for the project when it has work the next time. If these donated resources are not connected to other projects, then their compute capacity for the period is lost as compute capacity cannot be stored.

In this section I present an automatic application deployment method and security mechanism I developed as part of $M_{FED-BOINC}$. They enable the interoperability of tasks between projects of volunteer computing and desktop grid systems. I present the developed mechanisms via the BOINC volunteer computing system and one of the identified scenarios:

\begin{center}
\textbf{Theorem II.1.}: The automatic application deployment method I developed makes the transfer of tasks between desktop grids and/or volunteer computing systems possible.
\end{center}

Related publications: [78], [20], [79], [80], [81], [21], [82], [19], [83], [18], and [17].

3.1.1 Federation at a glance

The term federation is used as a specific method for interconnecting distributed computing systems. For a generic interconnection the Inter-* phrase can be used. For example for the generic interconnection of clouds the term Inter-Cloud was created and formulated by the Global Inter-Cloud Technology Forum [84]. This definition can be adapted for the notion of interconnected volunteer computing systems:

\begin{center}
\textbf{Definition 9} Inter-Volunteer Computing is a volunteer computing model that, for purpose of guaranteeing service quality (e.g., performance), allows on-demand transfer of workload through the collaboration of volunteer computing systems based on the coordination of each users requirements for service quality and the use of standard interfaces.
\end{center}
Based on this definition Figure 3.1 shows the different possible interconnected architectures for volunteer computing adapted from cloud construct architectures defined in [85] separated into two groups: In multi-constructs the different volunteer computing systems are accessed in a centralized manner which can be either a multi-access service (see a. in Figure 3.1) where users can access multiple VCSs through a single service or a meta-middleware library (see b. in Figure 3.1) that users can use to develop their own brokers to access multiple infrastructures. In contrast federations allow the infrastructures to collaborate with each other. This can be achieved either by using a central component (see c. in Figure 3.1) to facilitate workload distribution or by directly in generally using some peer-to-peer manner (see d. in Figure 3.1). While multi-constructs can be established by third parties using APIs or tools provided by the different VCSs, federations are mainly volunteer formations that require the agreement between the two or more parties. Basically a federation is established when a set of providers interconnect their infrastructures to share resources among each other\footnote{http://www.arjuna.com/what-is-federation} [86]:

**Definition 10** A federation of volunteer computing systems is formed through the direct interconnection of the systems based on the agreements of their providers for workload sharing.

![Figure 3.1: Possible architecture types for interconnecting volunteer computing systems](image)

3.1.1.1 Introduction of scenarios

In this chapter I present the formal model of a method that aims at achieving a subset of the federation methods shown in Figure 3.1. For a deployment of a volunteer computing middleware the BOINC deployment term “project” will be used from now on. I identified three real world scenarios for federation that my method and model aims at solving. Figure 3.2 depicts these. There the directed edges denote the direction of the workflow transfer, thus the “flow” of jobs. Contrary to the federations part of Figure 3.1 where the direction is not specified. The federated projects are ordered into levels based on their distance from the root of the graph (see L0, L1, etc. on Figure 3.2). The scenarios are as follows:

**Scenario 1: Hierarchy of desktop grids in institutes and universities**  DGs do not collaborate with each other since they are usually deployed within different organizational
3.1. Towards federating volunteer computing

There can be multiple DGs present within an institution, e.g., used by different departments or faculties. Institutes and companies have mostly a hierarchical organization structure (e.g., departments form a faculty and faculties form the university as whole). Combining the resources of the different organizational units in a hierarchical way similar to their administrative structure seems self-evident. For example in universities departments can be considered as the lowest element in the hierarchy (e.g., see DG_D and DG_E in Figure 3.2/a.). Each department has it own computing resources, e.g., office machines or computers in laboratories. The unused resources of these can be utilized by a department-wide desktop grid. A faculty consists of multiple departments and the department-wide DGs can be joined together into a faculty wide one (e.g., see DG_B in Figure 3.2/a.). This allows combining the resources of the different departments that would allow solving larger scientific challenges faster. Finally the different department-wide DGs can be joined into an university DG (e.g., see DG_A in Figure 3.2/a.). The collaboration of deployments are solved in some extent by middleware specific methods (e.g., the flocking mechanism of Condor [50]), but for example creating a collaboration that honors the hierarchical structure of the organization is troublesome. Such a hierarchical structure would allow that each organizational unit retains the control of their resources, but still the resources are pooled at a higher level and made available for larger compute intensive challenges.

**Scenario 2: Company supporting a volunteer computing project** Companies and institutes generally have many office computers that are not utilized to their full potential during the daily office hours and are turned off for the non-office hours, thus a lot of computing capacity is wasted. This capacity could be used to support a volunteer computing project. Usually most companies prohibit direct (or any) access to the internet for their computers thus, prohibit access to any VC project. This can be solved by opening access to the selected VC projects (either directly or via some proxy network solution) however, this is not desired since: (i) all hosts must have access to the VC project; (ii) for each new VC project network access must be granted to each host; (iii) each host must be individually connected to each new VC project and (iv) there is no easy (centralized) method for monitoring the flow and status of the jobs. For this scenario a “Proxy” DG project within the institute that has no own application only transfers jobs from the selected VC projects is a possible solution. In this case all hosts can be connected to this
proxy project and solving problems i–iv. The scenario is depicted in Figure 3.2/b.

**Scenario 3: Volunteer computing projects collaborating** BOINC is currently the most popular VC framework with more than 2.5 Million donors and 8.5 Million hosts in 80 distinct volunteer projects [68]. Each of these has its own grand scientific goal and does not collaborate with others. Some of these projects provide jobs for their donors 24/7 and would require more resources. On the other hand some projects provide jobs in batches (e.g., they run distinct experiments that do not overlap in time) and are not able to supply all their donors continuously with work. These are currently mitigated by the donors by signing up and contributing to multiple projects. Volunteer projects are either created specifically for solving a single scientific problem or nurturing multiple third party applications. In the latter case they are referred as umbrella projects. In either case they may not be able to provide jobs for their volunteers continuously as jobs usually arrive in batches. In such cases an ”idling” project could transfer part of the workload to help out the other project. The scenario is depicted as Figure 3.2/c. Although the graph of this scenario contains one or possibly more cycles it is assumed that the workload transfer is unidirectional, thus it always flows in one direction at once. For example workload from VC$_A$ is only transferred to VC$_B$ only if VC$_B$ is willing to accept it. In this case there is no reason to transfer back the same workload in parallel. Thus any cycle in the graph will not cause problems. It is assumed that the federation is used for workload transfer (e.g., in case it has free capacity one project accepts jobs from another) thus jobs won’t traverse the graph in cycles (back and forth between parties) under normal circumstances.

### 3.1.1.2 Forming federations

Based on Definition 10 the basic concept behind federations is allowing the transfer of workload between parties. It can be split into multiple relations of parties of two where one is allowed to transfer workload to the other:

**Definition 11** For VC$_A$ and VC$_B$ projects let VC$_A$→VC$_B$ denote that VC$_B$ accepts workload transfer form VC$_A$ thus, accepts jobs from it. In this relation VC$_A$ is referred as source and VC$_B$ is referred as destination.

**Definition 12** For each job $j \in JOB$ let origin denote the project of the manager where it was first submitted to in the federation.

**Definition 13** A federation of VC or DG projects can be described as a set of pairwise relationships of workload transfer referred as blocks.

**Definition 14** Let VC$_A$⇌VC$_B$ denote the mutual workload transfer allowed between VC$_A$ and VC$_B$ projects. That is: VC$_A$⇌VC$_B$ ⇐⇒ VC$_A$→VC$_B$ ∧ VC$_B$→VC$_A$.

Since workload transfer is unidirectional it is not symmetric (VC$_A$→VC$_B$ ≠ VC$_B$→VC$_A$), but it can be considered reflexive (VC$_A$→VC$_A$) and transitive (VC$_A$→VC$_B$, VC$_B$→VC$_C$ ⇒ VC$_A$→VC$_C$). However mutual workload transfer is bidirectional therefore:
Definition 15 The relation of mutual workload transfer (denoted by $\leftrightarrow$) is symmetric, reflexive and transitive.

Based on Definitions 11-15 the scenarios depicted in Figure 3.2 are as follows:

$$FED_a = \{DG_A \leftrightarrow DG_B, DG_A \to DG_C, DG_B \to DG_D, DG_B \to DG_E\}$$ (3.1)
$$FED_b = \{VC_A \to DG_A\}$$ (3.2)
$$FED_c = \{VC_A \leftrightarrow VC_B\}$$ (3.3)

The details of when and what part of a workload is transferred to another party – e.g., scheduling related topics – are specifics of the agreement between the two parties and is out of scope for this work.

The transferred workload can contain either abstract jobs or job instances. For example in case of $M_{GROUND-DG}$ – where there are no roles for jobs – all jobs act as both instances and abstract ones, there is no possibility to transfer only their abstract part thus, for this categorization they should be threat as instances. Abstract jobs can be either copied or moved to the destination. Copying means that the source can still process them while a copy is at the destination. Moving means that the source cannot process the job after the transfer. Copying a workload seems “unorthodox” since the workload is not transferred literally however, it may speed up execution since both the source and destination VC projects can process it. The method of the transfer can be either intrusive or non-intrusive. The former means that the manager component both of the source and destination projects require some additional functionality for establishing the workload transfer between the parties. Non-intrusive means that only one party is required to have the additional functionalities. I will focus on the Job Instance/Non-intrusive type of workload sharing:

Definition 16 Let $M_{FED-BOINC}$ denote the extended $M_{BOINC}$ model that contains all needed functionalities required for receiving workload in a federation using the non-intrusive and job instance transfer method.

The previous formal models for DGs, VC and BOINC were depicted from the point of view of a job in the system. Figure 3.3/a depicts the path of a job through the different nodes in those models. The job is submitted from a $ui$ to a $manager$ and executed by a $host$, after termination or abort it is returned to the $manager$ and finally to the $user$ through an $ui$. The nodes are represented by the solid rectangles with their type ($ui$, host, manager) inside. The dashed gray rectangle depicts what is described by the job state transition rules in the different models. In Figure 3.3/b the life-cycle of the job is presented for a block (see Definition 13) in a federation. The extended model ($M_{FED-BOINC}$) interacts with the BOINC model ($M_{BOINC}$) and these two form a block. In $M_{FED-BOINC}$ (see Figure 3.3/b) a new type of node is introduced that has a dual role: the gateway. It acts as a middleman. It is a modified host for one project (source) and is a $ui$ for another (destination) providing a 1:1 mapping between source and destination. Therefore:
Figure 3.3: The path of a job through nodes during its state transitions in the different models: (a) $M_{GROUND-DG}$, $M_{VC-VOTE}$, $M_{VC-SPOT}$ and $M_{BOINC}$; (b) $M_{FED-BOINC} \rightarrow M_{BOINC}$

**Definition 17** In a block $BOINC_A \rightarrow BOINC_B$ – where $BOINC_*$ denotes BOINC middleware based projects – $BOINC_A$ is always represented by the model $M_{FED-BOINC}$, and $BOINC_B$ by $M_{BOINC}$ thus, the block can be regarded as $M_{FED-BOINC} \rightarrow M_{BOINC}$.

**Definition 18** The block $BOINC_A \leftrightarrow BOINC_B$ is described by $M_{FED-BOINC} \leftrightarrow M_{FED-BOINC}$.

The principles presented here can be generalized to provide N:1, 1:N or N:N mapping between source and destination project(s) or even models of different (desktop) grid middleware [45]. However this thesis is restricted to the formal models and scenarios presented. First in the following section a method for federating desktop grid projects is presented through the first scenario described in Section 3.1.1.1: hierarchy of desktop grid projects. Later the additional constraints for $M_{BOINC}$ regarding the federation model $M_{FED-BOINC}$ are discussed.

Based on the scenario presented, relying on the application deployment method, several scheduling algorithms were proposed and evaluated for the hierarchical Desktop Grid by Zoltan Farkas in [87], [88] and in his PhD thesis [89]. Evaluation was done with his custom developed simulator “HierDGSim” [90]. Thus further evaluation – I think – is not required here for the scenario.

### 3.1.2 Hierarchy of desktop grids

The hierarchical desktop grid allows a set of projects to be connected in a form of a directed acyclic graph. The projects are the vertices and work is distributed along the edges. The projects are ordered into levels based on their distance from the top level (see Figure 3.4).

In a hierarchy every project is in a parent-child relationship. A project may request work from a project above (child) or may provide work for a project below (parent). The hierarchical interaction is always between a parent and a child regardless of how many levels of hierarchy are above or below. For a child every job is originating from its parent regardless where it is originally from or where were the input data for the job fetched (although the data are not always from the parent). It is allowed for a project to have more children and parents. Figure 3.4 shows a three-level example.
3.1. Towards federating volunteer computing

A project in the hierarchy serves two roles (see Figure 3.5): a master side which puts retrieved jobs in the database of the project and retrieves the completed results, and a client side which downloads jobs from the parent and uploads final results. Extending BOINC with this functionality allows providing basic hierarchical functionality without further (deeper) modifications, but it has drawbacks:

1. The application of the job has to be deployed previously at each project. All implementations of the application.
2. Application mapping between source and destination is required.
3. Work distribution is based on the local scheduling method implemented by BOINC, that might not be ideal in a hierarchical setup as it was not designed for this task.

3.1.2.1 Extending M_{BOINC} for use in hierarchy

Although the hierarchy prototype presented is very simple and was easy to implement, it had a major drawback: applications must be installed manually at each child in order
to be able to process jobs originating from the parent. Overcoming this limitation also requires replacing of the security model of BOINC.

The most important factor in desktop grid computing is the trust between the clients and the project providing the application. Allowing foreign code to run on a computer always has a risk of either accidental or intended misbehavior. BOINC mitigates this risk by only allowing to run code that has been digitally signed by the project the client is connected to. Clients trust the operators of the BOINC project not to offer malicious code, and digitally signing the application provides technical means to ensure this trust relation.

Of course it is not enough to only sign the application binary, the input data must be signed as well (think of the case when the application is some kind of interpreter and the input data can instruct it to do just about anything). Therefore BOINC uses two separate key pairs: one is used to sign the workunits (which in this context means the set of input files and a link to the application binary), the other is used to sign the application code. The private key used for workunit signing is usually present on the project’s central server, while the private key used for application signing is usually kept at a separate location. The different handling of the private keys stems from their usage pattern: the workunit signing key is used very often while the code signing key is seldom needed therefore it can be protected better. This technique significantly reduces the risk of compromising the application signing key even if the machine hosting the project is compromised, but this also means that installing new applications is a manual process – which is unfortunate for a hierarchical setup.

Therefore, solving the automatic application deployment issue presents two challenges:

- a lower-level project in a hierarchical desktop grid system must be able to automatically obtain an application’s binary from its parent and be able to offer the application to its clients without manual intervention, and
- this process must not increase the risk of injecting untrusted applications into the system.

These requirements mean that a lower-level project can not simply re-sign the application it has obtained from the parent, since that would require the private key to be accessible on the machine hosting the lower-level project which in turn would significantly increase the risk of a key compromise if the machine hosting the project is compromised.

3.1.2.1.1 Extending the security model to support hierarchy

As discussed above the security model used by BOINC is not adequate in a hierarchical setup and a new model is needed. The model must provide enough information for the operator of the client machine (Donor from now on) to decide if a downloaded workunit should be trusted to run on the client machine or not, independent from where in the hierarchy the workunit is originated from. The model must provide enough information for the following decision scenarios:
3.1. Towards federating volunteer computing

1. The *Donor* wants to trust any workunits of applications installed locally on the BOINC project she is directly connected to (i.e., the *Donor* trusts the project itself). This is the original trust model of BOINC.

2. The *Donor* wants to trust any workunits from a given project, regardless of how many levels of hierarchy did the workunit travel through. This is in fact a generalization of the previous requirement.

3. The *Donor* wants to trust a specific application regardless of where in the hierarchy it is hosted and regardless of what other applications does the hosting project offer.

The $t(⟨subject⟩, ⟨object⟩)$ trust relation for a workunit can be broken down to three parts:

1. Trusting the application code: $t(Donor, App)$.
2. Trusting the set of input files: $t(Donor, Input)$.
3. Trusting the link between the application, its inputs and the desired location of its outputs to prevent the application from processing data that was meant for an other application: $t(Donor, ⟨App, Input, Output⟩)$. I will use the shorthand $WUDesc$ for the $⟨App, Input, Output⟩$ triplet.

A workunit $WU$ is trusted if all components are trusted:

$$t(Donor, App) \land t(Donor, Input) \land t(Donor, WUDesc) \rightarrow t(Donor, WU) \quad (3.4)$$

The trust relation is realized by digital signature verification. Therefore, each of the three classes of objects $App$, $Input$ and $WUDesc$ are accompanied by one or more digital signatures:

$$Sig_X : X \in \{App, Input, WUDesc\} \quad (3.5)$$

It is assumed that $Donor$ has a set of trusted identities marked $TrustedID_{Donor}$. Thus the trust relation becomes:

$$t(Donor, X) \iff \exists s \in Sig_X : verify\_sig(X, s) \land subject\_of(s) \in TrustedID_{Donor} \quad (3.6)$$

The $subject\_of(s)$ function provides the identity that created the signature $s$. Special elements ($Any_X$) which satisfy the relation are also allowed:

$$Any_X : X \in \{App, Input, WUDesc\} \quad (3.7)$$

$$\forall s : verify\_sig(Any_X, s) = TRUE \quad (3.8)$$

The $Any_X \in TrustedID_X$ means that the user does not require a valid signature for that particular component.
Realization We decided to use the X.509 Public Key Infrastructure, since it is a widely accepted and used infrastructure that provides all the technical elements we need. Therefore, the TrustedID\textsubscript{Donor} set becomes a list of X.509 certificates.

We define 3 entities responsible for signing various components of the system. The Application Developer (AppDev from now on) can sign application code. This kind of signature testifies that the application binary comes from a known source and does not contain malicious code. The Project is the administrative body of the BOINC project and it may also sign application code testifying that said application is in fact part of the project. The Server is the machine where the project is hosted, and it signs input files and workunit descriptors. Using the original BOINC terms the AppDev provides the code-signing key, while the Server provides the workunit-signing key.

The TrustedID\textsubscript{Donor} list of trusted certificates must be determined by the user, since the trust is ultimately a human relation. This may be simplified by the Project by providing a list of Server and optionally AppDev certificates it trusts – this means the user can delegate the trust to the Project. This realizes the first scenario described in 3.1.2.1.1. The second scenario is realized if the Project also provides the aggregated list of certificates from all levels above it in the hierarchy. The third scenario is realized if the user lists only the certificate of the appropriate AppDev and specifies that she does not care about the signature of Input or WUDesc.

We described a model how a user can trust work received from a hierarchical desktop grid system. In a restricted environment however more is needed: it is not enough for the user to trust the workunit, but the project must also trust the user before it gives out possibly confidential information. Also it is not enough just to trust the receiving user, but the data also has to be protected from being disclosed to untrusted parties. This is a new requirement that is not present in public projects.

Protecting the confidentiality of the data can be easily achieved. BOINC by default uses plain HTTP protocol for communication, but it also supports the HTTPS protocol where the communication is encrypted. The Server certificate can be used with the HTTPS protocol to ensure that the Donor in fact talks to the server she thinks is talking to. Although BOINC uses a simple shared-secret based authentication scheme to identify users, this authentication applies only to interactions with the scheduler. Together with the use of HTTPS this may be adequate to prevent unauthorized users from uploading results, but it does not prevent unauthorized users to download application code and input data if they are able to guess the file name used on the server.

The protection of input data from unauthorized download can be achieved by giving every user a certificate. The Project can act as a Certificate Authority and can sign the certificates of all authorized users. Then, the web server that is used for downloading the input files can be configured to only allow downloading if the client authenticated itself with a properly signed certificate.

The workunits are always signed by the server running a specific project, so the projects need a way to make their known and accepted signing certificates available for their clients and other projects. This is solved by an extension to the web based interface of the BOINC project allowing to query for the certificates via the HTTP(S) protocol and depending on the trust model described in 3.1.2.1.1.
3.1. Towards federating volunteer computing

3.1.2.1.2 On-demand application deployment

BOINC allows the creation of a workunit that refers to external servers for the input files. This means that lower-level projects in a hierarchy do not need to install the input files locally, they may just refer to the original location of the files in the workunit description. However, due to security considerations BOINC does not allow to refer to outside of the project for application binaries, they must always reside on the project’s server. Thus, lower-level (child) projects must deploy all applications whose workunits they offer locally.

The automatic deployment of applications presents two problems. The first problem arises from the need to properly sign the binary and is solved by the introduction of the AppDev role as described in the previous section. If the users have configured their TrustedIDDonor sets to contain the appropriate certificate of the AppDev, then the project does not need to sign the application binary, thus its secret key is not needed for application deployment.

The second problem arises from the fact that BOINC uses the \( \langle \text{AppName}, \text{Version} \rangle \) tuple to identify applications and in a complex hierarchy it is possible that at different levels different applications are installed under the same name. This problem can be solved by automatically renaming the application when a workunit is transferred from a parent to lower level child project. Using an Universally Unique Identifier (UUID) as the new application name ensures that there will be no name collisions.

For the following we assume that the application consists of just a single binary. Compound applications or applications with accompanying shared libraries are not considered in this report.

The Gateway (see Section 3.1.1.2) keeps track of the name mapping of the application between parent projects and child project. Such a renaming is possible because on the server side only the workunit-generating master application cares about the name of the application, and in this case this master application is the link between the members of the hierarchy and therefore has full control. The UUID is generated by the hierarchy after downloading it from the parent project, before registering at the child project. Additionally, the following requirements have to be met for the application deployment:

- The registration method should be consistent with the original registration method, allowing already deployed projects to be added to a hierarchy without any modification and any project to leave the hierarchy anytime.

- Different versions of the same application should be allowed to run in parallel, since each parent may run different version of the same application.

- Since each application instance is tied to a platform, the application name should be the same for all platforms, allowing any child to query for the different platform instances of the application.

- Instances of the same application originating from different parents should be treated as different ones, to ensure that results are reported to the appropriate parent.

The flow of the deployment is the following.
1. The Gateway periodically queries higher level projects for new applications. When a new application is available it receives the \( (\text{AppName}, \text{Version}, \text{Signatures}) \) tuple identifying the application for a given Platform.

2. The Signatures are checked against the \( \text{TrustedID}_{\text{Project}} \) set of the child project containing all accepted \( \text{AppDev} \) and \( \text{Project} \) certificates.

3. The \( (\text{AppName}, \text{Version}, \text{Signatures}) \) triplet is checked against the list of applications already registered for a specific parent.
   a. If found, the application is already available at the child project.
   b. If not found, the Gateway creates a new mapping:
      \( (\text{AppName}, \text{Version}, \text{Signatures}, \text{Parent}) \rightarrow (\text{UUID}, 1.0) \)

4. The Gateway registers the application with BOINC using UUID as the application name and 1.0 as application version.

The above procedure ensure that applications can still be installed manually as in a regular BOINC project and that will not cause inconsistency between the configuration files of the project, the database of the project and the Gateway. There is one significant difference though: an automatically deployed application is not signed using the code-signing key of BOINC, instead the signature retrieved by the Gateway is used. This requires that the Core Client requesting work (and receiving applications) is able to retrieve the certificates (depending on the trust scenario described in 3.1.2.1.1) from the given project, and is able to validate the signature of the application (and the ones of the workunits belonging to it) using the certificates. The workload transfer method is detailed in Appendix A.2.

### 3.1.2.1.3 Remarks

BOINC uses majority based voting (i.e., redundancy) for job certification (see Figure 2.3 in Section 2.3). Redundancy aims at increasing the probability that every job will have a correct result by simply sending the same piece of work to multiple clients and comparing the results to filter out corrupt ones. In case of Scenario 1 (see Section 3.1.1.1) this may result in exponential growth of the number of redundant workunits. Let’s assume redundancy is three at each level thus, each level creates three copies of any workunit received. By the second level there will be nine redundant ones. This means that nine clients will compute the same workunit instead of the supposed three (which was the requested redundancy on the first level). If more levels are added to the hierarchy this number will exponentially grow. This problem can be solved by forcing redundancy to be disabled on all but the top level.

### 3.2 Towards volunteer clouds

Virtualization is the mechanism when a logical representation is created on top of the real, physical software or hardware component(s). Virtualization, for example, enables de-
coupling software services and their resources, i.e., separating the actual resources (CPU, storage, and network) from the physical hardware. Virtualization provides more flexibility for maintenance and improves the utilization rate of the physical resources. Cloud Computing builds on the achievements of diverse research areas, such as Grid Computing, Service-oriented computing, business processes and virtualization.

