J Supercomput (2008) 46: 124–138
DOI 10.1007/s11227-007-0160-2

An adaptive and trustworthy software testing framework on the grid

Yaohang Li · Yong-Duan Song

Published online: 6 December 2007
© Springer Science+Business Media, LLC 2007

Abstract Grid computing, which is characterized by large-scale sharing and collaboration of dynamic distributed resources has quickly become a mainstream technology in distributed computing and is changing the traditional way of software development. In this article, we present a grid-based software testing framework for unit and integration test, which takes advantage of the large-scale and cost-efficient computational grid resources to establish a testbed for supporting automated software test in complex software applications. Within this software testing framework, a dynamic bag-of-tasks model using swarm intelligence is developed to adaptively schedule unit test cases. Various high-confidence computing mechanisms, such as redundancy, intermediate value checks, verification code injection, and consistency checks are employed to verify the correctness of each test case execution on the grid. Grid workflow is used to coordinate various test units for integration test. Overall, we expect that the grid-based software testing framework can provide efficient and trustworthy services to significantly accelerate the testing process with large-scale software testing.

Keywords Grid computing · Software testing

1 Introduction

During the last several decades, many revolutionary changes, such as object-oriented programming, object modeling, and pattern design have taken place in software engineering. The world of software development has undergone a lot of change these days

Y. Li (✉) · Y.-D. Song
Department of Computer Science, North Carolina A&T State University, Greensboro, NC 27411, USA
e-mail: yaohang@ncat.edu

Y.-D. Song
e-mail: songyd@ncat.edu

with the influx of modern methods, paradigms, technologies, and tools. At the same time, the functionality and complexity of modern applications grow exponentially. However, behind the evolution of software there has been one bigger challenge in quality control techniques. Software testing, which is a fundamental part of software engineering, lags behind the development ones and cannot provide the same level of quality for complex applications with manageable growth of effort. Software testing is an integral, costly, and time-consuming activity in the software development life cycle. In a complex software development project, time, resources, and effort seem to be never allocated enough to testing activities. The ISSRE'97 panel [1] concluded the following facets of large-scale software testing.

> "Large software product organizations spend 50% or more of their budgets on testing. Testers comprise 20% to 50% of software personnel in many companies. With all this effort, best in class software products still may contain 400 faults per million lines of fielded code. The cost of repairing field failures is growing rather than shrinking."

As one can foresee, reducing software testing cost will directly lead to significant overall software development cost and time reduction.

Grid computing is characterized by large-scale sharing and cooperation of dynamically distributed resources, such as CPU cycles, communication bandwidth, and data to constitute a computational environment [2–4]. A computational grid can, in principle, provide a tremendously large amount of low-cost CPU cycles, which may be utilized to speed up software testing or to examine many more test cases [5]. Moreover, software portability testing on different computer system architectures and configurations is always a formidable task within the traditional software testing process. A computational grid is a heterogeneous computing environment, which is usually comprised of a large array of hardware architecture, operating systems, and middleware library combinations. All these make a computational grid a natural and possibly an ideal testbed for large-scale software testing. Several conceptual models [6, 7] have been developed with intention to speedup the software testing process by taking advantage of the grid computing resources.

Despite the attractive characteristics of grid computing, successfully fostering new approaches using the grid techniques for large-scale software testing depends on overcoming a number of challenges. First of all, the computational resources in the grid exhibit greatly heterogeneous performances which may change unpredictably as time evolves. Also, due to unreliable network connections and the possible unavailability of a grid service, a test case may be halted at any time. Efficiently taking advantage of these resources requires an adaptive scheduling algorithm. A more serious problem is a computational grid in a potentially untrustworthy computing environment [8]. Many grid computing projects, such as SETI@home [9, 34], Folding@home [10], and distributed.net [11] have recorded various misbehaviors of grid computational service providers, malicious grid users, or malfunctioned grid applications. Indeed, untrustworthy grid components post significant threats to the correctness of software testing projects being carried out on the grid. Identifying and excluding the untrustworthy test results is critical to the success of grid-based software testing.