Volunteer computing systems collect heterogeneous resources with regard to capacity, configuration and volatility. Additionally each system has its own idiosyncrasies and requirements for operation and architecture of applications. Contrary to this cloud computing promises homogeneous resources and environments: homogeneous resources means processor architecture, available memory and disk. A Infrastructure as a Service (IaaS) cloud a client can request the required amount of hardware resources and the required number of (compute) instances.

For this theorem I define different abstraction levels for volunteer computing systems: (i) middleware abstraction satisfies dependencies between a given system and application; (ii) cross-middleware abstraction hides dependencies between systems; (iii) environment abstraction provides an uniform environment over the heterogeneous software environment of volunteer resources; and (iv) full abstraction provides an homogeneous environment over different operating systems and partially over heterogeneous hardware resources. Based on these I present the DC-API, GenWrapper and GBAC environments.

The GBAC environment I developed – that builds on the defined abstraction levels (see i-iv) – provides a homogeneous software and hardware environment for executing tasks on volunteer resources. It hides the requirement-differences of different desktop grid and volunteer computing systems; and satisfies the system-requirements that are present between applications and VC systems:

**Theorem II.2.:** The GBAC environment provides a homogeneous software and hardware environment for executing work units on resources of volunteer computing systems, and provides the fundamentals for the virtualized resource handling of a new computing infrastructure called Volunteer Computing Clouds.

Related publications: [91], [36], [21], [82] [27], [24], [29], [92], [93], [28], [25], [37], [26], [35], [33], [23], [32], [34], [30], [94], [95] and [22].

### 3.2.1 Introduction

There are several works (trying) to define cloud computing (e.g., [9, 96, 97]). The National Institute of Standards and Technology (NIST) provides the following definition for cloud computing [10] that became widely adopted:

**Definition 19** Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released
with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.

**Essential characteristics**  First (i) Any consumer should be able to provision the computing capabilities (e.g., storage or server time) she needs without human interaction with each provider (*on-demand self-service*); (ii) the capabilities are available over the network through standardized mechanisms (*broad network access*); (iii) provider’s resources are pooled to serve multiple consumers in a multi-tenant model (*resource pooling*). (iv) Capabilities can be provisioned and freed in an elastically fashion to scale up or down as demand requires. To the consumer capabilities often appear as unlimited (*rapid elasticity*) and (v) resource usage can be controlled, monitored and reported providing transparency both for the consumer and provider (*measured service*).

**Service models** The three service models are depicted in Figure 3.6 and are as follows:

1. **Software-as-a-Service (SaaS)** provides the capability to use a specific application running on a cloud infrastructure. The application is accessible either through an application programming interface (API) or a thin client (e.g., web browser). The consumer does not control or manage the underlaying infrastructure. Such services are e.g., Google Apps² or Microsoft Office 365³.

2. **Platform-as-a-Service (PaaS)** allows for the consumer to deploy applications on the cloud infrastructure that were created using programming tools (e.g., models, libraries, services) supported or provided by the provider. The consumer does not manage the underlaying infrastructure. Examples for such services are AWS Elastic Beanstalk⁴, Windows Azure⁵, Heroku⁶ or Google App Engine⁷.

3. **Infrastructure-as-a-Service (IaaS)** provides the capability to the consumer to provision fundamental computing (processing, storage or network) resources where arbitrary applications and software can be deployed. The consumer does not control or manage the underlaying cloud infrastructure, but has full control over the operating system and the deployed applications. Such services are, e.g., AWS EC2⁸, AWS S3⁹, Windows Azure (has both IaaS and PaaS offerings) and Rackspace Public Cloud¹⁰.

**Deployment models** The (a) *private* deployment model provides an infrastructure to be used only by consumers from a single organization; (b) a *community* cloud is provisioned

²http://www.google.com/enterprise/apps/business/, accessed on 2013-09-15
⁵https://www.windowsazure.com/, accessed on 2013-09-15
⁶https://www.heroku.com/, accessed on 2013-09-15
⁷https://developers.google.com/appengine/, accessed on 2013-09-15
¹⁰http://www.rackspace.com/cloud/, accessed 2013-09-15
3.2. Towards volunteer clouds

![Cloud computing service models and their relationship.](image)

Figure 3.6: Cloud computing service models and their relationship.

for the specific needs of a community of consumers (of different organizations) that have some shared concern; (c) a public cloud is provided for the open public and finally (d) a hybrid cloud is a composition of two or more distinct infrastructures (public, private or community). These remain separate entities, but are glued together by some technology that enables data and application portability.

### 3.2.2 Definition for volunteer cloud

Volunteer Cloud Computing (VCC) is the term used in this thesis for a cloud infrastructure based on volunteer resources. Considering the special characteristics of VC it is most likely that a definition will contain a subset of the service and deployment models described by the NIST definition (see Definition 19), but should fulfill all essential characteristics. To achieve a definition first the relevant characteristics of VC must be considered. In a volatile and unreliable environment such as volunteer computing establishing a cloud(-like) infrastructure faces several challenges.

#### Service models

Based on these considerations the following scenarios are proposed for volunteer clouds for the different service models:

1. **IaaS** Based on assumptions 2 and 4 made for VC in Section 2.3 direct access (e.g., deploying VMs and accessing them) is not feasible. Proposed scenario: Volunteer data archive. There are several works that investigate the storage/ archiving potential of volunteer computing [57, 98, 99].

2. **PaaS** Proposed scenario: Scientific computation platform. Arbitrary applications conforming to the requirements set by the platform can be executed. Methods and mechanisms are required for deploying the application. Science gateways can provide the thin-client interface while methods presented in Section 3.1.2.1 can be used for example in case of BOINC to deploy the application. Other possibility is to not to deploy the application rather execute them in containers.

3. **SaaS** Proposed scenario: Scientific application execution service. By the provider deployed applications can be executed. These applications are made available through standardized interfaces and/or thin clients by the providers.
Deployment models All four deployment models (see Definition 19) should be supported. The private deployment model (it does not mean here desktop grids a.k.a. private desktop grids) means that a single organization is able to run the scenarios defined above. The community deployment model denotes that a community can use the cloud while the public denotes that anyone could use the volunteer cloud. Hybrid model can utilize the federation approach described in Section 3.1.

Essential characteristics Characteristics iii (resource pooling) and v (measured service) are also essential characteristics for any grid, in fact of any distributed computing service. Characteristic i is valid with the restriction discussed for the IaaS service model as direct access to VMs is not feasible. Broad network access (see ii) via standardized interfaces is provided either by science gateways [100] or multi-access services like 3G Bridge [20], see Section 3.1.1 for more details. Finally characteristic iv (rapid elasticity) can be adopted only in a limited manner as allocating additional volunteer resources to the pool is not so simple as buying new hardware. Since volunteer resources are owned by individuals (donors) thus, allocating additional resources explicitly means that new donors should donate resources. Also existing resources are volatile thus, the elasticity, capacity and performance of the resource pool continuously fluctuates. However in Section 3.3.2 a method is presented to augment volunteer (cloud) computing projects with dedicated resources.

Definition Based on these findings a very similar definition can be given for volunteer cloud computing as for cloud computing (see Definition 19) in general:

Definition 20 Volunteer Cloud Computing is a model for enabling network access in a ubiquitous, convenient, on-demand manner to a shared pool of volunteer and/or non-dedicated compute and storage resources that can be rapidly provisioned and released with minimal management effort or service provider interaction. This model incorporates the five essential characteristics, three service models and four deployment models introduced in Definition 19 in a restricted manner under the assumptions given in Section 3.2.2.

3.2.3 Abstraction frameworks for volunteer computing
Volunteer Computing relies the donated resources of its donors. The donor always retains full control of her computer while it is performing a task on behalf of a VC project (see assumption 2 in Section 2.3). The VC middleware assists by providing additional features that makes the VC 'experience' as seamless as possible for the donor, e.g., screensavers, check-pointing support or dynamic process priority during execution. These require specific modifications to the source code of the application and/or linking with specific libraries that the middleware provides like in the case of BOINC. This true not just for VC, but for distributed computing in general as well (e.g., the standard universe of HTCondor). Additionally VC utilizes a diverse and heterogeneous set of donated (computing) resources consisting of different CPUs and architectures, operating systems, installed libraries and tools that the applications must be prepared to handle. However modifying source code
3.2. Towards volunteer clouds

and relinking applications are not always possible or feasible. Such example is the ISDEP application [101] (see ‘Fusion/ISDEP’ in Table A.1) with around 10,000 lines of C source code that is evolving constantly. Modifying it to meet the requirements of a single middleware requires effort, but to apply the modifications for each new version is not feasible. Also there are applications which have no source code available or rely on external libraries or using environments like R [102]. Such application is the ‘Patient-Readmission’ (see Table A.2) application. These would require that all VC donors deploy R on their donated resources and that the VC middleware supports R, which is not feasible at all for a single scientific application.

In this section I am going to discuss abstraction frameworks (see Figure 3.8 for an overview) that allow the rapid and simplified development of new; and the adaptation (also known as ‘porting’) of existing scientific applications to VC environments. These abstraction methods are essential for the discussed service models and characteristics of a Volunteer Cloud Computing system.

3.2.3.1 Related work

A taxonomy of methods for developing and porting (adapting) applications for different distributed computing systems is shown in Figure 3.7.

There are efforts like the GAT [103], SAGA [104] or DRMAA [105] for creating a unified API for applications on distributed systems. However, these are usually modeled after traditional grid middlewares and batch schedulers but are not aimed at volunteer computing systems such as BOINC [57]. The above mentioned APIs are overly complex for such a restricted environment that BOINC provides and they also fail to cover areas like logical file name resolution, checkpoint control, redundant execution and result validation that are required in a BOINC environment. Also the volatility of desktop grid environments where clients may come and go at any time, there is no guarantee that a client that started a computation will indeed finish it, presents a problem for interface designs based on traditional job submission principles.

Ferreira et al. [106] designed a common set of VM manipulation functions to hide the hypervisor specific details. Five methods have been defined by the API covering information query, VM startup, file copy, checkpoint and finally command execution inside the virtual machine. In this experience they have implemented the API for local execution, VMWare and VirtualBox. Finally, they have integrated this library as an application into the BOINC wrapper and performed tests to measure the overhead. Its most powerful feature is the easy switching between different virtualization environments however, this work is somehow a duplication of the virtualization API and its libraries provided by libVirt.

The BOINC VBoxWrapper is a virtualized environment providing wrapper for BOINC created by the developers of BOINC. It interfaces between the BOINC client and VirtualBox. It is a BOINC specific wrapper as it relies on middleware specific functionalities (e.g., file transfer between the host and the shared directory, application execution) and does not aim at being a generic one.

CernVM [107] is also based on the BOINC VBoxWrapper tool however, the main difference is in the virtual appliance and work distribution. Usually, in a BOINC environment
the downloaded work unit (including the image as well) contains inputs and results are uploaded to the BOINC server when the job finished. The CernVM solution follows a new approach where the virtual appliance contains a complete job scheduler that performs job fetching from an external server by itself. So, this solution does not utilize BOINC work unit scheduling and the application validation framework rather uses the BOINC server only for distributing the CernVM images.

3.2.3.2 Abstraction A1: Cross-middleware abstraction

Users of scientific applications are usually concerned only about the amount of computing power they can get and not about the details how a distributed computing system (DCS) provides this computing power. Therefore, they want to develop a single application that in turn can run on any infrastructure that provides the most appropriate resources at a given time. Unfortunately existing applications have to be modified in order to run on different distributed systems and this makes volunteer computing less attractive for application developers than traditional distributed systems. As Figure 3.8 shows Abstraction
### Towards volunteer clouds

<table>
<thead>
<tr>
<th></th>
<th>A1</th>
<th>A2</th>
<th>A3-1</th>
<th>A3-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. BOINC API</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>2. DC-API</td>
<td>√</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>3. BOINC Wrapper</td>
<td>×</td>
<td>√</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>4. GenWrapper</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>×</td>
</tr>
<tr>
<td>5. GBAC</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

Table 3.1: Summary of frameworks and their supported abstractions

A1 hides middleware specifics so that a single interface provides access to different ones in an uniform manner.

### 3.2.3.2.1 DC-API: Distributed Computing API

DC-API is a simple API that is specifically targeted for VCSs. Its goal is to provide an API that requires only minimal modification to existing application source code. However, the DC-API is opaque in the sense that it can be implemented for traditional grid and other DCSs as well therefore, applications using the DC-API could be easily deployed on other distributed infrastructures as well, without the need to modify the source code of the application. An example use case can be found in Appendix A.6 and as it is shown in Table 3.1 it implements Abstraction A1.

VC provides a restricted environment and programming model compared to traditional distributed computing systems. This means that some application categories like generic parallel applications cannot be supported. On the other hand due to this restricted programming model the DC-API can be small and simple. Although the original motivation for creating the DC-API was to help porting existing applications, DC-API also provides extra features on top of the functionality provided by BOINC. Among these features are the proper support of client applications that create multiple output files (the BOINC API only provides support for accessing the first output file), or the suspension of workunits by transferring a checkpoint file back to the master when it can be submitted and continued on another resource. DC-API also makes it easier to run multiple independent applications under the same BOINC project.

DC-API backends exist to use the Condor job manager, BOINC and XtremWeb as well. A simple fork-based implementation that runs all workunits on the local host is also available. The ability of running the workunits locally makes application debugging easier. Since switching the application from using such a local implementation to e.g., BOINC needs only a recompilation without any changes to the source code, the complete application can be tested on the developer’s machine before deploying it to a complex grid infrastructure.

To accommodate the restrictions of different grid environments and to facilitate converting existing sequential code written by scientists not comfortable with parallel programming, the DC-API supports a limited parallel programming model only. This implies the following restrictions compared to general parallel programming:

- Master/Worker concept: there is a designated master process running somewhere on the grid infrastructure. The master process can submit worker processes called
- Every workunit is a sequential application.

- There is support for limited messaging between the master and the running workunits. However, this is not suitable for parallel programming, it is meant to be used for sending status and control messages only.

- No direct communication between workunits.

Following the Master/Worker model, DC-API applications consist of two major components (see Figure 3.9): a master and one or more client applications. The master is responsible for dividing the global input data into smaller chunks and distributing them in the form of workunits. Interpreting the output generated by the workunits and combining them to a global output is also the job of the master. The master usually runs as a daemon, but it is also possible to write it so it runs periodically (e.g. from cron), processes the outstanding events, and exits. Client applications are simple sequential programs that take their input, perform some computation on it and produce some output.

A typical master application written using DC-API does the following steps:

1. Initialises the DC-API master library by calling the DC_initMaster function.

2. Calls the DC_setResultCB function and optionally some of the DC_setSubresultCb, DC_setMessageCb, DC_setSuspendCb and DC_setValidateCb functions, depending on the advanced features (messaging, subresults, etc.) it wants to use.

3. In its main loop, calls the DC_createWU function to create new workunits when needed and after specifying the necessary input and output files (DC_addWUInput, DC_addWUOutput) it can hand them over to the grid infrastructure for processing by calling the DC_submitWU function. If the total number of workunits is small (depending on the grid infrastructure), then the master may also create all the workunits in advance. If the number of workunits is large, the master may use the DC_getWUNumber function to determine the current number of workunits processed by the grid infrastructure, and create new workunits only if it falls below a certain threshold.
4. Also in its main loop the master calls the \texttt{DC\_process\_Master\_Events} function that checks for outstanding events and invokes the appropriate callbacks. Alternatively, the master may use the \texttt{DC\_wait\_Master\_Event} and \texttt{DC\_wait\_WU\_Event} functions instead of \texttt{DC\_process\_Master\_Events} if it prefers to receive event structures instead of using callbacks.

A typical client application performs the following steps:

1. Initializes the DC-API client library by calling the \texttt{DC\_init\_Client} function.

2. Identifies the location of its input/output files by calling the \texttt{DC\_resolve\_FileName} function. Note that the client application may not assume that it can read/ create/ write any files other than the names returned by \texttt{DC\_resolve\_FileName}.

3. During the computation, the client should periodically call the \texttt{DC\_check\_Client\_Event} function and process the received events.

4. If possible, the client should call the \texttt{DC\_fraction\_Done} function with the fraction of the work completed. On some grid infrastructures (e.g. BOINC) this will allow the client’s supervisor process to show the progress of the application to the user. Ideally the value passed to this function should be proportional to the time elapsed so far compared to the total time that will be needed to complete the computation.

5. The client should call the \texttt{DC\_finish\_Client} function at the end of the computation. As a result, all output files will be sent to the master and the master will be notified about the completion of the work unit.

### 3.2.3.3 Abstraction A2 and A3-1: Middleware and environment abstraction

Any application to be deployed on a VCS typically requires modification. For example BOINC infrastructure needs source code modifications and re-linking. However, there are many so called legacy applications, that have either no source code available to modify or simply would require to much effort to port. Here an abstraction is required that acts as a middleman or translator between the legacy application and the middleware so the two can interact with each other. For this purpose is \textit{Abstraction A2: Middleware detachment} is introduced (see Figure 3.8). For example for BOINC provides a wrapper (see Table 3.1) for \textit{Abstraction A2}. This wrapper can be used to handle the communication with the BOINC Core Client, while executing the legacy application as a subprocess.

Additionally applications may require external tools and libraries to run (see ‘Patient-Readmission’ in Table A.2 and Section 3.2.3). In turn this would require that all donated resources have these tools and libraries deployed for the application, which is impossible. For this problem is \textit{Abstraction A3-1: Environment detachment} is introduced. The purpose of this abstraction is to provide the required external dependencies for the applications in the execution environment (see Figure 3.8).
3.2.3.3.1 GenWrapper: A generic wrapper for volunteer computing

GenWrapper aims to provide a generic solution for wrapping and executing an arbitrary set of legacy applications by utilizing a POSIX like shell scripting environment to describe how the application is to be run and how the work unit should be processed. This choice provides great flexibility and a powerful tool to adapt legacy applications to VC with very little effort. It implements Abstractions A1, A2 and A3-1 (see Table 3.1), thus it is middleware independent, supports legacy applications and ensures that the required dependencies and libraries are deployed at the execution environment.

Applications under BOINC  To use the distributed resources gathered by BOINC, the application performing the computation also needs some preparation to be able run on the client machines under the control of the Core Client. Apart from having executables for all possible platforms that are member of the DG, the application also has to be prepared to be run by the BOINC Core Client which has two main aspects: i) it should be able to run in the directory structure used by the client, i.e. application executables are placed in the project directory while the working directory is a separate slot directory where input and output files are linked; ii) it should be able to interact with the Core Client, i.e. handle suspend, resume and quit requests and report used CPU time and checkpoints.

BOINC provides an API that applications should use to communicate with the Core Client and handle running in the DG environment. This API provides functions for resolving links to files that are accessed from the slot directory, communicate exit status to the Core Client so it can handle errors and report statistics as needed.

![Figure 3.10: Legacy application using (a) the BOINC wrapper; and (b) GenWrapper](image)

Legacy applications or applications which cannot be modified to use the API are not able to run under BOINC because without calling the right API functions they would find links instead of their input files, write their outputs to the wrong place and without properly reporting statistics to the Core Client the application would be restarted over
and over and eventually it would be marked as failed. For these applications BOINC offers the BOINC Wrapper (see Figure 3.10/a) which acts as a main program managing communication with the Core Client calling the appropriate API functions and running the real application executable as a subprocess. An application using the BOINC Wrapper contains the wrapper executable besides the application files.

**Applications with GenWrapper** A typical GenWrapper application consists of a zip file holding all the files belonging to legacy application(s), the two GenWrapper components: GitBox and Launcher executables and a *profile script* to perform platform specific preparations. A typical work unit for a GenWrapper wrapped application contains the input files and another shell script (the *work unit shell script*), which allows to control and execute the legacy applications in an arbitrary manner for each work unit. The work unit shell script should be platform independent as the work unit can be executed by any supported platform.

GenWrapper consists of an extended version of BusyBox\[108\], a single binary providing essential UNIX commands (such as *sed*, *grep*, *unzip*, *tar*, *awk*, etc.) and a POSIX shell interpreter (based on *ash*). GenWrapper is ported to run on Windows, Linux and Mac OS X platforms and it is extended to make BOINC API functions accessible from the shell (e.g. *boinc resolve_filename* or *boinc fraction_done*). GitBox is a stripped-down Windows only port of BusyBox originally created for the Windows version of the git version control system which internally relies on running shell scripts. Although GIT on Windows later abandoned GitBox, this port was used as the basis for GenWrapper after extracting it from GIT and porting back to UNIX preserving functionality on Windows. Later the GenWrapper GitBox version was updated to match newer BusyBox releases, and thus diverged from the original GitBox significantly, although it is still referred to as GitBox in the GenWrapper distribution for historical reasons. Besides Windows the GenWrapper GitBox (which we will simply call GitBox now) is also supported on Linux and Mac OS X, some of the stripped down parts were put back and new BusyBox functionality (e.g. lzma compression) were added and it was extended with BOINC specific shell commands.

A GenWrapper wrapper legacy application is executed as follows (see Figure 3.10/b). The client downloads the Launcher executable (named like the application as BOINC expects), an application zip file and an optional profile script as the BOINC application and a work unit (input files and a work unit shell script). The Launcher is started by BOINC and acts as a BOINC application, handling all communication with the Core Client. After starting, the Launcher looks for a .zip file with the same name as itself and extracts all files from it to the slot directory. Storing application files in a .zip file is optional, if not found no extracting is performed and the work unit script should access it by resolving its logical name as any other input files. It is recommended to use application zip files, because the BOINC server stores all application files in one common location on the project web server and when files with the same name, but different content are required by different applications a conflict may happen which is prevented by storing application specific files in a zip archive. The most obvious scenario would be that different applications require different versions of the same DLL (Windows shared library) files. These files
commonly have the same 8 character filename (plus the ",dll" extension) regardless of their version. Without packaging application files together in a zip, a later deployed application could overwrite the same named files belonging to a previously installed application. The application zip file may also contain a "profile script" which serves as a platform specific bootstrap script for the application (e.g. on Linux the library include path may need to be adjusted or local optimization options could be enabled depending on the presence of optional features, etc.). After unzipping the application archive, the Launcher generates a starter script which first sources the profile script if exists and then executes the work unit shell script. Then Launcher calls the built in POSIX shell interpreter (ash) of GitBox which starts to execute this generated script.

The Launcher remains running while GitBox executes the script and handles communication with the Core Client and performs similar tasks as the BOINC Wrapper. In fact Launcher was originally based on BOINC Wrapper, but it is heavily modified to fit the needs of GenWrapper. Modifications include: i) suspending and resuming GitBox and all the subprocesses started by it when the Core Client asks for this; ii) measuring and reporting the CPU time used by the running subprocesses and iii) killing the subprocesses if the requested or the client is stopped. The Launcher is spawning a new process for GitBox which is also spawning a new process for each legacy application it is executing. If the functionality of the original BOINC Wrapper were used here, only the GitBox process could be controlled and measured, while losing control over the legacy application processes (which do the actual work) and the Core Client having no information about them. These tasks are implemented differently on UNIX and Windows systems due to the lack of common API concepts and Windows’ limited support of POSIX.

On Linux and Mac OS X there are process groups that the Launcher utilizes by simply putting the spawned GitBox process in a new process group and by default all its child processes will also belong to the same group. The limitation here is that no child process should break away from the process group, thus currently no subshells are supported (scripts should not create background processes or in practice & and parentheses should be avoided) and also the legacy applications should avoid creating new process groups (although they can spawn subprocesses which are not breaking away from the process group).

On Windows, there are no process groups (a feature with the same name exists, but it is only vaguely similar to UNIX process groups and cannot be used the same way). The closest feature the WIN32 API provides is called JobObject. Each JobObject represents a collection of processes. But the problem with them is twofold: i) by default a process started by a process in a JobObject (child) should also belong to the same JobObject as its parent, but unfortunately it depends which system function was used to create the child process (CreateProcess() is fine, but _spawn() is not always), thus not every child process may end up in the JobObject; ii) there is no official, documented API function to suspend or resume a process, only threads can be controlled; if a process has more than one thread suspending them in the wrong order might lead to a dead-lock. This last problem is solved by using undocumented Windows NTAPI calls that can directly suspend and resume processes. The first problem was overcome by periodically checking the list of running processes whether there is a new one whose parent process belongs to the JobObject but it isn’t. If such processes are found they are added to the JobObject.
Towards volunteer clouds

There is no function to suspend or resume all processes in a JobObject, but it is possible to terminate all processes of it. Thus, if suspend or resume is requested the JobObject is queried for the list of its processes and each one of those is handled one by one.

GitBox (as well as BusyBox) has a modular structure, which allows to easily extend it with arbitrary commands by so called “applets”. The BOINC extension is implemented in such an applet and currently consists of the most important BOINC API calls such as, resolve_filename, fraction_done, fraction_done_percent. A minimalist sample work unit script for demonstrating the basic capabilities can be seen in Figure A.4. This sample is reading an (integer) value from the file with the logical filename ‘in’ (by first resolving the link to the real file), performing a loop which: i) calculates how much of the total work is done; ii) print the fraction done in percent into the file with logical name ‘out’ and iii) sleep for a second in each iteration. There is no need to provide or call boinc_init() or boinc_finish() from the script itself, because it is called by the Launcher, which also takes care of forwarding the exit status to boinc_finish (thus the script can normally exit with a non-zero status to signal an error). The Launcher also measures used CPU time and reports to the Core Client automatically. The only BOINC API functionality required for this simple example is to resolve logical filenames and report the fraction done which is also the case for the majority of legacy applications.

GenWrapper utilizes Abstraction A1 (relies on DC-API) and further extends it (Abstraction A2) by removing the need to modify application binaries. It also provides support for custom application environments (Abstraction A3-1). GenWrapper provides a robust and mature method to adapt existing scientific applications for volunteer computing. A complete list of known applications ported with GenWrapper can be found in Appendix A.1, and sample GenWrapper scripts are listed in Appendix A.5.

Abstraction A3-2: Full abstraction

VC utilizes donated, usually desktop, resources where the majority is running Microsoft Windows, while scientific applications usually prefer Linux. For SZTAKI Desktop Grid (SZDG) over 70% of resources is running some variant of Windows. This means in order to utilize all resources an application on SZDG is required to have a Linux and a Windows version. Abstraction A3-1 provides a method for (legacy) applications to have their external dependencies handled. However this does not take into account OS level dependencies. For such problems Abstraction A3-2: Full detachment is introduced. This incorporates Abstraction A3-1, but also provides independence from the OS, and provides a single uniform execution environment. It is shown in Figure 3.8.

In the next sub-sections (Sections 3.2.3.4.1 and 3.2.3.4.2) two approaches are detailed for Abstraction A3-2 but also utilize Abstraction A1 and A2 (see GBAC in Table 3.1). The first is a middleware based solution, it is tailored for a specific VC middleware, while the second is a standalone cross-middleware application itself. Both are based on sandboxes and virtualization.

In general sandboxes provide an isolated execution environment for applications. Unix-like systems have traditionally supported this by techniques and tools like chroot or BSD jail. The problem with these solutions is that they are only available for Linux/BSD,
while VC is aimed at the general public thus, it is not enough to provide a solution for a single platform. With the emergence of hypervisors for the desktop computer (like VMware Player/ Fusion, VirtualBox, Bochs or QEMU) which are cross-platform this obstacle can be easily mitigated.

However it is not a solution to simply take and use any of these existing tools, since donors cannot be expected to be Grid or operating system administrators thus to have the knowledge (or be willing) to deploy (in Table 3.2) any “complicated” piece of software. For the long term if the donor notices any slowdown or the computer becomes less responsive while the sandbox is run the VC client will be removed (“slowdown” in Table 3.2). So in the first place a transparent sandbox should comply with the philosophy of VC and should not be noticeable for the donor transparent for the donor.

There are many already deployed VC projects. If applying a sandbox would require either to perform modifications to on the server deployed applications, for example they needed to be moved in a disk image no administrator can be expected to do this. Also there should be no restrictions for the application whether it is run in a sandbox or just simply executed by the client meaning the same capabilities should be available that are normally provided by the client, namely: checkpoint and resume, suspend and continue, start and stop and to report data about the running application (percent complete, resources used, etc.). These facilities should be available not just from a graphical interface, but also from an API-like one so the sandbox can be “remote controlled”. So in the second place transparent sandbox should mean transparent for the system.

A sandbox should isolate the running application inside meaning should have no access to the resources of the host and should prohibit any outside connection. It should be also “bulletproof” thus no malicious application may render the sandbox unusable or requiring administrator intervention. This may be impossible to fulfill for any currently running task since an application might destroy it, but at least for the next task the sandbox should be once again available (and the current one should be stopped and marked as failed). The transparency for the system criteria requires that the there is no restriction for the client whether running applications normally or inside a sandbox, thus it needs a “backdoor” to copy applications and input data into the sandbox and later to extract results from it to be uploaded to the server. It is not enough for this backdoor to provide a simple file put/get interface it also should enable to start or stop and to get status information about the work unit running. This backdoor should be one way, meaning the client can put and extract data from the sandbox, but the sandbox cannot reach the host via this interface.

Multi-core processors are common for desktop computers thus more instances can be executed parallel. In this case independent “copies” of the sandbox should be started for each application (running on different cores) meaning that the sandbox should provide instantiation. Worst case this means that a copy is made every time of an immutable sandbox (“base”) and the copy is started for the application. Best case no copy is need to be made, just a thin overlay which references the base and every modification to the instance should be stored there. This would also lower the used disk space and the time required for creating new instances. By having an immutable sandbox we ensure the “bulletproof” criteria, since the instance can be thrown away whether the work unit finished successfully or not and a new instance can be created for the next one.
3.2. Towards volunteer clouds

When using virtualization software licensing issues apply, meaning that the software should be preferably open source and freely redistributable at best, but worst case it must still allow the software to be deployed with the Desktop Grid client. Manual download and installation might be too much of a task for any user. It should be able to run in the background without any windows or pop-ups presented at the user (“background” in Table 3.2).

Any potentially to be used virtualization software must fulfill these requirements in order to be used as a transparent sandbox.

<table>
<thead>
<tr>
<th>Method (Emulation/Virtualization)</th>
<th>Bochs</th>
<th>QEMU</th>
<th>QEMU + KVM (KQEMU)</th>
<th>VMware Player</th>
<th>VirtualBox</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <strong>transparency for the donor</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a. deployment</td>
<td>√</td>
<td>√</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>b. slowdown</td>
<td>×</td>
<td>√</td>
<td>× (KQEMU)</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>2. <strong>transparency for the middleware</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a. checkpoint and resume</td>
<td>×</td>
<td>√</td>
<td>√</td>
<td>×</td>
<td>√</td>
</tr>
<tr>
<td>b. suspend and continue</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>c. remote control</td>
<td>×</td>
<td>√</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>d. backdoor</td>
<td>×</td>
<td>√</td>
<td>× (KQEMU)</td>
<td>×</td>
<td>√</td>
</tr>
<tr>
<td>3. <strong>isolation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. <strong>cross-platform</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. <strong>performance</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. <strong>instantiation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. <strong>background</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8. <strong>licensing</strong></td>
<td>LGPL</td>
<td>GPL</td>
<td>GPL</td>
<td>Proprietary</td>
<td>Proprietary/GPL</td>
</tr>
</tbody>
</table>

Table 3.2: Comparison of major hypervisors for desktops and emulation software

**Virtual Machines** Although we refer all tools described in this section as “virtual machines”, we must distinguish two approaches. Pure emulation models the desired architecture/platform from software. No code from the guest is ever run on the host cpu, everything is emulated. The biggest benefit is portability, since there is no dependency between the emulated platform and the hosting one, the trade-off is a significant performance loss, a typical example is Bochs [109]. The second approach is usually referred as “virtualization”: it is used to implement a virtual machine environment so that it provides simulation of the underlying hardware. There are several levels of virtualization which can be accomplished. Full virtualization provides a complete abstraction of the underlying hardware enabling execution of all software that runs on the raw hardware to be run in the virtual machine, examples for full virtualization are VMware and VirtualBox [110]. Hardware-assisted virtualization (or native virtualization) is a virtualization approach that enables efficient full virtualization using help from hardware capabilities like AMD-V (originally codenamed Pacifica and AMD Secure Virtual Machines) [111] or Intel VT-x [112], primarily from the host processors. Paravirtualization presents a software interface to virtual machines that is similar but not identical to that
of the underlying hardware. It may allow the virtual machines that run on it to achieve performance closer to non-virtualized hardware. However, operating systems must be explicitly ported. The best example for paravirtualization is XEN [113].

Bochs is an open source x86 emulator written in C++. It is running in user-space, and emulates the x86 processor with several I/O devices, and provides a custom BIOS. Bochs is highly portable, but rather slow since it emulates every instruction and I/O device. The primary author of Bochs reported 1.5 MIPS on a 400 MHz Pentium II, compared to the processor’s original speed of $\approx$ 1100 MIPS.

QEMU is an open source processor emulator and virtual machine, released under the Lesser GNU Public License and BSD License. It is available on every major platform and is able to host every major platform and can boot many guest operating systems. It runs as a single user process and is intended as a desktop product, but can be run as a background process presenting no windows or terminals at the user. Since it is emulating the guest architecture the performance is far from native, but on x86 Windows/Linux it has an accelerator named KQEMU or QEMU Accelerator, which speeds up x86 emulation to near native level. This is accomplished by running user mode code directly on the host computer’s CPU, and using processor and peripheral emulation only for kernel mode and real mode code. However KQEMU was deprecated in the favor of KVM [114]. QEMU implements overlay images, meaning it can keep a snapshot of the guest system, and write changes to a separate image file. If the guest system breaks, the original image or an earlier snapshot can be used to resume. It can save and restore the state of the running instance (checkpoint itself), also to an overlay image. QEMU does not need administrative rights to run. Implements copy-on-write disk image formats, meaning that images only use that much disk space what they actually use and supports compressing images. Has a console for managing running instances, and this console can be accessed either from an user interface or via a socket. It has support for virtual network card emulation, or can disable outside network at all, but still can forward a specific port to the host and bind it to a socket or port allowing one way communication with applications within the guest. It has a set of command line tools which provide full control over QEMU without having to run a separate graphical client.

VMware Player is a free, but not open source x86 virtualization product with the limitation of not being able to create, only able to start VM images. It provides the appearance of full virtualization by using binary translation. It requires Windows or Linux to run. It consists of several components: a user-level application (VMApp), a device driver (VMDriver) for the host system, and a virtual machine monitor (VMM). I/O initiated by a guest system is trapped the the VMM and forwarded to the VMApp, which executes in the host’s context and performs the I/O using regular system calls, this way it achieves nearly native performance. VMware Player has a graphical interface and needs administrative rights to run on Windows. VMware does not implement support for overlay images.

VirtualBox is an x86 only virtualization software with optional support for hardware-assisted virtualization (through AMD-V and Intel VT). Its main components are open source released under the GNU Public License V2. Some additional components (e.g., USB support) is not open source, but still free for personal use and evaluation. VirtualBox originally relied on some QEMU components, it tries to run as much code as possible
native, but if problems arise it falls back to a dynamic recompiler, which based on QEMU, to emulate the x86 processor. Actually VirtualBox makes use of QEMU in two ways: first, some of the virtual hardware devices have their origin in the QEMU project. Secondly it utilizes the recompiler of QEMU as a fallback mechanism for situations where it’s Virtual Machine Manager (monitor) cannot correctly handle a certain situation. VirtualBox needs administrative privileges to run.

By using some kind of virtualization, preferably virtual machines (VMs), as sandbox providing tool we would have several additional benefits:

- **Simplified application development.** Separate the host operating system from the guest, which allow to run the same os (preferably Linux) on the guest regardless of the host os. This would allow to have a single version of the application, but it would still run on all resources connected.

- **System-level checkpointing.** VMs usually provide a method to save or serialize their current state to disk which can be later used to resume the VM, thus running applications do not need to have the capability to checkpoint themselves. For example in the case of BOINC where each application implements its own checkpoint function (application-level checkpointing) this would mean that there is no need to write this non-trivial function. Combining system-level checkpointing with instantiation would allow easy migration of the task, if the same base image is used amongst clients, only the instance image needs to be migrated to a new client which would greatly reduce upload times and save bandwidth.

- **Legacy applications.** By having a separate operating system available inside the VM, applications could be run whose source code is not available, so cannot be ported to any new platform, or those that have too many dependencies to be run on a volunteer’s computer.

- **Enforce resource limits.** Any application running is guaranteed not to exceed the resource limits (e.g.: memory, disk or cpu) allocated for the VM.

### 3.2.3.4.1 Middleware based approach

The main goal was to design a generic architecture that can be later integrated with DGs (especially with BOINC and XtremWeb). The basic concept is to provide tools and APIs which allow to create and start VM instances, upload input files and executables into the instance, start tasks using the uploaded files, request status information and when finished retrieve the output files. The architecture is shown in Figure 3.11, the main components are the following:

- **The VM API** is to hide the under laying architecture and to provide a simple API for creating VM instances using the VM Manager and interacting with the created instances.

- **The Backend** stores meta data about the status of each created VM Instance. This allows the sandbox to be shut down and resumed any time.
- The **Base Image(s)** serve as immutable basis for the sandbox. Each instance is created from one of these base images. It contains a minimalist installation of a operating system (preferably Linux) and the Communication Daemon.

- The **Instance Image(s)** store the difference between the base image and the created instances. This way nothing is written back to the Base Image whether something is created, modified or removed. Also checkpoints should be stored here.

- The **Communication Daemon** handles the communication and data transfer between the VM Instance and the host. Communication is always initiated by the host, the daemon can only reply to requests, thus the host has to poll the daemon periodically for updates.

- The **Execution Environment** acts is where all application and task data is put and the task is run inside the VM Instance.

---

![Figure 3.11: Architecture of the sandbox](image)

The client downloads a new task which consists of a binary executable, input files for the binary and several other files (libraries and other dependencies). Normally these are to be run by the client using a `fork()` based execution method, but we provide a *sandbox* based one. A separate sandbox for each task is created with the capability to suspend, continue, checkpoint and resume if requested by the client. For managing sandboxes a C API is provided and a JAVA API is being developed, now we detail the C API to

demonstrate how the sandbox works. Sandboxes consist of a VM instance and a library providing the VM API to manage the instance. The VM API consists of two parts, first the `vm_sb_*` functions are to control tasks inside an instance, the remaining `vm_*` functions are to manage the instance(s). The functionality of this API is similar to the one provided by libvirt, but for our approach we did not need the complexity and most of the features of libvirt, and so we choose to go with an own simpler one.

The `vm_init` and `vm_cleanup` functions are called by the client on start up and before exit. All `vm_*` functions operate on a VM STATE structure which represents the VM Instance. The state and meta data of the instance is always kept updated in the Backend, thus no need to serialize this structure manually. When needed any previous (not deleted) instance can be loaded using `vm_load`.

First the sandbox is created by calling the `vm_create` function, which creates a new VM instance using a selected Base Image as basis. `vm_start` starts the instance, the instance might not be available instantly, so the client waits and issues `vm_get_state` periodically. When the instance is running the files of the current task are copied in to the sandbox inside the instance using the `vm_sb_put` function. Any disk operation is written to the instance image not the base image, this way even if a malicious application destroys somehow the VM Instance, it cannot make any damage to the Base Image or to any other existing instances.

Inside the VM Instance the Communication Daemon handles the incoming messages and passes them either to the Message Handler which is responsible for controlling the application or to the Data Handler which is responsible for moving data in or out from the instance. `vm_sb_start` is used to start the execution of the application, any additional parameters (e.g.: environment variables, command line, etc) are passed by this function. Since the sandbox cannot initiate outward communication the client needs to periodically poll the status of the running application using `vm_sb_ping`. After the application finished (whether successfully or failed) the output files are copied back to the client using `vm_sb_get`, the sandbox will report the client that the application exited and the results can be uploaded to the server. After finished the VM Instance is not needed anymore, so first it is shutdown by issuing `vm_stop` and second it is deleted by calling `vm_destroy`.

The client may be instructed to suspend computation for example by local policies or user request. In this case the `vm_suspend` function suspends the VM Instance, but it still remains resident in memory waiting for `vm_continue` to continue operation. If the client is shutdown all running instances need to checkpoint, this is done via `vm_checkpoint`. Each checkpoint is assigned an unique id which can be used later to resume using `vm_resume`, or from the last checkpoint if the uuid is omitted.

There are messaging functions provided for sending custom messages from/to the instance., which can be used also to upload partial results to the host while the task is active. `msg_from_host` function is for sending custom messages to the task running in the VM Instance, while the `msg_to_host` is used to send a message from the task to the host. Since the VM Instance cannot initiate any communication outwards, the host needs to periodically call this function to check if there are any messages from the guest. A type is assigned for every task which identifies the DG the task is originating from. Currently BOINC and XTREMWEB types are supported (and UNKNOWN type is set as default).
The Communication Daemon allows to install custom message/data handlers for the different task types, this allows to easily extend and customize it according the needs of the selected DG.

**Performance aspects** To test the performance of our prototype we took a real-world application, run by a BOINC based VC at SZTAKI, namely SZTAKI Desktop Grid (SZDG) [77]. The original intention of SZDG was to serve demonstration purposes among scientists mainly in Hungary and also worldwide to prove that it’s not necessary to have expensive hardware and truly monumental aims to catch the attention of the voluntary public. The Computer Algebra Department of the Eötvös Loránd University applied for the project with their already developed and running single-threaded program: project BinSYS [115, 32]. It was ported to run on SZDG with DC-API. The goal of BinSYS was to determine all of the binary number systems up to the dimension of 11. The difficulty in this is that the number of possible number-system bases explodes with the rising of the dimension. The input of the program is a huge, but finite part of the number space and the output is a bunch of matrices, or more precisely their characteristic polynomials fulfilling certain criteria. Further narrowing the criteria on these selected matrices, the resulting ones make up the generalized binary number systems of the given dimension. Knowing the complete list of these number systems the next step is to further analyze from the view of information theory. Sketching the integer vector of the vector space in the usual way and in the generalized number system, their form can greatly vary in length. Also the binary form of vectors close to each other can vary on a broad scale. With this in mind the research will continue further with the use of number systems in data compression and cryptography. The assumption was that the program will be able to handle the dimensions up to 11 on a few computers and by the time the cooperation has been started it has already finished up to dimension 9. The prediction has assumed that the processing of dimension 10 will last for about a year, yet it has been successfully finished by the end of the year 2005 with the help of the few thousand computers of volunteers joining the project. After this success the application has been further developed making it able to handle dimensions higher than 11 and also to break the barriers of the binary world and process number systems with a base higher than 2.

BinSYS is a typical “embarrassingly parallel” application, meaning the size of the executable and input files for tasks are minimal, while it has a rather long run time varying for each task. When using virtualization we have to take in account the overhead generated by the virtualization: (i) the memory needed by the guest operating system; (ii) the memory needed by the virtual machine itself; (iii) the CPU time overhead by

<table>
<thead>
<tr>
<th>Type</th>
<th>Slowest run</th>
<th>Fastest run</th>
<th>Mean time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native Linux</td>
<td>711.06 sec</td>
<td>708.17 sec</td>
<td>710.20 sec</td>
</tr>
<tr>
<td>Windows host, Linux Guest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sandbox Normal Priority, -kernel-kqemu</td>
<td>747.56 sec</td>
<td>744.21 sec</td>
<td>745.12 sec</td>
</tr>
<tr>
<td>Windows host, Linux Guest</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sandbox Below Normal Priority, -kernel-kqemu</td>
<td>759.76 sec</td>
<td>757.60 sec</td>
<td>758.71 sec</td>
</tr>
</tbody>
</table>

Figure 3.12: Run-time (real, in seconds) of the test work unit
the virtual machine; (iv) the disk space used by the virtual machine image(s); (v) the additional disk space used by the virtualization software binaries; (vi) the time needed to create and start the virtual machine instance; and finally (vii) the time needed to copy files and start the task in the virtual machine.

The QEMU binaries occupy around 2 MB and we ended up with a 350 MB Debian GNU/Linux 4.0 compressed image created using the Debian Network based install image, the size can be further cut down by removing non needed files (like documentation) from each installed package. We ran the test on a Linux/ Windows XP machine with Pentium IV 2.53GHz processor and 1GB RAM, from that only 160MB was allocated for a virtual machine instance. The instance images size varied from a couple of MB to 150 MB depending on whether the VM had to checkpoint or not (checkpoints are written to the instance images).

We ran each part of the test (shown in Figure 3.12) 20 times (chosen arbitrary). We used the same work unit for each run. While the VM Instance creation is done instantly, starting and stopping the instances takes time. We’ve found that around 50-60 seconds are needed to boot the image with only the minimal required services enabled, and around 20-22 seconds needed to shut down the instance. The results in Figure 3.12 do not include these times, they only show the real time the task was running, during the runs no checkpoints were made.

BOINC tasks consist of specially prepared binaries (application) and input files. The application is ought to implement an application-level checkpointing function and is constantly communicating with the client to report its used cpu time and status. The application asks and the client tells when is it time to checkpoint. BOINC applications have also a "fallback mode" implemented, if they are launched outside the client they enter the "standalone mode", thus they are able to run without the client. XtremWeb tasks have no such requirements, they are simple executables and input files.

3.2.3.4.2 GBAC: An application based approach

In the previous sub-section I (i) defined a criteria system for comparing different desktop virtualization solutions for desktop grids; (ii) evaluated the available tools (Bochs, QEMU, KQEMU, VMWare Player and VirtualBox); (iii) defined a generic architecture which allows building virtualized environments for task execution on desktop grid and volunteer resources; and (iv) did a reference implementation with focus on integration with BOINC and XtremWeb. I originally chose QEMU and KQEMU as the virtualization software for our implementation since at that time it was the solution best fitting to our criteria system. The second best solution was VirtualBox. Later I faced two issues which made us revise our initial approach. (1) My original proposition was an integrated solution where the virtualization based execution service is integrated with the given (desktop) grid middleware client. Others chose this approach as well [106]. The problem is that this would require extensive modifications in the client desktop grid software. In case of BOINC there are currently 8.7 million hosts running the client software [68] and only after an update would these hosts be able to run virtualization enabled tasks. This is not feasible and I think that a solution is required that is not tightly coupled with the client.
software and does not impose any modifications to the client. This can be achieved the easiest by including all virtualization related parts in a “traditional” application which would act as a wrapper. (2) QEMU dropped support for KQEMU and is left without acceleration support on Windows. QEMU itself (without KQEMU) is too slow, as our evaluation has shown [23], to be considered as a viable alternative. Contrary, VirtualBox has progressed a lot in the recent years. It was improved in many ways and currently based on our criteria system [22] I consider it the best virtualization solution for desktop grids. Several other implementations aiming at utilizing virtualization for desktop grids have emerged and these also build upon VirtualBox. For example, CernVM [107] and VBoxWrapper\footnote{http://boinc.berkeley.edu/trac/wiki/VboxApps} are available for BOINC. The common in these solutions is that both are explicitly developed for BOINC and are not intended to become a generic framework. Based on these two considerations I decided that instead of relying on QEMU in the future (without KQEMU support) I would switch to VirtualBox (see (1)); and instead of tight integration I have chosen a wrapper like approach (see (2)). We put the QEMU based implementation aside and chose VBoxWrapper as basis for a new implementation. The difference between our implementation and the basic VBoxWrapper one is that I intend to provide a generic framework and not a BOINC specific one. Currently our implementation supports BOINC, Condor and XtremWeb middleware beside standalone execution. Our approach differs in the following: (a.) I are providing a generic framework that can be used with multiple middleware, although for demonstration purposes in this work I use BOINC; (b.) I want to support multiple virtualized environments through multiple layered virtual appliances (see Section 3.2.3.4.3); and (c.) I consider GBAC as one of the foundations for volunteer clouds rather than only a wrapper able to execute applications within a virtualized environment.