In this article, we try to address a question—can we compose the heterogeneous, widely-distributed, dynamic, and even potentially untrustworthy grid resources efficiently to support trustworthy and reliable large-scale distributed software testing? We present our undergoing development of the grid-based software testing framework to facilitate the automated process of utilizing the grid resources for software unit and integration testing. Inside the grid-based software testing framework, several novel techniques are developed to address the challenges posted by the characteristics of the grid. A bio-inspired scheduling algorithm is developed to provide adaptive test case scheduling; mechanisms supporting consistency checking, such as redundancy, intermediate values checking, separation of test execution and result comparison, and verification code injection are used to identify and exclude potentially untrustworthy test results; and grid workflow is employed to coordinate various test units for integration test. Overall, we expect that the grid-based software testing framework can provide trustworthy and efficient services for large-scale complex software testing.

2 Unit testing and integration testing

In software testing, within all levels, unit test is the fundamental one, which goes together to make the "big picture" of testing a software system. In software engineering literature [12], a unit is defined as the smallest collection of code which can be usefully tested. Typically, a unit would be a nontrivial object class, a subroutine, a script, or a module of source code. A unit test is a procedure used to verify whether a particular unit is working correctly or not. The main idea about unit tests is to write test cases for all units so that whenever a change causes a regression, it can be quickly identified and fixed. Ideally, each test case is separate from the others, constructing mock objects that can assist in separating unit tests. A software system with nontrivial complexity is usually composed of a large number of units while each unit may have test cases ranging from several to thousands or even more. Every test case must be executed many times along the software development life cycle when a new module is added, an existing functionality is modified, or a software defect is fixed. As a result, in a large and complex software system, running a large number of unit test cases is rather computationally costly.

The integration testing is the phrase of software testing in which individual units are combined and tested as a group. In integration testing phrase, units that have been checked out by unit testing are grouped in a particular order into larger aggregates and applied tests defined in an integration test case to these aggregates. All integration test cases are constructed to test that all components within an aggregate interact correctly. As a result, compared to the unit testing phrases, more complicated test cases with different combination of units are demanded in the integration testing phrase.

Therefore, a powerful testbed with large-scale of computational capability, which can effectively carry as many unit test cases as possible and automatically integrate individual units is desired for complex software system testing. Fortunately, most of these unit or integration test cases are embarrassingly parallel, which is a natural fit for a massively parallel computing environment like a computational grid.

3 Issues of software testing on a computational grid

Since the late 1990s, grid computing has emerged as an important new area in parallel computing, distinguished from traditional distributed computing by its focus on large-scale dynamic, distributed, and heterogeneous resource sharing, cooperation of organizations, innovative applications, and high-performance orientation. Many applications have been developed to take advantage of grid computing facilities and have already achieved elementary success. However, as the field grew, so also did the problems associated with it. Traditional assumptions that are more or less valid in traditional distributed and parallel computing settings break down on the grid. In traditional distributed computing settings, one often assumes a "well-behaved" system: no faults or failures, minimal security requirements, consistency of state among application components, availability of global information, and simple resource sharing policies. While these assumptions are arguably valid in tightly coupled systems, they break down as systems become much more widely distributed [8]. First of all, the grid is a dynamic computing environment without centralized control. Nodes providing grid services join and leave dynamically, possibly without any notice. Secondly, grid service providers in a computational grid exhibit heterogeneous performances. The capabilities of each grid service vary greatly. A service provider might be a high-end supercomputer, or a low-end personal computer, even just an intelligent widget. As a result, a task running on different grid service providers will yield a huge range of completion times. In [5], Durate et al. showed that the instability of network significantly affects the performance of distributed software testing. Thirdly, there may be untrustworthy or unreliable services existing in the grid [34]. Collaborating with the potentially untrustworthy grid services may pose a security threat to the target tested software. Fourthly, hundreds or even thousands of various test units may be carried out in different distributed grid services. Integrating and managing these test units can be a complicated job. As a result, all these issues mentioned above must be addressed before a large-scale computational grid can be efficiently utilized to speedup software testing in complex systems.