The Generic BOINC Application Client (GBAC) is a virtualization based wrapper. Contrary to its name it aims to be a generic framework providing virtualized environments for various distributed computing infrastructures (DCIs). GBAC is implemented using the DC-API Meta API and does not rely on any middleware specific functionalities, thus it is possible to use it on any DCIs that are supported by DC-API. In the following I refer to the BOINC version of GBAC for demonstrating its concepts and internals. GBAC wrapper consists of the following components as shown in Figure 3.13: First, the wrapper...
binary (see 1. in Figure 3.13) itself is a BOINC enabled DC-API application that contains all BOINC related parts and handles communication with the BOINC client. Its task is to set up the client execution environment and manage the virtual machine on the client machine. Second, a supplied XML based configuration file (see 2. in Figure 3.13) is used to set the different parameters of the virtual machine: (i) the operating system type (e.g., Linux 64bit); (ii) the size of the allocated memory for the virtual machine; (iii) whether the machine should have network access; (iv) which virtual appliance to use; and (v) whether to enable a shared directory between the host and the guest (the virtual machine). The third component is a virtual appliance (see 3. in Figure 3.13) that contains the operating systems and libraries for the virtual machine. This image contains a 32bit Linux installation with some GBAC related components that will be detailed later in Section 3.2.3.4.2. The wrapper sets up the client execution environment first by creating a shared directory for the virtual machine. It puts all input files in this directory. GBAC does not separate the binaries of the legacy application (typically a parameter sweep application as described in the introduction) and its input files (see 4. and 5. in Figure 3.13). This means that all legacy application binaries and their input files are normal input files for GBAC and are not part of the GBAC BOINC application. As a next step the virtual machine is started using VirtualBox. GBAC does not contain VirtualBox; it is a prerequisite that every host has VirtualBox preinstalled before GBAC can be used. Next, the legacy application is executed in the virtual machine and the results are copied to the shared directory. Once the application finished the virtual machine shuts down and finally GBAC copies the results from the shared directory to the work directory of the BOINC client and terminates. In the following subsections I detail the functionality of GBAC by describing a job submission generated from a parameter sweep application that is not registered at the BOINC server and hence without GBAC BOINC would not be able to handle it. In this description of BOINC I assume that the PS application is initiated either in a service grid VO (gLite, ARC or UNICORE) that is extended with a BOINC DG system. 3G Bridge [44] plays a crucial role connecting the SG systems with BOINC and enabling the job submission from WS-PGRADE to BOINC.

Architecture The 3G Bridge provides a job handler service interface for accepting job submissions (see 1. in Figure 3.14). It assigns jobs to different job queues and stores them in its job database. Different DCI plug-ins can be used to submit (forward) jobs to different DCIs. 3G Bridge also provides extended services like a web service interface to add and query jobs.

3G Bridge keeps a list of the supported algorithms (applications in case of desktop grids) for each configured DCI. When a job is received (see 2. in Figure 3.14), 3G Bridge checks if the job fits to a registered algorithm (see 3. in Figure 3.14). If the algorithm is not registered it means that the application is not deployed so the job cannot be executed. When GBAC is registered in the algorithm Queue of 3G Bridge the job for the “unknown” PS application is redirected to it with one constraint, namely the task should contain not only the input files (which is normal for desktop grid job submission) but the application binaries as well or else the execution will fail. It is possible to include all application
related files in a special named bundle (zip or tar.gz). This is only for user convenience and it eases job submission. Once the job is internally redirected to the GBAC queue, 3G Bridge submits it to the desktop grid via its configured DC-API plug-in (see 5. and 6. in Figure 3.14) that generates a GBAC work unit for the connected BOINC server (see 6. in Figure 3.14). After this BOINC will register the new work unit for the deployed GBAC application and when a client with deployed VirtualBox asks for tasks BOINC will assign the work unit to the client. The current client side implementation of GBAC is based on VirtualBox and the BOINC client is able to detect and report to the BOINC server whether VirtualBox is installed on the client. The BOINC server will assign GBAC tasks only for those hosts that have VirtualBox preinstalled.

First a legacy application with its inputs is submitted via the 3G Bridge to BOINC (see 1. and 2. in Figure 3.15) is transformed into a GBAC application work unit containing the binaries and input files of the legacy application as inputs of the GBAC application. When a BOINC client connects, from a host where VirtualBox is installed, to the BOINC server and asks for tasks, it receives the work unit containing the GBAC application with its inputs. The BOINC client first downloads the GBAC binary, its configuration file and the virtual appliance along with the legacy application binaries and the input files (see 3. in Figure 3.15).

After the download finished the client starts the GBAC application. The first task of GBAC is to bootstrap the execution environment: it creates a directory that will be shared between the host and guest and all legacy application binaries and input files are copied here (see 4. in Figure 3.15). Also a special file is put into this directory that contains additional parameters of the application (e.g., command line parameters and environment variables). After this the virtual machine is started (see 5. in Figure 3.15) using the Vir-
3.2. Towards volunteer clouds

Figure 3.15: GBAC: execution of a sample task

tualBox command line interface (“VBoxManage”). All parameters for the virtual machine are set in a configuration file for GBAC. This configuration file is currently part of the application and common for every GBAC application work unit, but it can also be supplied individually for each task allowing more customization for the virtual machine. The configuration file also contains a reference to the virtual appliance to be used. Currently a single Linux appliance has been developed for GBAC but using more appliances is not prohibited by the GBAC concept. The appliance contains the GBAC guest extensions (see 6. in Figure 3.15) and these are started right after the boot process finished. First, a component checks if there are any application bundles found in the shared directory. If yes, it extracts the contents to a separate sandbox within the virtual machine. If not, then all files are simply copied to the sandbox. After this the legacy application is started in the sandbox (see 7. in Figure 3.15). Volunteer resources are highly volatile and any application running on these resources in the framework of a volunteer computing project should be prepared for interruption or shut down. This problem is usually solved by including an application specific (application-level) checkpointing mechanism in each application. The advantage of GBAC is clear here: application-level checkpointing is not needed since GBAC uses the system-level mechanism provided by the virtual machine monitor. GBAC can be suspended or stopped at any time regardless of what application it is executing and the suspended GBAC application can be resumed either on the current client or on a new client. While the virtual machine is running, GBAC continuously monitors its status through the virtual machine monitor. After the execution of the legacy application has finished, all new and changed files are copied to the shared directory by the guest extensions (see 8. in Figure 3.15). Finally the virtual machine is instructed to shut itself down.
Next GBAC notices that the virtual machine has terminated and it will copy all changed files from the shared directory to the working directory and terminate itself (see 9. in Figure 3.15). From here the BOINC client takes over, it will contact the BOINC server (see 10/a. in Figure 3.15), upload the output files (see 10/b. in Figure 3.15) and report the completion of the task. Next 3G Bridge fetches the results from BOINC (see 11. in Figure 3.15) and will return it to the submitter (see 12. in Figure 3.15). BOINC employs reporting techniques which allow (i) the native applications to report their status (e.g., completion ratio) and (ii) measure the CPU time used for computation of the current task. These data are visualized in the BOINC client so that volunteers know what the status of the current task is and what to expect. Also for each task an upper limit for used resources (FLOPs, disk and memory) is set by the entity that created the task (in this case by 3G Bridge). In case of applications running in a virtualized environment (which acts basically like a sandbox) the reporting techniques do not work. The current implementation of GBAC does not allow reporting completion status (see (i)), instead the ratio is calculated from the ratio of used and estimated FLOPs. CPU time is not measured directly either (see (ii)) rather the total time used by the virtual machine is reported.

We think this is better since the overhead caused by the virtual machine should also be included in the total (and thus the volunteer should be rewarded for that). GBAC also considers the overhead (CPU and memory) introduced by using virtualization. Currently when a task is submitted via 3G Bridge to GBAC the upper resource limit for the task is automatically increased by a predefined percentage. In case the submitted task takes more memory than predefined, the OS (in virtual machine) can use its swap partition (if configured) or the app will simply aborts with “run out of memory” error.

### 3.2.3.4.3 Virtual appliance management

GBAC provides a standardized virtual environment for tasks of applications and services. Currently there are base virtual appliances available that are based on Debian Linux and Scientific Linux. For each task a new virtual machine instance is started with the appliance by GBAC and shut down after the task is finished. There are two concerns with this: (i) although tasks are executed inside a sandbox within the virtual machine, if a task is somehow able to break out and do modifications to the file system of the virtual machine (e.g., the kernel is deleted) that might render it unusable permanently; (ii) after each execution a cleanup procedure is required to remove the remnants of the previous task.

To overcome these problems I chose to compose the virtual appliance for GBAC of multiple overlay images (appliances) as shown in Figure 3.16 First our base virtual appliance is immutable, meaning that no modifications can be done to it (see (i)). This is enforced not at the file system level, rather it is guaranteed by the hypervisor. It redirects all disk I/O to a separate overlay image which is specific to each running virtual machine instance. This also solves the problem of cleaning up the sandbox (see (ii)) after each execution, since the instance image can be simply thrown away. Second, I introduce optional application specific overlay images which can contain all application specific dependencies that are supplementary for the base image. In this case the immutable image is a compos-
3.2. Towards volunteer clouds

Figure 3.16: A virtual appliance for GBAC is composed of a base image, an optional application overlay image and an instance image.

3.2.4 Evaluation

Table 3.3 evaluates the discussed tools and APIs based on four criteria. The first criteria is Application development. This evaluates what middleware is supported, whether both native and legacy applications are supported and the supported programming languages. As GBAC utilizes Abstractions A1 (via DC-API), A2 and A3-2 it provides the most complete solution. For supported middleware DC-API has explicit support for Condor, BOINC, however DC-API enabled applications run perfectly under XtremWeb and possibly other middleware as well.

The second criteria is Execution environment. It compares whether the applications are executed in a native environment (e.g., BOINC applications are run under the Core Client either directly or via a mediator; or as standalone); whether there is any checkpointing support for long running workunits; and whether complex applications can be satisfied (i.e., Abstraction A3-1 or A3-2). All evaluated tools provide a native execution environment with the exception of GBAC as it executes applications in a sandbox, however this has no drawbacks since it implemented in a way that it forwards or handles all messages between the application and the worker. Only GenWrapper and GBAC support complex dependencies for applications such as deploying R for ‘Patient-Readmission’ (see Table A.2. Finally only GBAC (through its hypervisor) is able to provide middleware independent check-pointing support. For example native applications should have an own check-pointing and resume function implemented (e.g., see Rules R4.19 and R4.23 in MOINC).

The third criteria is Security from the aspects of isolation and resource limits enforcement. In case of BOINC the Core Client (worker) provides resource limiting capabilities and also operating system level isolation. Applications are executed via a limited user account. Here all tools rely on this middleware and OS based limits with the exception of GBAC since it is utilizing Abstraction A3-2 (a hypervisor effectively).

The final criteria is Deployment. It compares the requirements and ease of deployment. Here from the perspective of the application GBAC is the most convenient as there appli-
### 1. Application development

<table>
<thead>
<tr>
<th>BOINC API (-)</th>
<th>DC-API (A1)</th>
<th>BOINC (A2)</th>
<th>Wrapper</th>
<th>GenWrapper (A1, A2, A3-1)</th>
<th>GBAC (A1, A2, A3-2)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>a. Supported Programming languages</strong></td>
<td>C/ C++/ FORTRAN/ Python</td>
<td>C/ C++/ Java/ Python</td>
<td>XML based Control-flow</td>
<td>POSIX shell scripting</td>
<td>None required</td>
</tr>
<tr>
<td><strong>c. Legacy application support</strong></td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>d. Native application support</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Partial</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>e. Application level check-pointing</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Partial</td>
<td>Partial</td>
<td>Partial</td>
</tr>
</tbody>
</table>

### 2. Execution environment

| **a. Type** | Native | Native | Native | Native | Virtualized |
| **b. Support for complex application dependencies** | No | No | Partial | Yes | Yes |
| **c. System/process level checkpointing** | No | No | No | No | Yes (system) |

### 3. Security

| **a. Isolation (sandbox)** | Partial (middleware, OS) | Partial (middleware, OS) | Partial (middleware, OS) | Partial (middleware, OS) | Yes (hypervisor) |
| **b. Resource usage limit enforcement** | Partial (middleware) | Partial (middleware) | Partial (middleware) | Partial (middleware) | Yes (hypervisor) |

### 4. Deployment

| **a. No server-side application deployment** | No | No | No | No | Yes |
| **b. Same application binaries for all platforms** | No | No | No | No | Yes |
| **c. Supported major platforms** | Windows/Linux/Mac OS X | Windows/Linux/Mac OS X | Windows/Linux/Mac OS X | Windows/Linux/Mac OS X | Windows/Linux/Mac OS X |
| **d. No client-side third party dependencies** | Yes | Yes | Yes | Yes | No (VirtualBox predeployed) |

Table 3.3: Capabilities and requirements of different tools supporting application porting for volunteer computing

Applications can be run as a per workunit basis and this requires no deployment beforehand. However it requires that the donated resource is able to run the VirtualBox hypervisor and has it deployed (current BOINC Client installations come with VirtualBox bundled).

### 3.3 Improving batch completion times

A typical BOINC based VC project has thousands of registered volunteer hosts [68]. For example EDGeS@home has over 21,000 (see Section A.8.1 for more details). However the active number of hosts is always lower and varies over time depending on the volunteers’ usage pattern, e.g., some only run BOINC when their computer is idle, but some turn off completely theirs, when they are about to be idle for longer periods of time. Hardware failures on volunteer resources cannot be neglected either. The consequence of these is that the task failure and completion deadline miss ratios can be high. In case of batches of tasks a single delayed task is going to affect the completion time of the whole batch resulting in the so called tail-effect [116]. There are different methods available for improving task completion times: (i) redundant computing, (ii) reliability and availability based scheduling, and (iii) resubmission. Redundancy (i.) means that multiple copies of the same task are distributed to clients with the expectation that at least a single one is going to finish in time. Reliability and availability based scheduling (ii.) means that the system will
3.3. Improving batch completion times

Prioritize the reliable resources over less reliable and less available. Finally, resubmission (iii.) identifies problematic tasks and submits them again, e.g., in case of a batch, based on empirical results, the last 10% uncompleted tasks can be resubmitted to mitigate the tail effect [11]. Each of these methods on its own will improve the completion time to some extent; however, usually a combination of these is used. The problem is always twofold. First, the set of resources that is considered highly available and reliable must be determined. Second, a redirection mechanism is required for the tasks. Determining reliability and availability is a challenge of its own; however, there are already good mechanisms implemented e.g., for BOINC, that can be used as a foundation [117].

For this theorem I present work unit handling algorithms that I developed that reduce the known “long-tail” effect for makespan in volunteer computing systems:

**Theorem II.3.:** Algorithms "Adapt redundancy", "Prioritize delayed jobs" and "Prioritize job sources" mitigate the negative footprint of the “tail-effect” for makespan on volunteer computing systems under the predefined conditions.

I demonstrate with measurements that the algorithms successfully reduce makespan in real-world scenarios: deployed at SZTAKI Desktop Grid and EDGeS@home volunteer computing projects. The improvement and effectiveness of the algorithms depends on certain criteria that are detailed in the following sections.

Related publications: [41], [118], [119], [39] and [40].

3.3.1 Related work

Makespan (or batch completion time) improvement on desktop grids through the use of (a) different direct scheduling strategies (e.g., periodic resubmission) [120, 121], (b) statistical modeling of availability and reliability [99], (c) checkpointing [122], or (d) elasticity via on-demand dedicated resources (“cloud-bursting”) [123, 116], is a very active field of research. Reynolds et al. [123] investigate how to reduce the makespan of scientific workflows running on desktop grids by augmenting them with dedicated resources from Infrastructure as a Service Clouds. They are addressing the “tail problem” of volatile resources where a small percentage of a batch or workflow is assigned to resources which will never return the results (e.g., the resource or the desktop grid software running on them is shut down permanently; they have some hardware failure, etc.). In these cases the task will “time out” after a period and the desktop grid will resend it to a different resource. However this radically increases the makespan of the whole batch or workflow. To overcome this problem the authors use dedicated cloud resources (“cloud-bursting”) to compute the remainder of the batch after a fixed percentage is completed by the desktop grid. The authors assume that the start of the tail can be characterized by the completion of a fixed percent of the total number of tasks in a batch, and their results show that they were able to reduce the makespan by up to 40% this way.
Chapter 3. Challenges and future directions of volunteer computing

SpeQuloS [116] aims to provide Quality of Service (probabilistic guarantee for makespan) for best-effort distributed computing services like desktop grids by re-allocating jobs to more reliable resources and allocating dedicated resources when needed (cloud-bursting). It is similar to the here proposed solution however, it requires user interaction as users must explicitly request the speed-up of their batches and must pay for it by virtual credits that can be earned by volunteering their own machine for the target desktop grid site. The more capacity they offer, the more credit they collect. The more credit they have, the more jobs can be redirected to cloud resources and the less completion time the batch will reach. Kondo et al. [120] propose resource selection techniques to improve performance and reduce makespan on desktop grids for applications that require rapid turnaround times. Three resource selection techniques are proposed: (i) resource prioritization, (ii) resource exclusion and (iii) different task replication strategies. They evaluated these via simulation running on data based on collected traces from real desktop grid configurations. Their main conclusions are the following: (a) in resource prioritization using static clock speed information delivered the best results; (b) using dynamic information like historical host availability does not improve application performance much more, but (c) by using different task replication techniques performance can dramatically increase. Bouguerra et al. [122] investigate strategies for scheduling checkpoints of sequential jobs on desktop grids. Their model is based on “lost computation time” and “re-execution ratio” and introduces the “cumulative checkpointing overhead”. Their results are achieved using a simulation framework (SimGrid) with parameters from real-world systems. They conclude that (i) their proposed model outperforms periodic checkpointing and standard execution and (ii) using fault tolerance blindly (both checkpointing and replication) can lead to dramatic system performance deterioration. Iglesias et al. [99] investigate long-term availability for groups of volatile volunteer hosts to broaden the set of applications usable on desktop grids. They consider collective availability as a key factor for enabling parallel applications and workflows for volunteer computing. They evaluated their methods using availability traces from an existing volunteer computing project (SETI@home). They represent the availability of each host as a binary vector where each bit denotes an hour of a week. They use the binarization threshold for deciding whether a host is considered online during a given hourly period. The authors used the k-means algorithm to cluster the hosts, and they found that the largest clusters are made of the always-on and always-off hosts. They used the first weeks of life of a host as training data and they use this to predict the behavior of a host, and define different metrics for measuring the quality of prediction. Their results show that service deployment based on the binarization approach can achieve higher availability with smaller redundancy.

Lei Ni et al. [124] propose the “next generation Volunteer Computing systems on top of well studied Peer-to-Peer techniques to fully take advantage of its decentralized characteristic and its very large, shared data storage capacity”. Their proposed system is based on a P2P-Tuple system where the execution of a job is done by a kind of cooperation of the peers. The paper proposes to handle the tail effect in a way that peers are able to push unfinished jobs to each other. Based on this, completion time of a job can be much lower than in a normal BOINC system.

Various pilot systems like DIRAC [125] or Diane [126] are widely used by scientists
3.3. Improving batch completion times

these days. In these systems, a single pilot job is submitted through gLite, as a placeholder, to a given computing element (CE). The pilot job pulls jobs from the pilot system’s job repository for execution on the CE, and transfers information and results back. Pulling jobs and all communication are executed by-passing the gLite infrastructure, reducing the overhead of job submission from linear to constant time. This is also a way of decreasing the overhead of the gLite system; however it is not targeting to use volunteer resources and hence the number of accessible resources will be much smaller than in a volunteer system.

3.3.2 Improving completion time with on-demand reliable resources

In the following I present a job assignment method to improve task completion times using resubmission with reliability based scheduling. A scheduler keeps track of the number of assigned tasks for each resource group and assigns new tasks based on the number of resources available in that group. It also provides an interface extending the administrator interface of BOINC for querying the status of assignments. Reassignment is performed in round-robin order, where each configured cloud has a weight and capacity assigned and a global limit for the assignments total. The scheduler keeps track of the total number of assigned jobs and the currently processed ones for each resource group. It continuously keeps the number of assignments at the global limit while removing finished assignments and adding new ones based on the group limit and prioritizing between resources based on the group weights.

3.3.2.1 Job assignment

In the following I present the method for the job assignment. First I define the basic constructs, and later discuss in detail the selection algorithm.

JOB is an abstract unit of (compute-intensive) work in the desktop grid, while JOBS represent all units of work present. An instance of a JOB is referred as a TASK and is represented as follows:

\[ \text{JOB : } \langle \text{APPLICATION, CREATED, STATUS, PRIORITY} \rangle \]  
\[ \text{TASK : } \langle \text{CREATED, STATUS, JOB, PRIORITY} \rangle \]

The APPLICATION attribute represents a deployed application. Each job belongs to an application and each task belongs to a job. The CREATED attribute represents the date when the job or task was created. STATUS construct represents the status of one:

\[ \text{STATUS } = \{ \text{unsent, inprogress, finished, error} \} \]

A job or task is either waiting in the server (\( J \in \text{JOB} : \text{status}(J) = \text{unsent} \)), or being processed on a host (\( \text{status}(J) = \text{inprogress} \)), successfully finished (\( \text{status}(J) = \text{finished} \)) or is erroneous (\( \text{status}(J) = \text{error} \)). PRIORITY denotes a ranking between jobs made during the assignment. The ASSIGNMENTS tuple contains the list of assignments in the system, while ASSIGNMENT represents a single assignment:

\[ \text{ASSIGNMENT : } \langle \text{JOB, TASK, CLOUD, CREATED} \rangle \]
During an assignment, a new entry is made with the JOB, CLOUD and CREATED attributes set. Initially TASK is undef. Once a resource provided by a cloud starts processing the assignment, TASK is updated with the actual instance it is processing. A cloud is represented by:

\[
\text{CLOUD : } <\text{VMLIMIT}, \text{VMCOREUNIT}, \text{VMCOREFREE}, \text{WEIGHT}> \tag{3.13}
\]

While CLOUDS represents all configured clouds in the system. VMLIMIT represents the upper limit of running virtual machines (VMs) in the cloud, may it be from a predefined quota (private clouds) or monetary or contractual limits (public clouds). However, it is always set manually. VMCOREUNIT represents the number of CPU cores per VM. It is assumed that all VMs are homogeneous in a single cloud. VMCOREFREE represents the number of free CPU cores in the cloud with regard to VMLIMIT. We assume that the VMs are available (e.g., automatically started without delay) when needed. WEIGHT allows defining a rank between the different configured clouds based on arbitrary parameters (e.g., performance, capacity or monetary cost). The assignmentMain (see Algorithm 3.1) main loop performs the assignments of jobs. It is configured via five parameters: (1) SLOTPERCORE represents the number of assignment slots per CPU core assuming that a task is single threaded and requires a single CPU core. (2) APPSELECTED represents the applications on the desktop grid that are selected for assigning their jobs to clouds. (3) ASSIGNMAX represents a global limit for the total number of assignment entries. (4) MINAGE represents the minimum age of an assignment to be considered obsolete and removed from the assignments. Finally (5) VMLIMIT must be set for each configured cloud. AssignmentMain utilizes different functions as follows. assignJobs is a CLOUD \times JOB^n \times INTEGER \to JOB^m function and assigns INTEGER number of jobs to CLOUD from the JOB^n list, by creating a new ASSIGNMENT in ASSIGNMENTS. Finally it returns a list of the jobs that remained unassigned from the input job list (JOB^m | m \leq n).

slotsFree is a CLOUD^n \to INTEGER function. It calculates the number of free assignment slots for CLOUD based on VMCOREFREE and SLOTPERCORE. These two parameters are not used directly by the algorithms, rather indirectly trough slotsFree. slotsFree is a CLOUD^n \to INTEGER function. It calculates the number of free assignment slots for CLOUD^n. The olderThan: ASSIGNMENT, JOB \times TIME \to BOOLEAN function checks if an assignment is older than the given TIME and returns \{true, false\}.

The main assignment function is a loop (see Algorithm 3.1). During each iteration first it starts with housekeeping by removing the assignments that are already processed (or currently being processed), or are older than set by MINAGE (typically set to 2 weeks), to free up assignment slots. Next it calls the getjobsAvail function. This function contains the job selection algorithm for the assignment and it returns the available jobs in decreasing priority order (see JOBSAVAIL in Algorithm 3.1). Next it iterates through the configured clouds in decreasing weight and tries to assign the maximum possible number of jobs to them. This amount is determined by the minimum of (i) the available jobs (JOBSAVAIL in Algorithm 3.1); (ii) the number of free assignment slots in cloud C (slotsFree(C) in Algorithm 3.1) and (iii) the total number of available free slots in
3.3. Improving batch completion times

Algorithm 3.1 Common loop for invoking the job selection algorithm for Prioritize delayed jobs and Prioritize job sources algorithms and performing the job assignments.

**Requires:** ASSIGNMENTS ≠ undef, MINAGE > 0, JOBS ≠ undef

**Returns:** ASSIGNMENTS ≠ undef

```
loop
  for A ∈ ASSIGNMENTS do
    if task(A) ≠ undef ∨ olderThan(A, MINAGE) then
      ASSIGNMENTS(A) ← undef
    end if
  end for
  ASSIGNAVAILC ← ASSIGNMAX − |ASSIGNMENTS|
  JOBSAVAIL ← getJobsAvail(
    min(slotsFree(CLOUDS), ASSIGNAVAILC),
    ASSIGNMENTS,
    JOBS)
  if ASSIGNAVAILC > 0 ∧ slotsFree(CLOUDS) > 0 ∧ |JOBSAVAIL| > 0 then
    for C ∈ sort(CLOUDS, desc(WEIGHT)) do
      if slotsFree(C) ≤ 0 then
        continue
      end if
      ASSIGNCOUNT ← min(|JOBSAVAIL|, slotsFree(C), ASSIGNAVAILC)
      JOBSAVAIL ← assignJobs(C, JOBSAVAIL, ASSIGNCOUNT)
      ASSIGNAVAILC ← ASSIGNMAX − |ASSIGNMENTS|
      if ASSIGNAVAILC ≤ 0 ∨ slotsFree(CLOUDS) ≤ 0 ∨ |JOBSAVAIL| = 0 then
        break
      end if
    end for
  end if
  wait()
end loop
```

the assignment table. JOBSAVAIL is updated by the assignJobs function during each iteration. It removes already assigned tasks from the list.

The first algorithm is Prioritize delayed jobs and is described in Algorithm 3.2. It provides a getAvailableJobs function for Algorithm 3.1. This algorithm returns the requested number of unfinished jobs in ascending creation date order. It allows to continuously flush out the oldest delayed unfinished jobs. Usually, these jobs already have some finished tasks; however, not enough to form a consensus and select a final solution. BOINC grants credits to donors only after finalizing the result and this led to some complaints from the donors that although they finished a task long ago, they did not yet received credit for it.

Additionally to prioritizing generally old jobs, other attributes should be considered. Namely in progress tasks that can be considered late or delayed according to some metric. Also selecting specific applications for the assignment mechanism. These attributes make the selection algorithm conform to the "non-interactive" QoS behavior for EDGI, since it needs primarily to support EDGI applications (and EDGI job sources), thus ensure that these have priorities over others.
Algorithm 3.2 Prioritize delayed jobs: Delayed job prioritization algorithm (getJobsAvail procedure for Algorithm 3.1)

Requires: JOBCOUNT ≥ 0, ASSIGNMENTS ≠ undef, JOBS ≠ undef
Returns: —

function getJobsAvail(JOBCOUNT, ASSIGNMENTS, JOBS)
    JOBSOUT ← ∅
    for J ∈ sort(JOBS, "asc created") do
        if J ∈ ASSIGNMENTS ∨ status(J) = finished ∨ status(J) = error then
            continue
        end if
        if |JOBSOUT| ≥ JOBCOUNT then
            break
        end if
        priority(J) ← 0
        JOBSOUT[] ← J
    end for
    return JOBSOUT
end function

Algorithm 3.3 Prioritize job sources: Prioritize delayed jobs from distinguished job sources algorithm (getJobsAvail procedure for Algorithm 3.1)

Requires: JOBCOUNT ≥ 0, ASSIGNMENTS ≠ undef, JOBS ≠ undef
Returns: JOBSOUT

function getJobsAvail2(JOBCOUNT, ASSIGNMENTS, JOBS)
    JOBLIST_a ← getJobsAvail2(JOBCOUNT)
    JOBLIST_b ← ∅
    JOBSNEEDED ← JOBCOUNT − |JOBLIST_a|
    if JOBSNEEDED > 0 then
        JOBLIST_b ← getJobsAvail2(JOBSNEEDED)
    end if
    JOBSOUT ← {JOBLIST_a, JOBLIST_b}
    return JOBSOUT
end function

Algorithm 3.3, 3.4 and 3.5 show the second job selection algorithm: Prioritize job sources. Here the getTasks function is a JOB → TASK function that returns all tasks belonging to a job. The isFromEDGI: JOB → BOOLEAN function checks if the job originates from an EDGI job source, e.g., from a specific gLite Computing Element or 3G Bridge. These jobs are arriving to the desktop grid with a higher priority than other jobs. Note this priority denotes a different attribute than the PRIORITY used during assignment. It acts as a simple filtering attribute. The ageMin: APPLICATION → TIME function is used to query the minimum age for a job belonging to a specific application before it can be considered late. Being late means that the task was assigned to a donor machine, and it should have been finished and returned to the server by now. This is at least an application specific parameter but can vary greatly job by job. We assume that for all jobs belonging to the same application a single scalar represents this characteristic. This value is empirically set usually to \( \bar{x}_P + \sigma(x_P) \), where \( \bar{x}_P \) is the mean and \( \sigma(x_P) \) is the standard deviation of execution times of tasks belonging to
3.3. Improving batch completion times

Algorithm 3.4 Prioritize job sources: getJobsAvail function for Algorithm 3.3.

Requires: JOBCOUNT ≥ 0, ASSIGNMENTS ≠ undef, JOBS ≠ undef
Returns: JOBSOUT

function getJobsAvail(JOBCOUNT, ASSIGNMENTS, JOBS)
    JOBSOUT ← ∅
    for J ∈ JOBS do
        if J ∈ ASSIGNMENTS ∨ ¬ isSelected(application(J)) ∨
            status(J) = finished ∨ status(J) = error ∨ ¬ isFromEDGI(J) ∨
            getTasks(J) = undef then
            continue
        end if
        priority(J) ← 0
        for T ∈ getTasks(J) do
            priority(T) ← 0
            if olderThan(J, ageMin(application(J))) ∧ status(T) = inprogress then
                priority(T) ← priority(T) + 2
            else
                if status(T) = finished then
                    priority(T) ← priority(T) − 10
                end if
            end if
        end for
        priority(J) ← max(priority(getTasks(J)))
        JOBSOUT[] ← J
    end for
    return JOBSOUT
end function

Algorithm 3.5 Prioritize job sources: getJobsAvail function for Algorithm 3.3.

Requires: JOBCOUNT ≥ 0, ASSIGNMENTS ≠ undef, JOBS ≠ undef
Returns: JOBSOUT

function getJobsAvail(JOBCOUNT, ASSIGNMENTS, JOBS)
    JOBSOUT ← ∅
    for J ∈ sort(JOBS, desc(CREATED)) do
        if J ∈ ASSIGNMENTS ∨ ¬ isSelected(application(J)) ∨
            status(J) = finished ∨ status(J) = error ∨ J ∈ JOBSOUT ∨
            isFromEDGI(J) = false then
            continue
        end if
        if ¬ (inprogress, finished ∈ status(getTasks(J))) then
            JOBSOUT[] ← J
        end if
    end for
    if |JOBSOUT| ≥ JOBCOUNT then
        break
    end if
    return JOBSOUT
end function
application $P$. This assumes that we know the characteristics of the applications and jobs. This conforms to the desktop grid use case where usually parameter study type jobs are executed and the applications rarely change. The $getJobsAvail$ function in Algorithm 3.3 supports the EDGI use case. It contains two parts. The first part acts as the main selection algorithm while the second part fills the remaining assignment slots. This is achieved by first selecting jobs that have tasks based on a priority descending and creation date ascending (see Algorithm 3.4); and filling the remainder slots with other jobs that have either no tasks at all and no finished or in progress ones (see Algorithm 3.5). These ones are sorted from the opposing direction (ordering by creation time descending). First the jobs are checked against the following filtering criteria: (a) they should originate from an EDGI job source; (b) belong to a supported application, (c) are not currently assigned, (d) are not finished ones and (e) have tasks. Next, the priority is set based on whether a task belonging to the job is considered late or it already has a successfully finished task. The latter is penalized compared to the former and normal tasks since we want to prioritize jobs that have overdue tasks. The priority of a job is determined by its highest priority task. At the end of the section the selected jobs are ordered based on their previously set priority and their creation date. If there are not enough jobs available ($|JOBSOUT| < JOBCOUNT$) the second part tries to fill up the remaining slots.

The second half (see Algorithm 3.5) performs the same checks as the first part with one exception (see list of filtering criteria defined above: a–e.): the job is not required to have tasks. Only those jobs are added to the list that have either no tasks or have erroneous ones. The two parts of the algorithm form disjunctive sets of jobs that are merged at the end and returned.

<table>
<thead>
<tr>
<th>Cloud Acronym</th>
<th>Cloud Provider</th>
<th>EDGeS@home User</th>
<th>Middleware</th>
<th>Number of Resources (up to)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CICA</td>
<td>Centro Informático de Andalucía, Spain</td>
<td>CloudCICA</td>
<td>OpenNebula</td>
<td>100</td>
</tr>
<tr>
<td>LPDS</td>
<td>MTA SZTAKI LPDS, Hungary</td>
<td>CloudLPDS</td>
<td>OpenNebula</td>
<td>64</td>
</tr>
<tr>
<td>UniMainz</td>
<td>University of Mainz, Germany</td>
<td>CloudMAINZ</td>
<td>OpenStack</td>
<td>32</td>
</tr>
<tr>
<td>UNIZAR</td>
<td>University of Zaragoza, Spain</td>
<td>CloudUNIZAR</td>
<td>OpenStack</td>
<td>50</td>
</tr>
<tr>
<td>UoW</td>
<td>University of Westminster, UK</td>
<td>UoW</td>
<td>OpenStack</td>
<td>52</td>
</tr>
<tr>
<td>EC2</td>
<td>Amazon EC2</td>
<td>CloudAmazon</td>
<td>Amazon WS</td>
<td>20</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>318</strong></td>
</tr>
</tbody>
</table>

Table 3.4: Cloud (IaaS) resource providers and number of resources for EDGeS@home.

3.3.2.2 Evaluation

The performance of the augmented and non-augmented volunteer desktop grids are evaluated by submitting batches of 10,000 jobs and then monitoring their execution status over time. I executed the tests in the EDGeS@home volunteer desktop grid described in A.8.1. The whole scenario is detailed in Appendix A.8. Table 3.4 shows the IaaS clouds and
3.3. Improving batch completion times

Figure 3.17: Task reassignment in the non-interactive QoS process in EDGI for EDGeS@home

number of resources made available for EDGeS@home in frame of the EDGI project and Figure 3.17 describes the high-level architecture (detailed in Appendix A.8). The private clouds used are experimental (with the exception of the LPDS cloud) and have a quota set for the number of VMs (assuming a single CPU core per VM). Public clouds offer seemingly unlimited resources, but in practice for example Amazon WS limits the number of instances for each account (see Amazon EC2 in Table 3.4), however this limit may be increased upon submitting a request form\textsuperscript{12}. Please note that EDGeS@home was shut down at end of 2015 due to lack of funding, however all data is archived and available upon request.

To augment the desktop grid, I set up $VMLIMIT = 87$ dedicated clients in the pool of clouds defined in Table 3.4. With $VMCOREUNIT = 1$ cores in each VM (to save resources), and $SLOTPERCORE = 3$ slots for each core. $SLOTPERCORE$ value was defined based on experiences with the operation of the BOINC client. With a minimal caching configuration the client has always one workunit being executed, one in its local cache and I added one more workunit to its virtual queue by defining $SLOTPERCORE = 3$. I determined the global maximum number of assignments, $ASSIGNMAX = 261$. This is the result of the following equation:

$$ASSIGNMAX := VMLIMIT \times VMCOREUNIT \times SLOTPERCORE$$ (3.14)

This value can be any arbitrary positive integer; choosing the total number of available slots in the supporting clouds as a global maximum is a reasonable choice. I used the PublicAutodock \cite{PublicAutodock} application (desktop grid implementation of the well-known AutoDock \cite{AutoDock} tool) in the measurements. To determine the time threshold in the assignment algorithm $\text{ageMin}(PublicAutodock)$, the execution time of this application was determined. The execution time of a successful work unit of this application on the non-augmented desktop grid exhibited the distribution shown in Figure 3.18. The distribution was estimated based

\textsuperscript{12}http://aws.amazon.com/ec2/faqs/#How_many_instances_can_I_run_in_Amazon_EC2, accessed on 2014-03-20
on 8000 successful executions of the application in the desktop grid. Based on this dis-
tribution, I determined the threshold in the assignment algorithm as the mean execution
time plus the standard deviation. This was 2567 seconds, which I rounded up to 2600. This
nearly coincides with the 90th percentile supporting that any work unit running longer
than this can be considered as an outlier, a probable cause of the tail effect, and should
get priority to be assigned to the supporting cloud.

![Figure 3.18: PDF of the running time of the Autodock application in the non-augmented desktop
grid](image)

![Figure 3.19: Performance comparisons of the augmented and non-augmented desktop grids](image)

After I have determined the parameters for the algorithm, several batches of jobs were
submitted to the EDGeS@home desktop grid. After submitting a batch, in five minute
intervals, I recorded the number of jobs remaining in that batch. Over time, the graph of
these records shows us the completion time of the batch, the speed with which the batch has
been executed, and whether the tail effect is present. The results of our measurements are
shown in Figure 3.19. Thicker lines and bold tick labels on the x axis show the execution
graphs of the meta-jobs executed on the augmented desktop grid ("With cloud"). All
graphs start from 10000 remaining jobs (vertical axis), which is not shown in this figure.
The completion time for each meta-job is denoted on the horizontal axis. For both cases,
performed five measurements were performed, although only four are visible for each: with
cloud, two batches took the same amount of time (5.7 hours) to complete; without cloud,
one of the batches still had 58 unfinished jobs after 192 hours of running, then it was
3.3. Improving batch completion times

terminated. The slope of each graph shows the speed with which that batch was executed. We can see that as long as there are many remaining jobs, the batch is executed quickly with and without cloud support: all graphs fall steeply from 10,000 to about 30-70 at the beginning. This is because the throughput of the desktop grid is high in both cases. The cloud does cause a small speedup – all “With cloud” graphs are to the left of all “Without cloud” graphs – because, after all, these are dedicated resources. But the real problem arises when there are only a few jobs left in a particular batch: the plateaus at the end of most of the graphs are the results of the tail effect. Notice how shorter the tails are when the desktop grid is augmented with our cloud solution. In one such case (one of those ending at 5.7 hours), the graph falls into the horizontal axis instead of plateauing—the tail effect is not present at all. Figure 3.20 displays the number of running, finished, erroneous and on the server waiting jobs (“Init”) for six different batches. The first four are for cloud augmented executions while the last two are for non-augmented batches. The non-augmented ones use logarithmic scale for number of jobs to better emphasize that only a small number jobs delayed the completion of the batch.
Figure 3.20: Execution of cloud augmented batches 1, 3, 4 and 5 and non-augmented batches 3 and 5.
3.3. Improving batch completion times with redundancy adaptation

Volunteer computing provides access to a considerable amount of computing power; however reliability is always a problem when using volatile volunteer resources. Batch makespans are greatly affected even by a single faulty (or slow) resource. This is mitigated by increasing the redundancy (simultaneous copies) of jobs in the system so that eventually one copy is finished. Machine availability in wide-area distributed computing follow the Weibull or Pareto distributions [127], which are power law probability distributions, thus resulting in the so called “long tail effect” for batch completion times. In case of BOINC it provides mechanisms for increasing the redundancy of a job (workunit) in the system. However this can be performed only internally. The user has no control over this as the system is effectively a black box for them. A possible solution is to submit additional copies of the same job to the VC system. In this section our approach for this “black box” scenario is detailed. The middleware is considered as a black box, namely we have (a) no influence on the work unit settings (e.g., replication factor, required quorum), (b) no influence on the scheduling policies. However, we assume that (c) the middleware is not overcommitted and (d) its background load does not cause interference (or can be considered constant) for our application. We also do not handle failures explicitly, we assume that (e) in the first place the middleware is responsible for failure tolerance and (f) there is no permanent failure in the system and (g) if we still encounter a failed job from the middleware it can be resubmitted safely. We chose the black box approach (see (a), (b) and (c)) since we wanted to make a generic approach independent of the capabilities of the underlying middleware (in this case BOINC). However we acknowledge that in some cases this approach may result in increased load at the middleware, e.g., BOINC uses replication and a resubmission at a higher level will result in a new work unit which is then replicated to multiple instances instead of a single new instance of the previous work unit. Constraints (c) and (d) can be considered heavy restrictions however for example in our case public desktop grid projects usually run a single application which in turn can use all the resources available. In the following a novel algorithm is presented that provides an efficient solution with regard to the constraints discussed.

First let $t$ denote a job, and $B$ a batch consisting of $n$ jobs:

$$B = \{t_1, t_2, t_3 \cdots t_n\}$$

(3.15)

$i_{mj}$ is the $j$-th running instance of the $m$-th job, $r_c$ is a resend constant which defines how many jobs can be resubmitted for a smaller batch, $r_r$ is a relative value which defines the percentage of jobs that can be resubmitted, the function $unfinished(B)$ returns the unfinished $t$-s from a batch $B$, the function $useful(i_{mj})$ returns 1 if $t_m \in unfinished(B)$ and 0 else. All the unfinished jobs are resubmitted if the following criterion is met:

$$\sum_{m=1}^{n} \sum_{j=1}^{c} useful(i_{mj}) < max(r_c, r_r * |B|)$$

(3.16)

If at all resubmissions all unfinished jobs are resubmitted it can be rewritten: let $c$
denote the number of times jobs have been resubmitted:

\[
\sum_{m=1}^{n} \sum_{j=1}^{c} useful(i_{mj}) = (c + 1) \cdot |unfinished(B)| \tag{3.17}
\]

and with that the new criterion for resubmission is:

\[
(c + 1) \cdot |unfinished(B)| < max(r_c, r_r \cdot |B|) \tag{3.18}
\]

For our experiments we chose empirically the following constants:

\[
r_c = 100 \tag{3.19}
\]
\[
r_r = 0.1 \tag{3.20}
\]

These parameters mean that batches with less than 1000 jobs are resubmitted after having less than 100 unfinished ones running. For larger batches this occurs after having fewer potentially useful running jobs than 10 percent of the size of the batch. This 90%-10% ratio is commonly used in the literature [11]. The algorithm is summarized as Algorithm 3.6.

**Algorithm 3.6 Adapt redundancy resubmission algorithm**

**Requires:** \( c, r_c, r_r, B = \{t_1, t_2, t_3 \cdots t_n\} \)

**Returns:** None

\[
\text{if } (c + 1) \cdot |unfinished(B)| < max(r_c, r_r \cdot |B|) \text{ then}
\]

\[
\text{submit}(unfinished(B))
\]

\[
\text{end if}
\]

### 3.3.3.1 Evaluation

The algorithm was implemented and evaluated as part of the Wikipedia pre-processing for KOPI Cross-Language Plagiarism Search on SZTAKI Desktop Grid. The scenario including the architecture is described in Appendix A.7.

Until June 2008 static HTML dumps from all Wikipedia wikis were available from Wikimedia Foundation [22], but this project has discontinued since then. As these text versions can be used for several other purposes as well, we decided to share them and make them available for everybody [23]. Currently the English (5.5 GiB), German (2.1 GiB), French (1.4 GiB) and Hungarian (311 MiB) versions can be downloaded, other languages will follow shortly.

The KOPI application is deployed and running on SZTAKI Desktop Grid permanently. SZDG is an umbrella project, but donors can select which application(s) they want to run. This leads to a great number of donors who support KOPI exclusively. KOPI work units have higher priority set on the server, so if volunteers allow multiple applications from SZDG, still KOPI will be processed first. These steps ensure that there is computing capacity available for the KOPI application (see criterion (c) and (d) in Section 3.3.2) whenever required. For evaluation we included the measurements of six representative
3.3. Improving batch completion times

Wikipedia batch conversions: two English, one French, one German and two Hungarian batches.

Table 3.5 shows the results, namely (i) the number of jobs in the batch; (ii) the total number of jobs executed (including resubmissions) for the batch; (iii) the mean round trip time (RTT) for the initial submission of the batch; (iv) the standard deviation of the RTT’s; (v) the mean and (vi) standard deviation for all jobs; finally (vii) the mean and (viii) standard deviation for the “useful” jobs. The dates given in Table 1 denote the date of the Wikipedia dump used by the conversion and not the date when experiments started, as all experiments were executed independently. For discussion we group the six conversions (batches) into three groups based on their number of jobs: (1) the English ones, (2) the French and German ones and (3) the Hungarian conversions.

<table>
<thead>
<tr>
<th>Wikipedia dump</th>
<th>Job Count</th>
<th>First submission runtime (s)</th>
<th>All jobs runtime (s)</th>
<th>Usefull jobs runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>batch</td>
<td>total</td>
<td>µ</td>
<td>σ</td>
</tr>
<tr>
<td>English</td>
<td>3,326</td>
<td>4,277</td>
<td>120,464</td>
<td>157,630</td>
</tr>
<tr>
<td>2011-11-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>English</td>
<td>3,348</td>
<td>4,189</td>
<td>151,052</td>
<td>182,702</td>
</tr>
<tr>
<td>2011-02-12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>French</td>
<td>823</td>
<td>1,093</td>
<td>67,114</td>
<td>104,129</td>
</tr>
<tr>
<td>2012-01-17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>German</td>
<td>946</td>
<td>1,209</td>
<td>96,231</td>
<td>100,432</td>
</tr>
<tr>
<td>2012-01-17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hungarian</td>
<td>162</td>
<td>409</td>
<td>62,483</td>
<td>91,627</td>
</tr>
<tr>
<td>2012-01-15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.5: Statistics for round trip times of batches of jobs from Wikipedia dumps

For the first group our algorithm introduced 28.5% (951 jobs) and 25.1% (841 jobs) overheads respectively for the total job numbers (including resubmissions), while resulting in 36.29% and 37.28% improvements in mean round trip times and 66.18% and 61.9% in standard deviation considering the useful jobs and first submission round trip times. For the second group the algorithm caused 33.2% (270 jobs) overhead for the French and 27.8% (263 jobs) overhead for the German conversion in terms of job numbers. However the mean round trip times were improved by 38.63% and 28.06% while the standard deviations reduced by 79.78% and 78.22%. The algorithm caused the largest overhead for the last group (for the Hungarian conversions) in terms of percent, namely 152.47% and 135.19%, however these mean only 247 and 219 additional jobs. In this case the mean round trip time was improved by 66.73% and 64.18%, while the standard deviation by 86.31% and 87.74%. This algorithm proved to be effective, it only resulted in additional 25-28% of jobs at the largest, the English Wikipedia.

As can be seen in Table 3.5 smaller languages like Hungarian (which has around 20 times less content and thus fewer jobs in the conversion than the English) results in relatively the highest resend rate (152.4%), but even so in numbers it is less than for the English Wikipedia (245 vs. 841 and 219 vs. 951). The overall resend rate for the 4 languages is 31%. By resending all the unfinished jobs regularly the probability that all
copies of a given one will fail decreases exponentially (considering criteria (e), (f) and (g) in Section 3.3.2). This algorithm calculates the time of the resubmission based on the number of returned jobs. As our results show, this is more efficient for larger batches, where the resend rate is around 25-38%. As the size of the batch decreases the resend rate (in percentage) grows.

Figure 3.21 show the time lapse (left side) and jobs in round trip time order (right side) for three selected representative batches from the previous six. The time lapse measurements are displayed up to 4.00e+05 seconds (4.6 days) from start. Similarly round trip time measurements are limited in time, but show times up to 8.00e+05 seconds (9.25 days) and any job reaching this threshold can be considered as unfinished and highly responsible for the “tail effect”. Each chart shows the details of (i) the initial submission, (ii) total jobs and (iii) the useful jobs, which can be considered as three individual batches with own time lapse and round trip measurements on the same chart. The vertical and horizontal dashed lines on the time lapse charts represent the time and number of initial jobs when the first resubmission occurred.

We can see that the time lapses on the three charts have similar characteristics and the round trip time measurements as well. We can see that the number of initial and total jobs is the same obviously until the first resubmission; however the difference between the initial and useful jobs shows from the beginning that some jobs were considered from the first resubmission. Also the first size difference of the job number increase at the first resubmission (denoted with dashed horizontal and vertical lines) shows that a single resubmission was not enough, not all new jobs are considered useful. Similarly the round trip measurements show that there are jobs from the initial submission which are considered unfinished and resubmitted instances were useful jobs. We can also observe that although some jobs have lower round trip times they were still not considered as useful jobs, e.g., the chart of the French conversion shows this: solid line (total jobs) deviates upwards from the dashed line (total). This means that a job was resubmitted, but a previous running instance finished before the new one. In such case our algorithm currently does not cancel the remaining instances, so they remain running.