4 Grid-based software testing framework

The grid-based software testing framework is designed on top of the fundamental grid services provided by the Globus toolkit [13, 14], including GRAM (Globus Resource Allocation Manager), GIS (Grid Information Service), GSI (Grid Security Infrastructure), and GridFTP (File Transportation Protocol on the Grid). Via the fundamental grid services provided by Globus, the grid-based unit test framework can invoke the local unit test tools for various programming languages, such as JUnit [15], NUnit [16], dbUnit [17], pyUnit [18], or other unit test tools according to the programming language requirement of the target software system to carry out various unit test cases. Multiple unit testing grid services can be plugged into a grid workflow [19, 20] to formulate a particular integration testing case. Moreover, an adaptive scheduling scheme using swarm intelligence is employed to efficiently use the computational resources and mechanisms, such as redundancy, internal consistency check, and external consistency check are used to enforce trustworthiness in the test cases. Figure 1 illustrates the overall system architecture of the grid-based software testing framework.

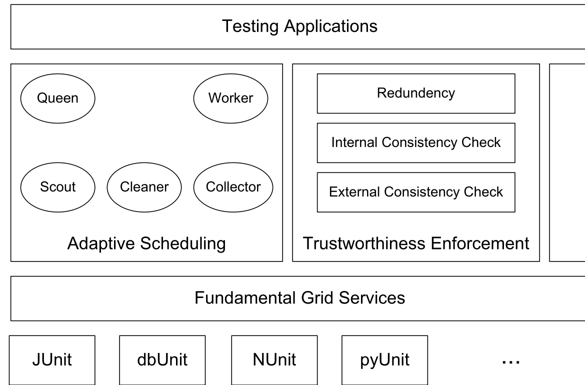

Fig. 1 System architecture of grid-based software testing framework

5 Adaptive test tasks scheduling on the grid

A computational grid is usually a dynamic computing environment, where participant computational services potentially exhibit unpredictably and dynamically changing behaviors. To effectively and efficiently take advantage of the computational grid for large-scale software testing, an adaptive testing task scheduling mechanism for the dynamically changing computing environment is necessary.

The nature is an excellent teacher of showing adaptability. One interesting example is referred as swarm intelligence [20], where social insects, such as bacteria [21], ants [22], and caterpillars [23], exhibit a collective problem solving capability, which shows strong adaptability and robustness to dynamically changing environment. Algorithms that take inspiration from swarm intelligence in finding shortest paths have recently been successfully applied to combinatorial optimization [24], circuit switched communications network problem [25], and adaptive routing problem [26]. In our grid-based software testing framework, we adopt the swarm intelligence approaches to develop an adaptive mechanism for scheduling unit test cases on the grid.

5.1 Dynamic bag-of-tasks model

The grid-based unit test framework manages test cases using the dynamic bag-of-tasks (also called bag-of-work) model [27–29], which applies to the situation when embarrassingly parallel computational tasks are to be executed a large number of times. The dynamic bag-of-tasks computing paradigm favors applications with embarrassingly parallel characteristics, i.e., situations where the overall computing task can be easily divided into smaller independent subtasks. This situation arises naturally in large-scale unit test scenarios, where every test case is independent. A set of parameters for a software unit forms a test case and a set of test cases constitutes a testing task. The collection of all tasks to be executed is called the bag of tasks, since they do not need to be solved in any particular order. A testing task may contain thousands of test cases and may come to the bag of tasks at any time. Also, to improve reliability of the software testing, replicated testing tasks may be present in the bag of tasks [30].

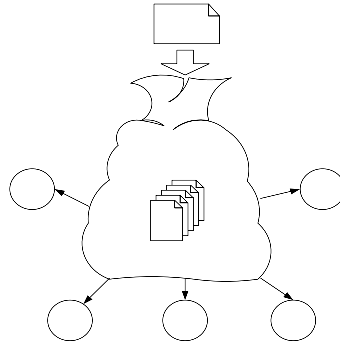

Fig. 2 The dynamic bag-of-tasks paradigm

The dynamic bag-of-tasks model usually employs the master-slave scheduling paradigm, where the master is responsible of dispatching appropriate tasks while multiple slaves carry out their tasks. When a slave is free, the master will select an appropriate testing task from the bag and assign it to the slave. After successfully receiving the task, the slave then computes a partial test result. Finally, the master collects the distributed partial test results to generate a testing report. Figure 2 shows the master-slave scheduling paradigm in the dynamic bag-of tasks model for large-scale unit tests.

5.2 Swarm intelligence approach for test units scheduling

The goal of a task-scheduling algorithm of the grid-based unit test framework is to minimize the execution time of the unit test tasks by efficiently taking advantage of the large amount of distributed computational resources available in the grid. In our implementation of the grid-based unit test framework, we extend the dynamic bag-of-tasks scheduling model by a swarm intelligent approach based on Ant Colony Optimization (ACO) to tackle the performance heterogeneity and resource dynamism problems presented in the grid computing environment.

5.3 Ant colony optimization

Ant colonies are distributed systems and exhibit a collective problem solving capability, which shows strong autonomous adaptability in dynamically changing computing environment as well as robustness to system failures. This property is a famous example the swarm intelligence. Within an ant colony, ants are specialized in particular unsophisticated functionalities and interact with their environment to exhibit globally collective intelligence. As a result, a swarm of ants in an ant colony can accomplish astonishingly complex tasks such as "foraging," i.e., finding the shortest paths between food sources and their nest. These tasks could never be performed by a single ant. The foraging behavior and the collaboration of specialized type of ants in an ant colony inspire us to investigate in the ant colony's behavior and adopt this mechanism in adaptive unit test task scheduling on the computational grid.

5.4 Agents in a testing task scheduling ant colony

In our scheduling mechanism using swarm intelligence, we design various software ant agents with simple functionalities. No direct communications occur among these ants. The only indirect communication is via the pheromone values stored in a grid resource table. These specialized ants are categorized as follows:

- Scout: The responsibility of the scout is to discover the new grid services providing appropriate computational services. Once such a new grid service is found, the scout adds it to the available resource table with an initial pheromone value.
- Worker: A worker chooses an available grid services and carries out a testing task in the system. The grid services with higher pheromone value will be assigned a "bigger" task with possibly more test cases.
- Collector: When a testing task is complete on a grid service, a collector will retrieve the partial testing results. Also, according to the task completion time and the number of test cases in the task, the collector updates the pheromone value of this particular grid service.
- Cleaner: A cleaner maintains the available grid resource table in the system. It removes the unavailable resources (with low pheromone value) from the grid resource table.
- Queen: The queen is responsible of producing the specialized ants, including the scouts, testers, cleaners, and workers.

All these ants fulfill their own simple functionalities. There is no direct communications among all these ants.

5.5 Scheduling mechanism

Let us put all the pieces of the scheduling algorithm together. The swarm intelligence mechanism of testing task scheduling on the computational grid is depicted as follows:

1. Initially, the queen spawns scouts, cleaners, and workers. The queen also produces testers at a time period of T.
2. A scout visits the information services providers of the grid and explores those software unit testing service providers. The scout finds the available grid services and adds them to the grid resource table with initial pheromone value, θ.
3. Once a testing task is submitted to the computational grid, a worker will try to schedule this task to an available grid services in the grid resource table. A grid service having a higher pheromone value will be selected with a higher probability. A grid service i will be selected with probability, q_i, of

$$q_i = p_i \Big/ \sum_{j=1}^{n} p_j,$$

where p_i is the pheromone value of grid service i and n is the total number of available grid services in the grid computing environment.

4. Global and local pheromone updates are presented in the algorithm. In local update, when a test task is complete on a grid service, the collector will retrieve the test result and update the pheromone value

$$p_i \leftarrow p_i + \Delta p_i,$$

where Δp_i is the newly added pheromone amount.

Global updates take place at every period of time T_1. The grid service who completes most schedule units will obtain an extra bonus update of its pheromone value

$$p_i \leftarrow p_i + \rho,$$

where ρ is the bonus pheromone value. The global update increases the chance of a fast grid service to be selected, and thus accelerates the convergence to the optimal grid path.
5. The pheromone values of grid services evaporate. At every period of time T_2, the pheromone value of every grid service is updated as

$$p_i \leftarrow \gamma p_i,$$

where $\gamma < 1$ is the evaporation constant.
6. When the pheromone value of a grid service is lower than some threshold value, τ, which usually means that this grid service has been unavailable for a long time or this grid service is an extremely slow with an undesired task completion time, the cleaner will remove it from the grid resource table.

In this swarm intelligence scheduling algorithm, variables T_1, T_2, θ, γ, and τ are tunable parameters subject to the specific grid computing environment.