If we look at the useful “batch” completion times we see that the English conversion finished after 3.1e+05 seconds, the French one after 1.76e+05 seconds while the German one after 1.97e+05 seconds. However, if we look at the initial batch we can see that in each case jobs were running at the 8e+05 cutoff thresholds of the measurements. Considering this threshold we can state that the English conversion took 61.25% less time with 25.1% (841 jobs) overhead, the German conversion took 75.38% less time with 27.8% (263 jobs) overhead and the French conversion took 78.01% less time with 33.2% (270 jobs) overhead total.
3.3. Improving batch completion times

Figure 3.21: Typical time lapse for a batch (a,c,e) and its jobs in round trip time order (b,d,f) for an English, French and German Wikipedia dataset.
The work presented in this thesis aims to improve desktop grid and volunteer computing (DG/VC) from different aspects.

The initial steps aim to present an informal comparison of desktop grids and volunteer computing. After this four formal models are presented using the Abstract State Machines (ASM) method. The models are based on a formal model for grids by Nemeth et al. [1] and Kertesz et al. [2]. The first model, $M_{GROUND-DG}$, is a formal model of desktop grids. The second and third, $M_{VC-VOTE}$ and $M_{VC-SPOT}$ respectively, are formal models for volunteer computing, while $M_{BOINC}$ is a model of the Berkley Open Infrastructure for Network Computing (BOINC). For each model conditions are formalised by using ASM in order to give a precise definition. Each model is validated against the conditions to show that models correctly represent them. Also for each model the model itself is discussed and different middleware is evaluated against the model. Also for each model their correspondence with the previous ones is elaborated.

The second half of the thesis discusses three aspects and future directions of DG/VC systems.

First a method is described for federating DG/VC systems. Based on the method a new model ($M_{FED-BOINC}$) is introduced and three scenarios are presented, namely (1.) hierarchy of desktop grids; (2.) DG supporting a volunteer project; and (3.) volunteer projects collaborating. From these scenarios the first one is detailed with the required extensions for application deployment in the inter-connected DG/VC system.

Second a definition for Volunteer Cloud Computing is introduced and the related service models and characteristics are discussed. Four abstraction frameworks are introduced ($Cross-middleware; middleware, environment and full abstraction$) and evaluated that support these.

Finally three algorithms ($Adapt redundancy, Prioritize delayed jobs, Prioritize job sources$) are presented and each is evaluated against real world scenario. Two scenarios are presented representing a different volunteer computing project (SZTAKI Desktop Grid and EDGeS@home) one with a single application (KOPI) and another as an “umbrella” project running multiple applications.

4.1 Applicability of the results

The formal models developed for DG/VC systems allow – similarly to the formal model for BOINC – modeling other systems and categorizing existing ones. Based on the evaluation along the guidelines of the models and the categorization recommendations can be formulated for the usability of a particular middleware for a given use case.
Zoltán Farkas evaluated several scheduling algorithms for the hierarchical desktop grid as part of his Ph.D. thesis [89]. These were based on the federation method (including the security and application deployment method) I developed and discussed in Section 3.1.2.

Based on the abstraction frameworks I defined for volunteer computing (see Section 3.2.3), I implemented three tools available as open source projects: DC-API, GenWrapper\(^1\) and GBAC\(^2\). DC-API is going to complement the native API of BOINC and is currently in the process of merging with the BOINC source code base. GBAC is going to be part of the VBoxWrapper\(^3\) solution of BOINC and I am in the process of merging. To my present knowledge there are 58 scientific applications ported or developed with DC-API, GBAC or GenWrapper. The list of applications and the methodology used for porting or development can be found in Appendix A.1. I would like to specifically mention an application from the list (number 55.) that was ported using GenWrapper: “patterncount”. It is a fundamental part of a Ph.D. thesis accepted at Brunel University of London [128]. Further my results were/ are used by the following EU and Hungarian R&D projects: EDGeS\(^4\), EDGI\(^5\), DEGISCO\(^6\), CancerGrid\(^7\), IDGF-SP\(^8\) and WEB2GRID\(^9\). These projects ported or developed numerous applications and supported the research and development behind DC-API, GenWrapper and GBAC.

Implementations of the algorithms presented in Sections 3.3.2 and 3.3.3 are deployed in production environments at SZTAKI Desktop Grid [77] and EDGeS@home [73]. However EDGeS@home was closed down at the end of 2015 due to lack of funding.

### 4.2 Future work

The formal models could be further validated by creating additional models of existing middleware and compare them against each other. This would also provide the possibility to evaluate the models of existing middleware against \(M_{\text{GROUND-DG}}\), \(M_{\text{V C-VOTE}}\) or \(M_{\text{V C-SPOT}}\) to see whether they are desktop grids or volunteer computing systems. \(M_{\text{BOINC}}\) could be further extended to include the abstraction frameworks and the service models, and characteristics defined in Section 3.2.2.

For federating Volunteer Computing investigating and paraphrasing scenarios 2-3 remains future work, also based on the work of Zoltán Farkas [89] further scheduling algorithms could be proposed and evaluated both for the hierarchical desktop grid (scenario 1) and scenarios 2-3 as well.

Volunteer clouds could be investigated from a different perspective, namely from what additional functionality would be required for current cloud middleware to be able to use volunteer, non-dedicated and volatile resources. The two approaches could be compared

---

\(^1\)http://genwrapper.sourceforge.net, access date: 2014-05-26
\(^2\)http://gbac.sourceforge.net, access date: 2014-05-26
\(^3\)http://boinc.berkeley.edu/trac/wiki/VboxApps, access date: 2014-10-09
\(^4\)http://edges-grid.eu/, access date: 2014-01-01
\(^5\)http://edgi-project.eu/, access date: 2014-01-01
\(^6\)http://degisco.eu/, access date: 2014-01-01
\(^7\)http://cancergrid.eu/, access date: 2009-01-01
\(^8\)http://idgf-sp.eu/, access date: 2014-05-26
and evaluated against each other. As for GBAC the development was discussed with the BOINC team in order to avoid redundant and parallel research and development. Some parts of its implementation relies on VBoxWrapper and the modifications are being merged with the BOINC version now. Further instead of using multiple layered VAs for instance and application layers in GBAC, more lightweight and generic solutions should be considered like a Linux cgroups (Control Groups) \cite{129} based approach. For example Docker \cite{130} provides containers based on cgroups (and also has its own implementation). These allow isolating the application from the rest of the VM and bundling all dependencies (i.e., libraries and binaries) into a single unit – a container – that can be deployed and run on top of virtually any VM.

The algorithms presented in Sections 3.3.2 and 3.3.3 could be further evaluated via simulation (with their complete parameter space) using traces from either The Grid Workloads Archive\footnote{http://gwa.ewi.tudelft.nl/, access date: 2016-06-10} or from other DG/VC deployments.
Appendix A

A.1 Scientific applications using DC-API, GenWrapper or GBAC

GBAC, GenWrapper and DC-API are tools based on the abstraction frameworks presented in section 3.2.3. Tables A.1-A.4 list all scientific applications developed or ported using these tools.

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Institute</th>
<th>Country</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Athena</td>
<td>University of Copenhagen - Niels Bohr Institute</td>
<td>Denmark</td>
<td>GBAC</td>
<td>Virtual Machine and GBAC based Atlas analysis package.</td>
</tr>
<tr>
<td>2</td>
<td>Autodock</td>
<td>University of Westminster - CPC</td>
<td>UK</td>
<td>GW</td>
<td>Autodock [131] is a suite of automated docking tools. It is designed to predict how small molecules, such as substrates or drug candidates, bind to a receptor of known 3D structure.</td>
</tr>
<tr>
<td>3</td>
<td>Autodock3</td>
<td>University of Westminster - CPC</td>
<td>UK</td>
<td>GW</td>
<td>Autodock3 is a suite of automated docking tools version 3. It is designed to predict how small molecules, such as substrates or drug candidates, bind to a receptor of known 3D structure [131, 132].</td>
</tr>
<tr>
<td>4</td>
<td>Biome-BGC</td>
<td>MTA Centre for Ecological Research</td>
<td>Hungary</td>
<td>GW</td>
<td>The Biome-BGC model [133, 134] estimates the ecosystem scale storage and fluxes of energy, carbon, nitrogen and water, controlled by various physical and biological processes on a daily time-scale.</td>
</tr>
<tr>
<td>5</td>
<td>Biome-BGC MuSo</td>
<td>ELTE / MTA Centre for Ecological Research</td>
<td>Hungary</td>
<td>GW</td>
<td>Biome-BGC MuSo 1.2.2 [133, 135] (MuSo is the abbreviation of multilayer soil module) is an extended version of Biome-BGC 4.1.1 ecosystem modeling software.</td>
</tr>
<tr>
<td>6</td>
<td>BNBGrid</td>
<td>Institute of System Analysis - Russian Academy of Sciences</td>
<td>Russia</td>
<td>GW</td>
<td>The global optimization framework BNB-Grid is aimed at solving hard combinatorial, discrete and global optimization problems in a distributed heterogeneous computing environment.</td>
</tr>
<tr>
<td>7</td>
<td>Blender</td>
<td>University of Westminster - CPC</td>
<td>UK</td>
<td>GW</td>
<td>Blender [136] is an integrated application that enables the creation of a broad range of 2D and 3D content. Blender provides a broad spectrum of modeling, texturing, lighting, animation and video post-processing functionality in one package.</td>
</tr>
<tr>
<td>8</td>
<td>CNS</td>
<td>University of Westminster - CPC</td>
<td>UK</td>
<td>GBAC</td>
<td>Crystallography &amp; NMR System (CNS) is the result of an international collaborative effort among several research groups. The application provides a flexible multi-level hierarchical approach for the most commonly used algorithms in macromolecular structure determination.</td>
</tr>
<tr>
<td>9</td>
<td>CPDynG</td>
<td>G. V. Kurdymov Institute for Metal Physics</td>
<td>Ukraine</td>
<td>DC-API</td>
<td>CPDynSG processes experimental data and theoretical scenarios on the evolution of urban population and compares with simulations by kinetic Monte Carlo method several general scenarios of evolution of cities (municipalities, lands, counties, etc.) those should be explained by migration, merges, population growth, etc. with many initial designed and actual configurations.</td>
</tr>
<tr>
<td>10</td>
<td>Fusion (IS-DEP)</td>
<td>MTA SZTAKI LPDS</td>
<td>Hungary</td>
<td>GW</td>
<td>ISDEP [101] is a fusion plasma application programmed and highly optimized in C language which calculates the trajectories of the particles inside a fusion device in the presence of magnetic and electric fields and collisions with a background plasma of a given temperature and density.</td>
</tr>
<tr>
<td>11</td>
<td>GATE</td>
<td>CREATIS INSA Lyon</td>
<td>France</td>
<td>GW</td>
<td>GATE [137] is advanced open source software developed by the international OpenGATE collaboration and dedicated to numerical simulations in medical imaging and radiotherapy. It currently supports simulations of Emission Tomography (Positron Emission Tomography - PET and Single Photon Emission Computed Tomography - SPECT), Computed Tomography (CT) and Radiotherapy experiments.</td>
</tr>
</tbody>
</table>

Table A.1: Applications 1-11. using GBAC, GenWrapper (GW) and DC-API
<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Institute</th>
<th>Country</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>KOPI</td>
<td>MTA SZTAKI DSD / MTA SZTAKI LPDS</td>
<td>Hungary</td>
<td>GW</td>
<td>Pre-processor for the KOPI Multilingual plagiarism search portal.</td>
</tr>
<tr>
<td>13</td>
<td>LAM-MPSolver-DCI</td>
<td>G. V. Kurdymov Institute for Metal Physics</td>
<td>Ukraine</td>
<td>DC-API</td>
<td>The very popular non-commercial open-source package LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) by Sandia Labs (<a href="http://lammps.sandia.gov">http://lammps.sandia.gov</a>)</td>
</tr>
<tr>
<td>14</td>
<td>NetScape</td>
<td>University of Copenhagen - Niels Bohr Institute</td>
<td>Denmark</td>
<td>GBAC</td>
<td>Virtual Machine GBAC based fast neutron simulator application.</td>
</tr>
<tr>
<td>15</td>
<td>Multiscaler-VideoP</td>
<td>G. V. Kurdymov Institute for Metal Physics</td>
<td>Ukraine</td>
<td>DC-API</td>
<td>MultiscalerVideoP [138] is a suite of MATLAB image and video processing routines and functions that widely used in material science and other fields of science where thorough massive analysis of images (filtering, feature detection, edge selection, statistics, etc.) is necessary on many scales.</td>
</tr>
<tr>
<td>16</td>
<td>NETMAX</td>
<td>Institute of System Analysis - Russian Academy of Sciences</td>
<td>Russia</td>
<td>GW</td>
<td>The goal of the NETMAX application is to experimentally compare the performance of different network flow routing algorithms. This numerical comparison allows one to find the most efficient method and to identify the best set of algorithm parameters.</td>
</tr>
<tr>
<td>17</td>
<td>PDSAT</td>
<td>Institute of System Analysis - Russian Academy of Sciences</td>
<td>Russia</td>
<td>GW</td>
<td>Parallel and Distributed SAT solver. The application is the distributed SAT solver able to decompose an original SAT problem to lower dimension instances with consequent solving of these problems on desktop grid systems.</td>
</tr>
<tr>
<td>18</td>
<td>Patient-Readmission</td>
<td>University of Westminster - CPC</td>
<td>UK</td>
<td>GW</td>
<td>The Patient Readmission Application is a statistical model developed in R. Individual hospital performance could be measured through the propensity of patient readmissions. The national (England) hospital episodes statistics dataset comprises more than 5 million patient readmissions between Jan 1998 and Dec 2003.</td>
</tr>
<tr>
<td>19</td>
<td>Public-AutoDock423</td>
<td>University of Westminster - CPC</td>
<td>UK</td>
<td>GW</td>
<td>AutoDock is a suite of automated docking tools. It is designed to predict how small molecules, such as substrates or drug candidates, bind to a receptor of known 3D structure.</td>
</tr>
<tr>
<td>20</td>
<td>Public-AutoDock423</td>
<td>University of Westminster - CPC</td>
<td>UK</td>
<td>GW</td>
<td>AutoDock is a suite of automated docking tools. It is designed to predict how small molecules, such as substrates or drug candidates, bind to a receptor of known 3D structure.</td>
</tr>
<tr>
<td>21</td>
<td>Public-AutoDockVina112</td>
<td>University of Westminster - CPC</td>
<td>UK</td>
<td>GW</td>
<td>AutoDock Vina is an open-source program for drug discovery, molecular docking and virtual screening, offering multi-core capability, high performance and enhanced accuracy and ease of use.</td>
</tr>
<tr>
<td>22</td>
<td>Root</td>
<td>University of Copenhagen - Niels Bohr Institute</td>
<td>Denmark</td>
<td>GBAC</td>
<td>Virtual Machine GBAC based physics simulation tool.</td>
</tr>
<tr>
<td>23</td>
<td>SIMAP</td>
<td>University of Westminster - CPC</td>
<td>UK</td>
<td>GW</td>
<td>Systems biology is computational modelling, which is the process of constructing and simulating an abstract model of a biochemical system for subsequent analysis. The SIMAP Utility is a platform-independent environment for modelling biochemical networks, and also for simulating and analysing the dynamic behaviour of biochemical models.</td>
</tr>
<tr>
<td>24</td>
<td>UC-Explorer</td>
<td>KFKI / MTA SZTAKI LPDS</td>
<td>Hungary</td>
<td>DC-API</td>
<td>Universality classes occur very frequently in complex system exhibiting many degrees of freedom. When the correlations diverge, for example near a critical point, when fluctuations dominate (like in case of an financial/economic collapse or a changing climate) the microscopic details (interactions) become irrelevant. UC-Explorer is an application to model these phenomena [35].</td>
</tr>
<tr>
<td>25</td>
<td>Video-Converter</td>
<td>University of Portsmouth / University of Westminster - CPC</td>
<td>UK</td>
<td>GW</td>
<td>VideoConverter is an open-source program using the static version of ffmpeg allowing to convert video into the format wanted.</td>
</tr>
<tr>
<td>26</td>
<td>VideoMerger</td>
<td>University of Portsmouth / University of Westminster - CPC</td>
<td>UK</td>
<td>GW</td>
<td>VideoMerger is an open-source program using the static version of ffmpeg allowing to merge/videos together.</td>
</tr>
<tr>
<td>27</td>
<td>VideoSplitter</td>
<td>University of Portsmouth / University of Westminster - CPC</td>
<td>UK</td>
<td>GW</td>
<td>VideoSplitter is an open-source program using the static version of ffmpeg, allowing to split a video into pieces depending on the duration of the video.</td>
</tr>
<tr>
<td>28</td>
<td>VisIVOspace-Mission</td>
<td>INAF - Catania Astrophysical Observatory</td>
<td>Italy</td>
<td>GW</td>
<td>Modern visualization can aid astrophysicists in gaining good insights of highly complex datasets, through rapid and intuitive discovery of familiar patterns and correlations between properties, without involving CPU intensive analysis codes. As a result, suitably constructed visualization tools can be instrumental for future astrophysical advances, e.g. by allowing comparing simulations meaningfully or even correlating appropriately simulations and observational datasets.</td>
</tr>
</tbody>
</table>

Table A.2: Applications 12-28. using GBAC, GenWrapper (GW) and DC-API.
### A.1. Scientific applications using DC-API, GenWrapper or GBAC

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Institute</th>
<th>Country</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>VisIVO</td>
<td>INAF - Catania Astrophysical Observatory</td>
<td>Italy</td>
<td>GW</td>
<td>VisIVO is a suite of software tools for creating customized views of 3D renderings from astrophysical data tables. Their defining characteristic is that no fixed limits are prescribed regarding the dimensionality of data tables input for processing thus supporting very large scales datasets.</td>
</tr>
<tr>
<td>30</td>
<td>Audiveris</td>
<td>MTA SZTAKI</td>
<td>Hungary</td>
<td>GW</td>
<td>Audiveris is an open-source Optical Music Recognition software which processes the image of a music sheet to automatically provide symbolic music information in MusicXML standard. Audiveris is developed in Java and invokes Google Tesseract OCR (C++) for text recognition.</td>
</tr>
<tr>
<td>31</td>
<td>BinSys</td>
<td>ELTE/ MTA SZTAKI LPDS</td>
<td>Hungary</td>
<td>DC-API</td>
<td>BinSys[32] aims at finding many generalized binary number systems. An extensive search is performed in the finite set of matrices of given size fulfilling some necessary conditions. The program outputs a list of matrices (being more precise characteristic polynomials) that are already likely to be number system bases. Knowing all matrices up to a given dimension could help to a deeper understanding of the mathematics of generalized number systems.</td>
</tr>
<tr>
<td>32</td>
<td>BWA</td>
<td>University of Westminster CPC</td>
<td>UK</td>
<td>GW</td>
<td>Burrows-Wheeler Aligner (BWA) is an efficient program that aligns relatively short nucleotide sequences against a long reference sequence such as the human genome. It implements two algorithms, bwa-short and BWA-SW.</td>
</tr>
<tr>
<td>33</td>
<td>Classification</td>
<td>University of Westminster CPC</td>
<td>UK</td>
<td>GW</td>
<td>The application implements a so-called double-loop cross-validation protocol for the estimation and evaluation of prognostic classifiers for omics data. The application can be used with a variety of classification models (DLDA, neural networks, support vector machines (SVM), decision trees, PAM, PCDA, logistic regression, random forests etc) and is easy to extend with new classification models.</td>
</tr>
<tr>
<td>34</td>
<td>CORSIKA</td>
<td>University of Westminster CPC</td>
<td>UK</td>
<td>GW</td>
<td>CORSIKA (Cosmic Ray Simulation for Karlsruhe Shower Core and Array Detector) is a Monte Carlo program to study the evolution and properties of extensive air showers in the atmosphere. It allows simulation of behaviour of elementary particles up to energies $10^{20}$ eV. Currently in use by more than 50 experiments worldwide.</td>
</tr>
<tr>
<td>35</td>
<td>DIRAC</td>
<td>MTA SZTAKI</td>
<td>Hungary</td>
<td>GBAC</td>
<td>DIRAC (Distributed Infrastructure with Remote Agent Control) INTERWARE[139] is a software framework for distributed computing providing a complete solution to one (or more) user community requiring access to distributed resources. DIRAC builds a layer between the users and the resources offering a common interface to a number of heterogeneous providers, integrating them in a seamless manner, providing interoperability, at the same time as being optimized, transparent and reliable usage of the resources.</td>
</tr>
<tr>
<td>36</td>
<td>DSP</td>
<td>University of Westminster CPC</td>
<td>UK</td>
<td>DC-API</td>
<td>The main purpose of the Digital Signal Processing (DSP) Application is to design periodic nonuniform sampling sequences for digital alias free signal processing. Digital alias-free signal processing (DASP) is an approach that offers effective solutions to processing signals with conservatively estimated spectral support. This application produces periodic nonuniform sampling (PNS) sequences for DASP applications.</td>
</tr>
<tr>
<td>37</td>
<td>EMMIL</td>
<td>International Business School</td>
<td>Hungary</td>
<td>GW</td>
<td>The e-marketplace model called EMMIL (E-Marketplace Model Integrated with Logistics) models a single buyer looking for a set of tangible products in specified quantities, many suppliers offering certain quantities of the required goods and many third party logistics providers (3PLs) that undertake transportation. The marketplace operates a composite reverse auction with discrete rounds of sealed bidding that alternate between sellers and 3PLs.</td>
</tr>
<tr>
<td>38</td>
<td>GBAC</td>
<td>MTA SZTAKI</td>
<td>Hungary</td>
<td>DC-API</td>
<td>The Generic BOINC Application Client (GBAC)[31] is a virtualization (VirtualBox) based wrapper. Beyond its name it aims to be a generic framework providing virtualized environments for various distributed computing infrastructures. This package is a multi-application container. It can be run with any application submitted (as a job).</td>
</tr>
<tr>
<td>39</td>
<td>gengromacs</td>
<td></td>
<td>GW</td>
<td></td>
<td>GROMACS is a versatile package to perform molecular dynamics, i.e. simulate the Newtonian equations of motion for systems with hundreds to millions of particles.</td>
</tr>
<tr>
<td>40</td>
<td>gromacs</td>
<td></td>
<td>GW</td>
<td></td>
<td>GROMACS is a versatile package to perform molecular dynamics, i.e. simulate the Newtonian equations of motion for systems with hundreds to millions of particles.</td>
</tr>
<tr>
<td>41</td>
<td>laserac</td>
<td></td>
<td>DC-API</td>
<td></td>
<td>The main purpose of the CALD Application is to simulate the dynamics of laser devices using Cellular Automata-based discrete model.</td>
</tr>
<tr>
<td>42</td>
<td>mdrungbac</td>
<td>Johannes Gutenberg University Mainz</td>
<td>Germany</td>
<td>GBAC</td>
<td>MDRun (Gromacs) via GBAC. Gromacs is an open source molecular dynamics package primarily designed for biomolecular systems such as proteins and lipids.</td>
</tr>
<tr>
<td>43</td>
<td>mentalray</td>
<td>University of Westminster</td>
<td>UK</td>
<td>GW</td>
<td>Autodesk Maya 3D animation software delivers an end-to-end creative workflow with comprehensive tools for animation, modeling, simulation, visual effects, rendering, matchmoving, and composting on a highly extensible production platform.</td>
</tr>
</tbody>
</table>