5.6 Preliminary simulation results

To validate the effectiveness of the swarm intelligence scheduling algorithm, we simulate a computational grid with participant grid services exhibiting heterogeneous computational capabilities. Grid services have a variety of computational capabilities s, which indicates the number of test cases they can process at each time step, assuming that each test case requires roughly the same number of operations with an arrival rate conforming to a Poisson distribution. We assume that the computational capabilities of the grid service providers are normally distributed with mean, m and standard deviation, σ. We also assume that comparing to the execution time of a test task, the scheduling time, including decision time and data transportation time, of a testing task is trivial, and thus not considered in our simulation program. To introduce dynamism to the simulated grid, we allow the performance of a grid service change with a probability of ρ at each time step. The performance value of a grid service may change to 0, which indicates that the grid service leaves the grid-computing environment and its assigned testing task has to be rescheduled to an available grid service to rerun.

We compare the swarm intelligence scheduling approach with two widely used scheduling approaches, random scheduling and heuristic scheduling [31]. Random

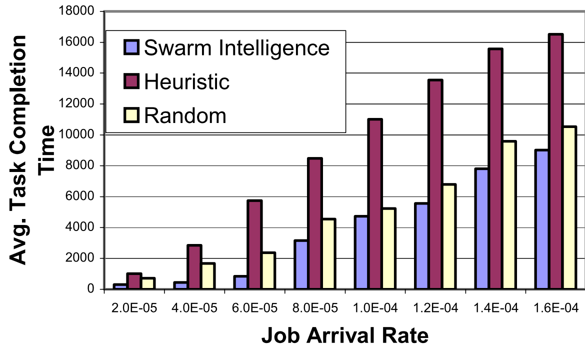

Fig. 3 Comparison of swarm intelligence mechanism, heuristic mechanism, and random mechanism in testing task scheduling on a simulated computational grid (performance changing probability $\rho = 0.0001$)

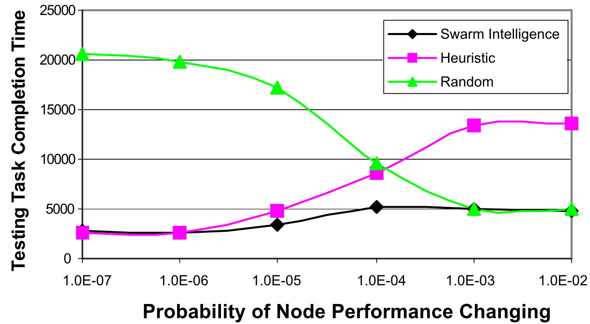

Fig. 4 Performance comparison of swarm intelligence mechanism, heuristic mechanism, and random mechanism in job scheduling on a simulated computational grid with different probabilities of grid service provider performance changing

scheduling approach has no extra information of the performance of a grid service, and thus schedules testing tasks to its grid services in a random manner. Heuristic scheduling approach considers the previous overall performance of a grid service provider in managing workload on different grid services. Figure 3 illustrates the testing task completion times verse task arrival rates on our simulated computational grid. At each time step, the performance of every grid service within the simulated grid changes with a probability of $\rho = 0.0001$. The swarm intelligence mechanism shows a better average task completion time than the random mechanism and the heuristic mechanism.

Figure 3 depicts the adaptability of the swarm intelligence scheduling mechanism. As the grid service provider performance changes probability, ρ, increases, and the simulated grid with the fixed task arrival rate evolves from a slightly dynamic system to a heavily dynamic system. The curves in Fig. 4 show that the performances of the heuristic mechanism and the random mechanism change dramatically; in contrast, the swarm intelligence mechanism exhibits a rather steady task completion time and yields almost the best task completion time in all these situations.

6 Trustworthiness enforcement

A computational grid is usually a potentially untrustworthy computing environment. For software testing on the grid, confidence must be obtained for testing cases carried

out on untrusted grid service providers. Noises generated by malicious or erroneous grid services may mislead the testing result analysis process and eventually increase the software testing cost.

In many grid-based applications, for example, Monte Carlo computation [32] or matrix computation [8], the problem itself can provide a way of validating the correctness of the computational results and an error caused by a grid service can be easily caught. Unfortunately, such a way does not exist in software testing on the grid because an error that deviates the computational results from the expected results may be caused by the malfunction of the grid service or a bug of the target code in the test case. As a result, consistency checking mechanisms, either internally or externally, must be used to obtain confidence for software testing on the grid. In our grid-based software testing framework, various mechanisms supporting consistency checking are employed to verify whether a test case is faithfully executed in the grid service and its test results are correctly transmitted. These trustworthiness enforcement mechanisms are described as follows.

(1) Redundant test cases.
Instead of submitting a single copy, redundant test cases are submitted to the grid and forced to be executed on different service providers. Inconsistency of test results indicates either a bug existing in the tested program or errors in the computation on the grid. At the same time, redundancy can also tolerate those test services that does not return test results due to unreliable network connections or temporary service unavailability [35].
(2) Check pointing and intermediate value checking.
In addition to checking the final results of a test case, intermediate values generated during the test process are saved at checkpoints in a test case and will be verified to ensure that the test task is faithfully executed.
(3) Separation of test case execution and test result comparison.
When performing software testing on the grid, a malicious grid service may fool the testers by simply supplying the expected results as the computational results and reporting "correctness" of the target code without actually executing the test case, if both test case execution and test result comparison are carried out on the same grid service. Our solution is to separate the test case execution and test result comparison. The software testing grid service only executes the test cases while the expected results are not provided. A designate trusted server is designated to retrieve the test results from the grid services and perform test result comparison. Using this mechanism, the grid service has to come up with its own computational result for a test case and a bogus result can be identified in the trusted server.
(4) Verification code injection.
One approach to check the validity of a test case carried out on a potentially untrustworthy grid service is to cleverly inject some verification codes into the beginning or the end of the target test unit program. To the grid services that run the test case, the computational result of the verification code is unknown until the subtask is actually executed. On the other hand, to the testers, the result of the verification is either preknown or easy to verify. A good candidate of this verification code is a program of calculating the inverse matrix [8]. A randomly

produced matrix **A** is produced and the verification code intends to compute the inverse matrix of $\mathbf{A}^{-1}$. Then if the product of **A** and $\mathbf{A}^{-1}$ is not equal to the identity matrix **I**, a malfunction in the grid service can be concluded.

7 Grid workflow for integration testing

An integration testing task on the grid is normally composed of a number of tasks—each is composed of the operations of multiple software units. A grid workflow is used to describe the execution of the integration testing task. Correspondingly, the testing workflow of an integration test case on the grid can as well be decomposed into smaller components. These components can be described as follows [33]:

- Testing unit: Testing units are the smallest elements in a grid workflow of an integration test case. Each testing unit executes the computing operations of a software unit and is usually carried out on an individual grid service.
- Sub-workflow: A sub-workflow is a flow of closely related testing units that is to be executed on the grid services within a virtual organization in an order predefined by the integration testing case. Each sub-workflow represents a specific task in the integration testing case. Multiple sub-workflows may be executed in parallel.

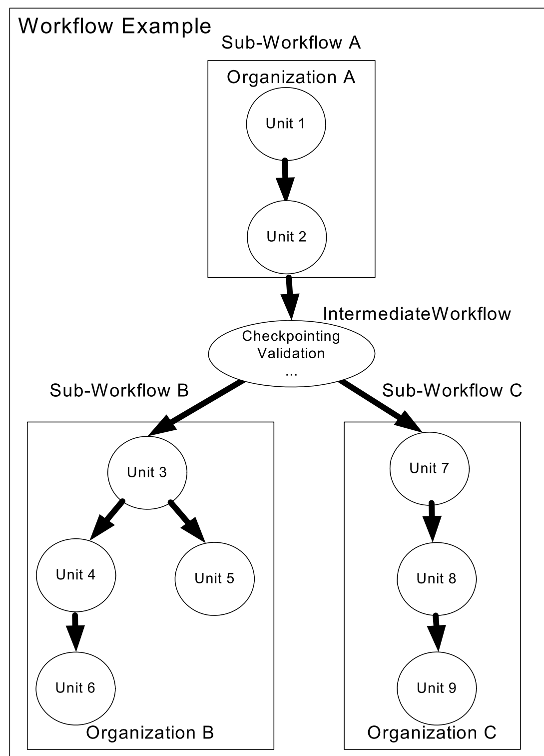

Fig. 5 Example of a grid-based integration testing workflow diagram with collaborations of 9 units

- Intermediate-workflow: An intermediate-workflow mediates sub-workflows running in different organizations. It carries out management tasks such as sub-workflows coordination, check pointing, computation consistency verification, and result validation operations.
- Workflow: A workflow can be represented as a flow of several loosely coupled activities described in an integration testing case. Each activity consumes various grid resources and can be represented by a sub-workflow.