Table A.3: Applications 29-43. using GBAC, GenWrapper (GW) and DC-API
<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Institute</th>
<th>Country</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>merlin</td>
<td>University of Westminster - CPC</td>
<td>UK</td>
<td>GW</td>
<td>The Merlin software carries out single-point and multipoint analyses of pedigree data, including IBD and kinship calculations, non-parametric and variance component linkage analyses, error detection and information content mapping.</td>
</tr>
<tr>
<td>45</td>
<td>simul8NBS</td>
<td>University of Brunel; Simul8</td>
<td>UK</td>
<td>GW</td>
<td>The application run Simul8 preinstalled model. Simul8NBS is simulation package for discrete event simulation which use Microsoft Excel to store temporary results.</td>
</tr>
<tr>
<td>46</td>
<td>simul8wE</td>
<td>University of Brunel; Simul8</td>
<td>UK</td>
<td>GW</td>
<td>Discrete event simulation with simul8 software.</td>
</tr>
<tr>
<td>47</td>
<td>singlechainmc-32000gbac</td>
<td>Johannes Gutenberg University Mainz</td>
<td>Germany</td>
<td>GBAC</td>
<td>The Singlechainmc32000 application simulates a globular polymer configuration of a simplified model for polyethylene that can for instance be used to examine statistics of knots and other topological features of globular polymers. The chain consists of 32000 monomers, which roughly represent three CH2 groups each.</td>
</tr>
<tr>
<td>48</td>
<td>singlerenderer-gbac</td>
<td>Johannes Gutenberg University Mainz</td>
<td>Germany</td>
<td>GBAC</td>
<td>In this application we analyze what speed-up can be achieved if a rendering process of a three-dimensional scene is distributed among many computation nodes. The rendering itself consists of different sequential phases each processed in a separate stage. Nevertheless, the image generation can be distributed due to the fact that an image can be subdivided into many independent sub-images. For our analysis we compare different system environments: a system with single standard CPU, Intel’s Single-Chip-Cloud (SCC) and a desktop grid environment using GBAC.</td>
</tr>
<tr>
<td>49</td>
<td>slinca</td>
<td>G. V. Kurdyumov Institute for Metal Physics</td>
<td>Ukraine</td>
<td>DC-API</td>
<td>SLinCA application simulates (by kinetic Monte Carlo method) several general scenarios of agent aggregation in clusters with many initial configurations of agents (random, regular, etc.), different kinetics law (arbitrary, diffusive, etc.), various interaction laws (arbitrary, elastic, non-elastic, etc.).</td>
</tr>
<tr>
<td>50</td>
<td>tcg</td>
<td>University of Szeged</td>
<td>Hungary</td>
<td>GW</td>
<td>The Conformer Generator is a biochemical application that generates conformers of flexible molecules by unconstrained molecular dynamics at high temperature to overcome conformational bias then finishes each conformer by simulated annealing and energy minimization to obtain reliable structures. These structures were successfully used to obtain the active conformation of the peptide for its receptor (mu opioid receptor) by QSAR modelling assisted with an efficient variable selection algorithm.</td>
</tr>
<tr>
<td>51</td>
<td>vina</td>
<td></td>
<td>Hungary</td>
<td>GW</td>
<td>AutoDock Vina is a new open-source program for drug discovery, molecular docking and virtual screening, offering multi-core capability, high performance and enhanced accuracy and ease of use.</td>
</tr>
<tr>
<td>52</td>
<td>wilson</td>
<td></td>
<td>Hungary</td>
<td>GW</td>
<td>Wilson application: searching wilson primes.</td>
</tr>
<tr>
<td>53</td>
<td>x-ray</td>
<td></td>
<td>Hungary</td>
<td>GW</td>
<td>X-ray diffraction can be measured and represented in a graph where the minimum and maximum intensities of the rays are shown. The particle information is obtained through parameters that originated the peaks in that graph. But in this graph some peaks can be severely overlapped. The problem is to define which parameters originated these peaks and how many peaks exist in the graph.</td>
</tr>
<tr>
<td>54</td>
<td>zeta</td>
<td>ELTE</td>
<td>Hungary</td>
<td>DC-API</td>
<td>Hennemann zeta research project. The aim of this project is locating as many large values of the zeta function as possible on the critical line for statistical analysis. Several thousands of values have been found and many records were achieved by applying some results on simultaneous Diophantine approximations.</td>
</tr>
<tr>
<td>55</td>
<td>Patterncount</td>
<td>Brunel University</td>
<td>UK</td>
<td>GW</td>
<td>Mohammadmersad Ghorbani: Computational analysis of CpG site DNA methylation, PhD thesis [128].</td>
</tr>
<tr>
<td>56</td>
<td>Pannon Optimizer</td>
<td>University of Pannonia</td>
<td>Hungary</td>
<td>GBAC</td>
<td>Parameter sweep of linear programming solvers. A new simplex algorithm based linear programming solver.</td>
</tr>
<tr>
<td>57</td>
<td>Spatial Biome-BGC 4.1.1 MPI</td>
<td>MTA Centre for Ecological Research</td>
<td>Hungary</td>
<td>GW</td>
<td>The goal of BioVeL - Biome-BGC ecosystem modelling is the estimation of behaviour of terrestrial ecological systems (forests, grasslands, agricultural areas) as accurately as possible.</td>
</tr>
<tr>
<td>58</td>
<td>Spatial Biome-BGC 4.1.1 MPI</td>
<td>MTA Centre for Ecological Research</td>
<td>Hungary</td>
<td>GW</td>
<td>Improved Biome-BGC to improve the ability of the model to simulate carbon and water cycle in managed herbaceous ecosystems.</td>
</tr>
<tr>
<td>59</td>
<td>electrostatic field simulations</td>
<td>Warsaw University of Technolog-</td>
<td>Poland</td>
<td>GBAC</td>
<td>Michal Rybinski: Use of SZTAKI Desktop grid solution for parallel computations during multiple electrostatic field simulations. Thesis work. Applicability of desktop grid computing for running multiple electrostatic field simulations using the Fenics project.</td>
</tr>
</tbody>
</table>

Table A.4: Applications 44-59. using GBAC, GenWrapper (GW) and DC-API.
A.2 Workload transfer for the Hierarchical Desktop Grid

This section gives an overview of the application and work distribution in the Hierarchical Desktop Grid. In our example scenario we use the simplest setup, which consists of just two projects Project A and Project B, one application App and one Donor. The flow of the deployment and distribution process is the following (Figure A.1 depicts steps 1-6., Figure A.2 describes steps 7-12., and Figure A.3 details steps 13-21.):

1. The application developer AppDev may initially sign the App using her secret key.

2. The certificate of the AppDev may be added, if not already done so, to the list of certificates belonging to Project A where the application is about to be installed by the administrator of the project.

   The list of certificates belonging to an entity (server, project, or client) holds all the certificates of the application developers, projects, servers and clients accepted by the entity.

3. The application is installed by the administrator manually. This initial procedure is the same as the normal application install process of BOINC.

4. The Project may also sign the application. This signature may either be appended to the signature of the AppDev or it may replace the original signature if the project does not wish to disclose the origin of the application. This step must be performed manually since the secret key of the Project should not be kept on the same machine where BOINC is running.

5. Workunits are created by the master application and are passed to BOINC.

6. For each workunit the input data (Input) and workunit descriptions (Desc) are signed by the Server 1 (Sig – I, Sig – D).

   At this point the results are ready to be sent to any client attached to the project. Clients may be normal Hosts or Gateway s.

7. The Gateway connects. Server 1 has a list of the certificates of all accepted clients. If the certificate of the Gateway is among them, it can continue to attach to the desired project running on the server. The project has a list of certificates too, containing the certificates of the accepted clients.

8. The Gateway checks for new applications. Each application is tied to a BOINC platform (OS and architecture combination). The Gateway will query for applications tied to each predefined platform. The application binary and the belonging signatures are downloaded.

9. The signatures of the application binary are verified using the client’s list of certificates. Donors have a TrustedID_Donor set defined, but the Gateway delegates the trust to the child project, in this case to Project B. It will accept any application Project B is trusting.
10. A unique name for the application is created, and the Gateway stores the name mapping as described in 3.1.2.1.2. The unique name guarantees that there will be no name collisions in the hierarchy, and the mapping allows the Gateway to update/remove applications at the child project. Project B might add its signature to the application, certifying the path of origin for its children.

At this point the application is deployed at the child project with the unique name. Gateway will continue querying for new applications (checking all available platforms) and repeat this procedure (8-10) until there are no new ones available.

11. The Gateway will now query for work for the applications deployed at Project B. The name mapping is used in this process, since for the same application a different name is set at the child and at the parent. A successful query will fetch a result, which consists of one or more input files, their signatures, and a workunit description (it is the same for each result created from the same workunit) and its signature.
A.2. Workload transfer for the Hierarchical Desktop Grid

The signatures of the input files and workunit descriptions \((\text{Sig} - I, \text{Sig} - D)\) are checked against the \(\text{TrustedID}_{\text{Project}}\) set of the child project.

12. From the result fetched from the parent a workunit is created at the child project by the Gateway. \textit{Server 2} may add its signature to the inputs and descriptions belonging to the newly created workunit. From the workunit one or more results \((\text{WU1} - \text{Result1} - \text{AI})\) are created by the child project.

At this point the application \(<\text{UUID}>\) and a workunit belonging to it is fully deployed at the child project, waiting to be downloaded by a client, which may be a Gateway or a Host. If a Gateway connects, the procedure is the same from step 7, if a Donor (using her Core Client) connects the following steps will be executed.

13. A Donor connects to the server. \textit{Server 2} and \textit{Project B} has all the certificates of the accepted clients pre-installed, meaning they can authenticate her. Afterward her Client queries for new applications belonging to its platform, and downloads their binary and signatures.

14. The signature belonging to the downloaded application is verified that it is by one of the trusted application developers, and if there are additional signatures, they are verified that they are by one of the trusted projects.

15. The Client will now query for work (results) belonging to one of the applications available at the client (the application is chosen by the local scheduling implemented in the Core Client). On success one or more results \((\text{WU1} - \text{Result1} - \text{AI})\) consisting of input files \((\text{Input})\), workunit description \((\text{Desc})\) and their signatures \((\text{Sig} - I, \text{Sig} - D)\)
Sig - D) will be downloaded. The signature(s) of the description and the input files are verified to ensure they are signed by (one of) the trusted servers.

16. The result $WU1 - Result1 - AI$ is ready to be processed by the application. Processing it, will produce one or more output files (Output). The Client signs these files.

17. The output files and signatures (Output, Sig - O) are uploaded to Project B by the Client, and the result is reported as finished.

18. The signatures of the uploaded files are checked if they are created by one of the trusted clients, using the list of certificates of Project B.

19. The Gateway notices that a result belonging to a workunit that was created by it is complete. It fetches the output files from Project B, so it is able to upload it to the parent when needed. It adds its signature to the output(s) of the result (Sig - O - HC). For the parent project the Gateway is the Client processing work, but in reality it is acting as a middle-man relaying work and binaries between the two projects.

20. The Gateway contacts Server 1, Project A, and uploads the output files and their signatures belonging to the result.

21. The Output is verified using the signature and the list of certificates by Project A.

At this point the completed result is available at Project A for validation. Workunit validation is performed only here at the originating project, the child projects use a trivial validator which is part of the Gateway, and it is accepting all incoming results. This may be adequate in a controlled environment, where only the selected clients allowed to return results, but this does not filter out syntactically incorrect results at the lower levels caused for example by some hardware defect.
### A.3 Abstract State Machines

The Abstract State Machine (ASM) is a mathematically well founded framework for high-level system design and analysis [140] originally introduced as evolving algebras by Gurevich [141]. ASM allows to hide easily the non-important details at the high-level design phase by formulating the model on a conceptional level rather than based on implementation details and attributes. Lower detail characteristics can be added to the models later gradually. It is an agent based modelling system where the system is described from the perspective of an agent. In ASM states are represented as algebras, i.e., basic sets (called universes) with functions and relations interpreted on them. A signature (or vocabulary) is a finite set of function names each with fixed arity. It also contains the usual Boolean operators (e.g., $\land$, $\lor$) and the symbols $true$, $false$, = and $undef$. A state $S$ of signature $\gamma$ is a nonempty set $X$ together with interpretations of function names in $\gamma$ on $X$. $X$ is called the superuniverse. A nullary function name is interpreted as an element of $X$ this corresponds to the notion of variables. An $r$-ary function name is interpreted as a function from $X^r \rightarrow X$. A location of $S$ is a pair $l = (f, x)$, where $f$ is a function name of arity $r$ in vocabulary $\gamma$ and $x$ is an $r$-tuple of elements of $X$. The element $f(x)$ is the content of location $l$. An update is a pair $z = (l, y)$, where $l$ is a location and $y$ is another element of $X$. Firing $x$ at state $S$ means putting $y$ into the location $l$ while other locations remain unchanged. The resulting state is $S'$ (the sequel of $S$), thus the interpretation of a function $f$ at argument $x$ has been modified producing a algebra, i.e., a new state. The special nullary $Self$ function is used to represent the agent and also allows to identify itself amongst other agents. It is interpreted differently by different agents. This $Self$ function can never be the subject of updates. ASM models are defined as a set of *transition rules* in the form of:

$$
\text{if } <\text{Condition}> \text{ then } <\text{Updates}> \text{ endif}
$$

where $Updates$ are a finite number of statements in the form of $f(t_1, \cdots, t_n) := t$.

Condition (also called guard) is an arbitrary predicate logic formula without free variables that evaluates to $true$ or $false$. If the condition is evaluated as true then $Updates$ are fired. In some models additional space is required during execution. Therefore the *reserve* of a state is the infinite source from where new elements can be imported to the universe $U$ using the following construct:

$$
\text{extend } U \text{ by } u_1, u_2, \cdots, u_n \text{ with } <\text{Updates}> \text{ endextend}
$$

There are other common notations like where, let, choose, if-then-else and forall available. More details on these can be found in [72]. Applying a step on ASM $M$ (model) to state (structure) $S$ will produce another state $S'$ on the same set of function names. First the set of fireable rules is determined where the condition is evaluated to true. Second all $t_i$ and $t$ are determined that are used in the $f(t_1, \cdots, t_n) := t$ expressions. Finally all these rules are evaluated simultaneously resulting in the sequential state $S'$.

In [142] the concepts of submachines, composition, parametrized machines, recursion, local values and return values are discussed for ASM. The concepts from these used in
this thesis are shown in Listing A.1 through a rule for factorial calculation. The “returned” values of functions are denoted with the reserved 0-ary function \textit{result}. In the following sections I am going to use the \textit{return} 0-ary function instead of \textit{result} since result is used in my models (e.g., \textit{MBOINC}) in another context.

\begin{verbatim}
rule Fac(n)=
  local x := 1
  if (n = 1) then
      result := 1
  else
      (x := Fac(n-1)) seq result := n * x
\end{verbatim}

Listing A.1: Example ASM rule using recursion, local variable, sequential atomic execution and return value.

The ASM ground model serves as a foundation for further detailed models of the same subject, as Boerger [143, p. 20] states: “The goal of building a ground model is to turn given informal requirements into a clear, unambiguous, accurate, complete and authoritative reference document for their intended content.” The ground model in this work is \textit{MGROUND–DG} detailed in Section 2.2. The relation of further models with this ground model is shown in Figure 1.1 in Section 1.2. The refinement method and equivalence of ASM models is discussed in Section 2.3.3.1.
A.4 ASM for grid computing

A formal model of (service) grids based on ASM was presented by Nemeth et al. in [1] and was refined later by Kertesz et al. [2]. Originally Nemeth et al. compared Grids with other distributed systems based on operational differences. They proposed a definition for Grids based on (runtime) semantics of the systems rather than comparing their static characteristics. They argue that conventional distributed systems (CDS) (e.g., batch systems on clusters) are a pool of computational nodes where the user has access (credential) to all nodes. This access means access to the resources on the node. The user is aware of the resources available and the nodes usually form a single administrative domain and are more or less static. Contrary grids form a virtual pool of resources where the user has access to the pool but not to individual resources. Also access to a specific resource may be restricted and the user has no knowledge about the specific resources. These resources may span multiple administrative domains. Size of the pool is larger than the pool in a conventional distributed environment and is dynamic. Nemeth et al. propose a model both for CDS and Grid. The Grid model is derived from the CDS model. I will focus on the grid model and detail CDS specifics only where they are needed for the description of the Grid model. Nemeth et al. argue that the main differences between grids and CDS stem from the acquisition of nodes and resources. CDS try to find nodes first and then satisfy the resource requirements locally while grids assume abundant resources. First the needed resources are found and then are the nodes selected on which the processes are mapped.

In their ASM model Nemeth et al. consider an application (members of universe APPLICATION) as consisting of several processes (universe PROCESS). All processes are owned by a user (USER) and need resources to perform work. Abstract resources are present in resource request and are represented by the ARESOURCE universe, while the PRESOURCE universe represents physical resources allocated to processes. Processes execute a specific task (universe TASK). The physical representation of a task is a static realisation of a running process, thus it must be present on the same node (universe NODE) where the process is. This is represented by the installed : TASK × NODE → \{true, false\} relation. Nodes, tasks and resources have certain attributes (universe ATTR). A subset of ATTR is the architecture type represented by universe ARCH. The relation compatible : ATTR × ATTR → \{true, false\} denotes whether to attributes are compatible according to some reasonable definition. A user can login to certain nodes if CanLogin : USER × NODE → \{true, false\} evaluates to true. A user is authorized to use given resource if the CanUse : USER × PRESOURCE → \{true, false\} relation evaluates to true. The model is centered around processes and their life-cycle is described by their states using the state : PROCESS → \{running, waiting, receive_waiting\} function. The model also has several functions related to handle messaging between processes, but these are not of interest here. During the execution of a task several events may arise that are represented by the event : TASK → \{⋯\} function. These involve a state change for one or more processes.

The model is a distributed multi-agent ASM where the agents are processes, i.e., elements of the PROCESS universe. It is depicted from the perspective of the processes, where the self function is represented as p ∈ PROCESS, i.e., p is interpreted differently
by different agents.

The initial state of the model describes the elements of the universes at the start. Nemeth et al. assume \( k \) processes belonging to an application and a user. All processes require some resources but own none. All processes have their assigned tasks but no one of them is mapped to a node. Finally there is a virtual pool of resources that the user has access to and her credentials are accepted by the pool.

In case of CDS the mapping between abstract (resources described in resource requests) and physical resources (resources provided by nodes) is implicit. However in grids the resource requests can be satisfied from various nodes in various ways. The user and the application has no information about the state of the pool of resources a new agent executing module is needed that handles the mapping between them, thus \( \Pi_{\text{resource\_mapping}} \) is introduced that provides the mapping via the \( \text{mappedresource} : \text{PROCESS} \times \text{ARESOURCE} \to \text{PRESOURCE} \) function. It does not specify how resources are actually chosen (it is rather an implementation detail), only assures that compatible physical resources are mapped to each resource request using the \( \text{compatible} : \text{ATTR} \times \text{ATTR} \to \{\text{true, false}\} \) relation. In grids the fact that a user can access the pool of resources does not mean that she can login to the nodes providing the resources \((\forall u \in \text{USER}, \forall r \in \text{PRESOURCE}, \forall n \in \text{NODE} : \text{CanUse}(u,r) \not\rightarrow \text{CanLogin}(u,n))\).

Resources are granted by the operating system to processes on the same node, thus a process of the application – belonging to the user – must be present on the node. However users are not authorized to login and start processes. This contradiction is resolved by providing a mapping between the real person, the user who has credentials to access to the resources of the pool (\( \text{globaluser} \)) and the user – not necessarily a real person – who has login rights on the node (\( \text{localuser} \)). The functionality that provides this mapping is done by the \( \Pi_{\text{user\_mapping}} \) agent executing module. Based on these statements Nemeth et al. propose a reference model and a definition for SGs.

In [2] the model was revised (and the concept of brokering added). There the model is depicted from the view of jobs rather than processes. Instead of the operating system perspective where processes can be considered as the basic units executing a batch executing system based view is adopted where jobs are submitted and executing. Conforming to this the self function is represented by \( j \in \text{JOB} \).

However after the revision the main statements of the original paper remain valid. As stated: “The modification is indicated by introducing more practical issues related to realization; [...] and finally by experiences in Grid Computing since the paper was published.” In this thesis the revised model by Kertesz et al. will be referred as \( M_{\text{GRID}} \).
A.5 GenWrapper examples

A.5 GenWrapper examples

1 IN='boinc resolve_filename input '
2 OUT='boinc resolve_filename output '
3 NUM='cat $IN ':
4 PERCENT_PER_ITER='expr 10000 / $NUM ':
5 for i in `seq $NUM ':
do
6 PERCENT_COMPLETE='expr $PERCENT_PER_ITER \* $i / 100 ':
7 boinc fraction_done_percent $PERCENT_COMPLETE:
8 echo -e "I am $PERCENT_COMPLETE\% complete.\n >> \$OUT "
9 sleep 1:
done

Figure A.4: Sample GenWrapper work unit shell script

1 BASEDIR='pwd'
2 gzip -cd 'boinc resolve_filename %{inputs}' | tar xvf -
3 exec 3<&2
4 exec 2> "$BASEDIR"/%{output_dir}/gridnfo.log
5 touch "$BASEDIR"/%{output_dir}/gridnfo.log
6 WORKFILE_PREFIX="WORKFILE"
7 MODEL_TYPE="Multilinear Regression"
8 cd "$BASEDIR"/%{input_dir}
9 mv IN-PARAMS IN-PARAMS.ori
10 echo "EXE_DIRECTORY = "$BASEDIR" >>IN-PARAMS
11 echo "WORKFILE_PREFIX = $WORKFILE_PREFIX" >>IN-PARAMS
12 echo "MODEL_TYPE = $MODEL_TYPE" >>IN-PARAMS
13 cat IN-PARAMS.ori | grep -v "EXE_DIRECTORY" | grep -v "WORKFILE_PREFIX" \
14 | grep -v "MODEL_TYPE" >>IN-PARAMS
15 rm IN-PARAMS.ori
16 mv IN-MATRIX "$WORKFILE_PREFIX.mt"
17 export LD_LIBRARY_PATH="$BASEDIR"
18 "$BASEDIR"/mda -p IN-PARAMS -r
19 mv "$WORKFILE_PREFIX.stdout" "$BASEDIR"/%{output_dir}/stdout.log"
20 mv "$WORKFILE_PREFIX.stderr" "$BASEDIR"/%{output_dir}/stderr.log"
21 mv "$WORKFILE_PREFIX.model" "$BASEDIR"/%{output_dir}/OUT-MODEL"
22 exec 2<&3
23 exec 3>&2
24 cd "$BASEDIR"
25 for i in 'find %{output_pattern} -name gridnfo.log'; do
26 cat $i
27 done 1>&2
28 tar cf - %{output_pattern} | gzip >'boinc resolve_filename %{outputs} ':

Figure A.5: Part of a more complex GenWrapper shell script used by CancerGrid
A.6 DC-API use case: Java applications for BOINC

Running Java applications on the BOINC platform represents two problems. First, BOINC API does not support Java, thus running an application written in Java would either require compiling it to native code or to use a wrapper designed for legacy (non-BOINC) applications. Second, Java requires a runtime environment on its own (Java Runtime Environment, JRE), which may not be already deployed on any client node or the already deployed version may not be suitable for the application. DC-API solves the lack of Java support in BOINC API by providing a Java binding of its API for Java applications via the Java Native Interface (JNI, [144]). JNI allows Java code to call and be called by native applications and libraries written in other programming languages, such as C or C++. The Java runtime deployment problem is solved either by bundling the JRE zipped with the application (license issues apply), or if the application is run in a Local Desktop Grid, then it can be assumed that the appropriate runtime is already deployed. Here we present the scenario when the Java runtime is assumed to be already deployed (see Figure A.6).

In this case Client 1 receives the following files as part of the application bundle:

- DC-API Java bindings and libraries
- A launcher application
- Java application .jar file(s) (WorkerApplication.jar)

A typical execution does the following steps:

1. The launcher is executed, but it does not contact the client. From the point of view of the BOINC client it’s an invisible application, it does not use any BOINC API or DC-API functions.

2. If the runtime is to be deployed with the application, then the launcher checks if it is already there (the application might be resuming from a checkpoint). If not found, then it has to be either installed, or uncompressed in the working directory.

![Figure A.6: A DC-API Java application on BOINC](image-url)
3. The launcher starts the Java application using the Java runtime. After that the launcher will wait for it’s termination. This step is necessary because the BOINC client determines the outcome of the task based on the exit status of the application, in this case on the exit status of the launcher.

4. The Java application behaves like a normal DC-API client application, it has access to the full set of DC-API client functions via the interface provided with JNI. Typically the following steps are executed:

(a) The application initializes the DC-API client library via the DCClient.init method.

(b) Resolves the location of it’s input and output files by calling the DCClient.resolveFileName method.

(c) During computation it calls periodically the DCClient.checkEvent method and processes the received events. One of the events is Event.isCheckpointRequest, upon this event the application should checkpoint itself.

(d) Whenever possible, the application should call the DCClient.fractionDone method with the percentage of the work completed. This will report the BOINC client, thus the user, the completion ratio of the current task.

(e) The DCClient.finish method should be called at the end of the computation with zero value (or anytime if error occurs with non-zero value). This will finish the execution.

5. After the application has finished, the launcher picks up it’s exit status and exits with the same value. The output files are sent back to the master, and it is also notified about the completion of the task.

Beside BOINC, the DC-API backend also supports Condor, the Grid Underground middleware and a simple fork-based implementation, thus the DC-API - Java interface is available on any of these platforms. Currently only DC-API client side functions are available via this Java interface, there is no support for using them on the master side yet. There is no restriction to use the same programming language both on client and master side, master applications should use the C/ C++ DC-API now.

Next I’ll show a functional demonstration and overhead measurements via a simple application: It searches for the first given number of prime numbers. It is a deterministic and CPU intensive application, thus fits perfectly the needs. It is configured to search for the first 100000 prime numbers (from 2 to 1299709) since this has a moderate, but similar run-time as a normal BOINC work unit (around 1 hour) on a nowadays PC. Two versions of the application was evaluated, one which uses DC-API (and BOINC) and a standalone version stripped of any DC-API dependencies.

The DC-API enabled one (see Figure A.7) initializes the DC-API library, reads the checkpoint file if any exists and continues the work. It checks periodically for any event and checkpoints itself on Event.isCheckpointRequest. When finished it calls DCClient.finish(0) which will terminate the application.
DCClient cli = new DCClient();
long x=2, c=0;
Event e;
cli.init();
readCheckpoint();
while (true) {
  if (c == count)
    break;
  c = isPrime(x);
  if (x % 1000 == 0) {
    e = cli.checkEvent();
    if (e == Event.isCheckpointRequest)
      doCheckpoint(x, c);
  }
  x++;
}
cli.finish(0);

Figure A.7: Structure of the DC-API enabled Java application

The standalone version reads the checkpoint file, does periodic work, checkpoints during the work and quits when finished. Since there is no event to signal the checkpoint request, we used the default interval of BOINC, which is 300 seconds, for checkpointing period. The invoked checkpoint function is only checkpoints when at least 300 seconds passed since the last invocation.
A.7 Wikipedia pre-processing for KOPI Cross-Language Plagiarism Search

When it comes to plagiarism, “Wikipedia is the most popular single source for both secondary and higher education students”, being responsible for more than 10% of the cases in higher education, getting almost 3 times as many hits as the next single source [145]. This means that it is crucial for any plagiarism detection engine to incorporate Wikipedia in its database to look for both copied and translated parts. The largest such source is the English Wikipedia with 3.9 million articles but the following two largest ones, German and French, have also 1.4 and 1.2 million articles respectively. Wikipedia is an always changing, constantly growing source. The number of English articles will reach 4 million probably before the end of the year 2012\(^1\).

The development of the KOPI Plagiarism Detection Portal [76] started almost ten years ago in 2003 at the Department of Distributed Systems of the Computer and Automation Research Institute, Hungarian Academy of Sciences (MTA SZTAKI DSD). The number of registered users increased constantly and reached 10,000 in 2009, with several university faculties already using KOPI on a regular basis. By the end of 2011 it was the first plagiarism checker to be able to detect translated plagiarism, searching in the full content of Wikipedia.

To keep the database up-to-date the new versions have to be incorporated as quickly as possible. The easiest way without constantly harvesting and stressing the servers of Wikimedia Foundation, is downloading the regularly published database dumps, chunk and convert them as fast as possible, then combine the results into a single dataset and finally update our database. We plan to convert much larger, never seen document collections harvested from the Internet, thus the periodical processing of Wikipedia data can be considered as a (very useful and important) test collection and as a first step. With the help of volunteer computing each Wikipedia database dump and other documents are converted in batches by volunteer resources. KOPI is deployed at the SZTAKI Desktop Grid (SZDG) (see Section A.7.3) which is a BOINC based desktop grid. However using volatile volunteer resources for computing batches of jobs raises some issues. The most important is that the roundtrip time of the job running on the most non-reliable or slowest resource is going to determine the completion of the whole batch (i.e., the conversion of a Wikipedia database dump) and thus it affects the update frequency of our search database. There are different approaches for mitigating this, e.g., resubmission of jobs that are overdue. We have employed such a method since the start of the desktop grid version of the project in October 2011. Based on my measurements, this approach allows to achieve up to 61.25% improvement of mean round-trip times for jobs within a batch with only 25.1% of overhead caused by the resubmission.

We refer to a Wikipedia dump as a “batch”, tasks belonging to a batch and submitted to 3G Bridge as “jobs”, and finally tasks in the SZDG middleware as “work units”. Depending on the application settings of the desktop grid a single submitted job can be executed by multiple volunteers.

A.7.1 Workflow

Figure A.8 shows the two main parts of the workflow of the Desktop Grid enabled KOPI system. First is the preprocessing of Wikipedia data with the help of SZDG. Here, one at a time, a periodically published Wikipedia database dump is sliced into multiple jobs (also called as work units) by the **Work Unit Generator**.

These work units are submitted as a single batch to the volunteer computing project for processing. The returned results are continuously assimilated into a single dataset. Once all work units are finished the dataset is complete and the KOPI database is updated. This process must be repeated for each different language Wikipedia dump and every time a new dump is published by Wikipedia. The exact steps are detailed in the next section.

The second process (see ii. in Figure A.8) is the plagiarism search initiated at the KOPI Portal (see Section A.7.2). This is a request in the form of a document submitted by the user of the portal. First, during candidate selection the submitted text is compared to the KOPI database and suspicious sentences and fragments are selected. These candidates are then evaluated based on a similarity metric and the most promising ones are returned to the user via the portal. Section 3.3.3 discusses the details of the selection and evaluation.

Figure A.9 shows the overview of the architecture that is used for preprocessing the Wikipedia content for the KOPI Plagiarism Search Portal using desktop grid resources. The architecture consists of four main components which are detailed in the following subsections. First the KOPI Server related parts: (i) the Master (Work Unit Generator and Dataset Builder) and the (ii) Scheduler used for improving batch completion times; then the (iii) SZTAKI Desktop Grid (SZDG) Server [13] components and volunteer resources;
and finally (iv) the KOPI desktop grid application deployed at the SZDG Project.

![Figure A.9: The architecture used for Wikipedia data preprocessing](image)

### A.7.2 KOPI Server and portal

The KOPI Server (shown in Figure A.9) interacts with the volunteer desktop grid and contains two components: (i) the Master and (ii) the Scheduler. A single Wikipedia dump is submitted and handled as a batch. The Work Unit Generator (Master) is responsible for partitioning these XML dumps into smaller pieces for the desktop grid and combining the results into a single dataset as they return. Wikipedia publishes its content periodically in an XML format file for each different language. They have a size of several GBs (e.g., Hungarian - 1.7GiB, French - 8.7GiB, German - 10GiB and English - 36GiB) and are therefore too large and require too much CPU time to be processed sequentially. Fortunately the pre-processing procedure of these dumps can be easily parallelized by splitting them into smaller chunks (e.g., by splitting on article boundaries). The chunks can be sized arbitrarily, however there are several constraints that should be honored: (1) size of the input and output files and thus the time spent doing network I/O should be small compared to the time spent processing (e.g., inputs and outputs less than 10MiB); (2) runtime should be less than an hour, since the legacy code on volatile (volunteer) resources has no checkpointing mechanism. Based on our empirical studies a work unit with around a total of 3 MiB input files requires an hour of processing time on a current dual core computer (however the application is not multi-threaded) and will create around a 7 MiB compressed output file. The Scheduler component is responsible for the submission and management of the work units of the batch. It is also responsible for minimizing the total time required to finish the batches (“makespan”). This is currently achieved by resubmission techniques which are discussed in detail in Section 3.3.3. The dataset building part of the Master is responsible for processing the computed results returned from the desktop grid and combining them into a single dataset that will be used for updating the KOPI database.
The KOPI Online Plagiarism Detection Portal is a unique, open service for Hungarian and English speaking web users that enables them to check for identical or similar contents between their own documents and the files uploaded by other authors. The monolingual plagiarism search function works in any European language, due to the language-independent algorithm. At the time of this writing Hungarian, English and German texts can be compared to the English and Hungarian Wikipedia, but the translational detection system is being continuously improved to support other languages as well. On the web interface (see Figure A.10) the user can upload documents and start a plagiarism search using these documents and the options defined above. When the search is finished, the result is emailed to the user and can be seen on the portal interface as well. If the document was compared to the Wikipedia, a list of possible Wikipedia articles is returned (see Figure 4), with the title of the article, the original sentences, and the suspicious ones from the document. Thus the user can hand-check the result given by the system, which is necessary as the system does not decide in place of the user: it does not distinguish between citation and plagiarism, the last word is that of the users.

The KOPI Server (work splitting and result combining) and Portal are developed and maintained by MTA SZTAKI DSD.

A.7.3 SZTAKI Desktop Grid

SZTAKI Desktop Grid (SZDG) Project [77] is a BOINC based volunteer desktop grid project running since 2005 at the Laboratory of Parallel and Distributed Systems (LPDS) of MTA SZTAKI. SZDG Currently has over 41,000 registered volunteers with more than 92,000 hosts total. Contrary to traditional desktop grids, which run usually a single application, SZTAKI Desktop Grid is considered as an “umbrella” project since it hosts many applications, KOPI being one of them. Any volunteer has the freedom to select which application they want to run from those deployed at the project. This means that appli-
cations usually compete against each other for a given set of resources and each of them can use a fraction of the available total for the project.

3G Bridge [146] running on the SZDG Server (as shown in Figure A.9) is a generic grid-grid bridge, which provides interoperability between different distributed computing infrastructures (DCIs) on a job based level. It is able to fetch jobs from different DCIs, queue them internally and submit them to different middleware using its modular plug-in architecture. Its “WSSubmitter” component provides a generic SOAP based web service interface for accessing infrastructures and remote job management: it allows submitting jobs directly to the different queues of 3G Bridge and also managing these jobs. 3G Bridge provides command line tools as well, which can access the web service interface, thus allowing (a) remote job submission to SZDG (or any BOINC based project with 3G Bridge deployed); (b) querying the status of running jobs; (c) cancelling jobs and (d) retrieving their outputs. In our case this service and the remote access feature are used by the KOPI Server components to interact with SZTAKI Desktop Grid.

A.7.4 Wikipedia pre-processing on SZTAKI Desktop Grid

As detailed in Section 3.2.3 developing and porting applications to volunteer computing is not a straightforward process. In the case of KOPI, chunking and natural language processing are used at two distinct places in the whole system. First, for the data used for dataset building and database update (see i in Figure A.8), second, when the suspected document is compared to the index and the candidate chunks (see ii in Figure A.8). It is necessary to use exactly the same method at these two points to ensure that even if there are some bugs or errors (like a stemming error) they are the same on both sides. The easiest way to ensure this is to have only one implementation of the functions. The functionality was originally implemented in PHP at the KOPI Portal, so rather reimplementing it again this code is run on the desktop grid as well.

The resulting code is possibly slightly slower than it would be in case of using compiled code (e.g., C++). However, the stemming, which is one of the slowest parts, is done by an external component, namely Hunspell\(^2\). This functionality (the preprocessing) is resource-intensive and embarrassingly parallel, therefore it can be delegated to volunteer computing. The preprocessing performs five distinct steps: (i) language identification; (ii) MediaWiki XML to plain text conversion; (iii) text cleansing and chunking; (iv) stemming; and (v) output creation.

Language identification (see (i) above) is detailed in [38]. The client can detect the file format of the input document; in case MediaWiki is detected it will be converted to plain text (see (ii) above). There are several possibilities to convert wiki format to plain text. The most obvious would be to install a MediaWiki instance and harvest the pages. This, however, raises two obstacles: first, on the desktop grid client this would be unfeasible; secondly, this would generate high load even on a server (which the clients could somehow access) as the XML to HTML conversion is compute-intensive. The other option is to convert it with some stand-alone software. Most of the software available freely are either operating system dependant or need installed software, which makes them unsuitable to be

\(^2\)http://hunspell.sourceforge.net/, accessed on 2012-05-01
used in desktop grid environments. Others make errors when converting special pages, or truncated long ones. For these reasons the conversion is done by a self-written component with the following features: \(i\) the names and boundaries of the Wikipedia pages are kept; \(ii\) within this only the textual information is necessary; \(iii\) “info boxes” – as they are duplicated information – are filtered out; \(iv\) comments, templates and math tags are dismissed; and \(v\) other pieces of information, like tables, are converted to text.

For the purposes of multilingual plagiarism search having text versions is only half of the work. Each XML file contains thousands of Wikipedia articles which have to be split to determine where the copied content can be found at the plagiarism search level. Articles are then split into smaller chunks (see \(iii\) above), according to the algorithm used by the search engine and all non-alphanumeric characters are disregarded, as they do not carry any useful information. After chunking the chunks are stemmed word-by-word (see \(iv\) above) and unlike by automatic translation engines, all the possible stems (lemmas) are kept (e.g., divers → divers (assorted), diver). In English this is not so important, but for agglutinative languages like Hungarian this step is essential as word-forms have more often multiple lemmas. The free Hunspell program is used for stemming, as it is available for both Linux and Windows with dictionary files for more than 100 languages and dialects. Dictionary files are 1-2 MiB large per language so only English, German, French and Hungarian are currently included in the client. The last step of the process is the creation of the output result file (see \(v\) above), which is used as an input for the database upload and indexing step at the server. This file contains all the necessary information for the search process: the fast indexed candidate selection and the linear similarity metric.

Our goal was to reuse the existing implementation from the KOPI Portal code – written in PHP – and to be able to update the desktop grid application with little effort if the portal code changes over time. This meant that we could not use the BOINC API directly, thus alternative methods for application development had to be considered. GenWrapper [30] (see Section 3.2.3.3) is aimed at specially solving these types of problems. In the case of KOPI this meant that no changes to the portal code were necessary. Only the GenWrapper script had to be written, which performs the following steps: (1) bootstraps the environment by deploying the bundled PHP and Hunspell binaries in the working directory, (2) prepares the input files; (3) executes the portal code via PHP which then processes the input files and finally (4) collects and compresses output files to be uploaded to the desktop grid server. The resulting desktop grid application bundle including all binaries is only 8-13 MiB, depending on the operating system.
A.8 Augmenting EDGeS@home with on-demand reliable resources

Service grid (SG) systems based on gLite [147], UNICORE [148], Globus [149] and ARC [150] middleware are intensively used to run parametric applications where the same code is executed with a large number of different parameter systems. The maintenance of such grid systems is quite expensive since they typically use managed clusters as computing element resources. A much less expensive alternative to run such parametric applications would be the use of volunteer desktop grid systems where instead of managed clusters the spare time of desktop machines is used as computational resource. Of course, it would make no sense to throw away the existing SG systems and rebuild the infrastructure on a VC basis. The EU FP7 EDGI project [20] proposed to maintain the existing SG systems and extend them with the extremely cheap VC resources. The extension should be as easy and as transparent as possible both for the users and for the administrators of virtual organizations (VO). Utilizing a VC project as computational resource for a service grid i.e., gLite infrastructure, raises two problems to be solved. The first one is originating from the different security mechanisms of the two systems. While gLite trusts the users, where a certified user can submit any (“just been developed”) kind of jobs, on VC jobs can be sent only for registered (at the volunteer desktop grid) applications. To overcome this limitation, EDGI has designed two alternative solutions: referring pre-validated applications in application repository or encapsulate the application into the GBAC virtualization framework (see Section 3.2). The second problem of marrying two such different systems is the difference between the throughput and completion time of multi-job applications. In a volunteer desktop grid the number of resources can significantly exceed the average number of resources in a service grid VO. As a result, the throughput of volunteer desktop grids is typically much bigger than that of gLite VOs. On the other hand, in gLite, jobs run on dedicated resources, while desktop grids utilize volunteer, untrusted resources. Because of this, it typically happens that a small fraction of jobs are stuck on unreliable volunteer resources; and if the application strictly requires the successful completion of all the component jobs, then the completion time of such application could be longer in the volunteer desktop grid than in the gLite VO, even if the number of resources is smaller in the VO. Notice that there are applications (e.g., Monte Carlo simulations) where we do not need all the jobs to be completed in order to get a successful completion of the application. For such applications EDGI has developed the “Meta-job” [151] concept. In order to significantly reduce the completion time for the other types of applications, I developed a scheduling solution on the BOINC server and a dedicated cloud infrastructure support solution where a small but dedicated cloud can solve this problem.

One of the goals of EDGI was adding certain Quality of Service (QoS) capabilities to desktop grids to improve task completion times. The project investigated different alternatives like redundancy with resubmission [38], however the final method chosen was resubmission with reliability based scheduling. BOINC provides information about reliability and availability of the connected (volunteer) resources that can be used to determine the most reliable resources. These resources can be used as complementary ones, but pri-
marily resources from private Infrastructure-as-a-Service (IaaS) clouds provided by the consortium members were utilized, with the option to scale out to public providers, like Amazon Web Services. It has been designed as a non-interactive solution, meaning that the system automates the process of task redirection for the users without any interaction from the users. Table 3.4 describes the available IaaS cloud resources for EDGeS@home. These are single virtual core resources, thus each resource provides a worker with a single core CPU. Each resource runs a special Virtual Appliance that contains the BOINC Client. These instances run 64bit version Debian Linux 6.0 and have at least 2GB free space for BOINC tasks. The instances are instructed via contextualization data to attach to a given BOINC project with given user credentials. As Table 3.4 shows (in its 3rd column) each private cloud has a dedicated user at EDGeS@home. The dedicated users are used to group together the resources coming from the different clouds for the reliability based scheduling. Also the profiles of these users are public i.e. their performance can be checked on the website of EDGeS@home project. As shown in Figure 3.17 the cloud resources are managed from a dedicated virtual machine (“EDGI Cloud Management”) from the LPDS cloud. This management VM instructs all configured clouds. It uses an enhanced version of the solution described in [40]. The management node is responsible for starting and stopping instances as required. It is able to interact with different IaaS middleware, namely OpenNebula, OpenStack, Eucalyptus and Amazon Web Services using the de facto standard Amazon EC2 interface. When a new worker is requested, the URL of the BOINC project and a specific authenticator unique to each configured cloud is used as contextualization. This allows them to connect to the BOINC project. The instances remain attached to the BOINC project and continue processing tasks until they are terminated by the Manager. This way, the different configured private clouds contribute reliable resources for EDGeS@home. However, this solves only the first part of the problem. For the second part, EDGI developed a complimentary Scheduler component for BOINC that reassigns problematic tasks to multiple sets of resources. This component currently assigns the oldest non finished tasks in the system to one of the configured reliable resource group. In this case a resource group is the set of resources made available from each connected private cloud.

A.8.1 EDGeS@home volunteer desktop grid

EDGeS@home [73] is a BOINC based volunteer desktop grid project. It has currently over 13,000 registered volunteers with 21,000 registered hosts total. Running since 2008, it was started in the frame of the Enabling Desktop Grids for e-Science (EDGeS) EU FP7 project [43, 80]. It was maintained further by the successor of EDGeS, the European Desktop Grid Initiative [20] (EDGI) until late 2012 and is currently maintained by the International Desktop Grid Federation support Project (IDGF-SP)\(^3\). Traditional desktop grids usually target a single grand challenge scientific problem. Contrary to the traditional ones, EDGeS@home is running multiple applications, making it an “umbrella” project. Any volunteer has the freedom to select which application(s) she or he wants to support from the deployed ones. This means that applications are usually competing against each other

\(^3\)http://idgf-sp.eu, accessed on 2013-10-01
A.8. Augmenting EDGeS@home with on-demand reliable resources

for a given set of resources and each of them can typically use a fraction of the client machines registered for the project. EDGeS@home currently hosts 10 applications from different science domains like physics, logistics and biology. In this section I describe in detail how the different problems have been addressed for the gLite → BOINC integrated systems. There are solutions (also developed by EDGI) for extending other SG systems (ARC and UNICORE) with another DG system (XtremWeb), but these are not explained here, rather details can be found in [152] about ARC and in [153] about UNICORE.

A.8.2 Augmenting virtual organizations with desktop grids and volunteer computing

The goal of extending gLite VOs with BOINC systems is to transparently transfer parametric jobs from the gLite VO to one or more supporting BOINC systems and to distribute large number of job instances of parametric jobs among the large number of BOINC client resources. In order to extend gLite VOs with BOINC systems a bridging solution was designed by MTA SZTAKI. The key component is the modified Computing Element (mCE), see “EDGI gLite modified CREAM CE in Figure A.11. It extracts the job from the gLite system and transfers it to a remote desktop grid site. On the remote BOINC server a 3G Bridge service [44] (see “3G Bridge” in Figure A.11) is running to receive the incoming jobs and to insert it into the BOINC server. These two components (modified CREAM CE and the 3G Bridge) in the infrastructure represent the two pillars of the bridge. Due to scalability issues on the gLite side the Meta-Job [151] concept was introduced that allows submitting a large number of jobs through the gLite mCE as one single job. To handle heterogeneity of the volunteer resources and the untrusted or unsafe applications, the GBAC virtualization framework (see Section 3.2) has been introduced to enable executing jobs inside virtual machines on the BOINC clients. To control which jobs can be transferred through the bridge, the EDGI application repository [74] (“EDGI Application Repository” in Figure A.11) has been introduced.

![Figure A.11: EDGI Infrastructure to bridge gLite jobs to BOINC DGs](http://www.edges-grid.eu/web/edges/49, accessed on 2013-03-23)
However, with the help of the virtualization environment GBAC provides, mCE became able to automatically convert non-registered applications into a GBAC workunit under the target BOINC project. Using virtualization on volunteer resources, untrusted applications can also be executed safely under BOINC. Moreover, gLite users do not need to modify their submission JDL file when changing from normal gLite resources to desktop grid ones. Finally, the extension of the desktop grid infrastructure with on-demand dedicated resources from academic and public clouds (see “On-demand dedicated BOINC clients” in Figure A.11) is the essence of the overall system. Before giving more detailed introduction of the cloud integration, first let’s discuss the two most popular deployment cases of creating such a combined infrastructure. In the first case, extending the existing gLite VO with already gathered and maintained volunteer resources is shown, while the second case details setting up the combined infrastructure based on an institutional desktop grid system.

A.8.2.1 Agumenting with an existing volunteer computing project

In EGI\(^5\), there are already many gLite VOs around Europe for which gLite administrators are already taking care of the infrastructure and are not really willing to take extra work by operating a separate desktop grid site for collecting volunteer resources. To ease their work, EDGI and IDGF-SP offers for the gLite VOs providing and maintaining the extension services (“Service Grid extension” in Figure A.11) as well as the required volunteer desktop grid project: EDGeS@home with a large number of volunteer desktop grid resources and on demand dedicated resources from (mainly) academic IaaS clouds in addition. In this scenario, the gLite VO admin has nothing to do; EDGI (or IDGF-SP) takes care of the operation of the modified computing element, the 3G Bridge and takes care of all the desktop grid related tasks. After EDGI performs the necessary configuration, the gLite mCE appears among the CEs of the supported VO. This experiment is an ongoing and continuous activity performed by the IDGF-SP project and feedbacks are positive related to performance detailed in Section 3.3.2.

A.8.2.2 Augmenting with a new desktop grid project

In institutes BOINC can be used as an effective solution for aggregating the available compute capacity of desktop computers. Universities often follow this strategy to create a campus-wide desktop grid system to provide huge computational capacity for the scientific researchers of the university. University of Westminster is a good example, where approximately 1800 machines have been collected / utilized by a campus desktop grid and offered for their scientists to run various simulations \[11, 154\]. Such a newly built campus DG site is a good candidate to be bridged to a gLite VO where the university is a member and is a good candidate to provide a significant capacity increase to that VO. To support this scenario, the IDGF-SP project has consolidated all the necessary software components (gLite mCE, AR, 3G Bridge, GBAC, etc.) and provides them freely for academic

\(^5\)http://www.egi.eu, accessed on 2014-01-01
A.8. Augmenting EDGeS@home with on-demand reliable resources

institutes. Moreover, the project set up a Wiki\textsuperscript{6} for system administrators how to setup and operate a combined gLite desktop grid infrastructure. This scenario requires very small effort from the administrators of the university if the required components (BOINC server, 3G Bridge, gLite mCE) of such an infrastructure are deployed in a cloud system. In order to support this easy deployment and maintenance, IDGF-SP provides the necessary virtual appliances, too.

\textsuperscript{6}http://doc.desktopgrid.hu, accessed on 2014-01-01
Bibliography


[34] A. Rivero, A. Bustos, A. C. Marosi, D. Ferrer, and F. Serrano, “Isdep, a fusion application deployed in the edges project,” in Enter the Grid: Proceeding of the Third AlmereGrid Desktop Experience workshop, pp. 18–22, AlmereGrid.nl, 2010. (Cited on pages 3 and 91.)


[144] “Java Native Interface.” (Cited on page 154.)