The grid-based software testing framework takes advantage of the grid workflow management service to schedule the sub-workflows within a workflow to the appropriate target organizations. Then, testing units are executed on the grid resources within the organizations. The intermediate-workflow coordinates the execution of sub-workflows. XML (eXtensible Markup Language) is used to describe a grid workflow of integration testing case. Figure 5 illustrates an example of a grid workflow diagram and Fig. 6 shows the corresponding XML description. The workflow is decomposed into three sub-workflows with each sub-workflow to be scheduled on a grid organization specified by the "organization" tag. The "DataTransfer" tag spec-

```
<WorkFlow id = "mainworkflow">
  <SubWorkFlow id = "subworkflow1", order = 1>
    <Organization id = "organizationA"> </Organization>
    <DataTransfer> . . . < /DataTransfer>
    <Unit> description of unit tests 1 and 2
    </Unit>
    <DataTransfer> . . . < /DataTransfer>
  </SubWorkFlow>
  <IntermediateWorkFlow id ="intermediateworkflow1">
    <DataTransfer> . . . < /DataTransfer>
    <Operation> checkpointing, validation, . . .
    </Operation>
    <DataTransfer> . . . < /DataTransfer>
  </IntermediateWorkFlow>
  <SubWorkFlow id = "subworkflow2", order = 2>
    <Organization id = "organizationB"> </Organization>
    <DataTransfer> . . . < /DataTransfer>
    <Unit> description of unit tests 3, 4, 5, 6
    </Unit>
    <DataTransfer> . . . < /DataTransfer>
  </SubWorkFlow>
  <SubWorkFlow id = "subworkflow3", order = 2>
    <Organization id = "organizationC"> </Organization>
    <DataTransfer> . . . < /DataTransfer>
    <Unit> description of unit tests 7,8,9
    </Unit>
    <DataTransfer> . . . < /DataTransfer>
  </SubWorkFlow>
</WorkFlow>
```

Fig. 6 Workflow described in XML

ifies the I/O interface of each unit in the workflow. The “order” tag indicates the execution order of these sub-workflows, and the “unit” tag carries out the software unit operations.

8 Conclusion and future research direction

In this paper, we presented our ongoing project of a grid-based software testing framework by taking advantage of the large-scale and cost-efficient computational grid resources to build a testbed for accelerating software testing process and reducing testing cost in complex software systems. In this grid-based software testing framework, to achieve adaptability and efficiency of grid resources usage, a dynamic bag-of-tasks model using a swarm intelligence approach is developed to schedule unit test cases. Mechanisms supporting consistency checks are employed to obtain high confidence of each test case execution on a potentially untrustworthy grid. We also discuss using the workflow for describing the test task, the tasks scheduling mechanism, and the grid services to support grid-based software testing.

Currently, our grid-based unit test framework is an ongoing project. We have developed approaches and mechanisms discussed in this paper. Our presented preliminary results are based on simulations, which are just for “proof-of-concept.” At the next step, we must verify the practical feasibility of the grid-based software testing framework as well as our approaches and mechanisms to achieve adaptability and trustworthiness.

Acknowledgements This work is partially supported by the “Building an NCA&T Campus Grid Project” of the University of North Carolina General Administration and the NC-HPC Project of the University of North Carolina Office of the President.

References

1. Horgan R (1998) Panel statement: large scale software testing. In: Proceedings of 8th international symposium on software reliability engineering
2. Foster I, Kesselman C, Tueske S (2001) The anatomy of the grid. Int J Supercomput Appl 15(3):200–222
3. Goble C, Roure DD (2003) The grid: an application of the semantic web. In: Grid computing: making the global infrastructure a reality, pp 437–470
4. Foster I, Kesselman C, Nick JM, Tuecke S (2003) The physiology of grid: open grid services architecture for distributed systems integration (draft)
5. Duarte AN, Cirne W, Brasileiro F, Duarte P, Machado L (2005) Using the computational grid to speed up software testing. In: Proceedings of 19th Brazilian symposium on software engineering
6. Li Y, Dong T, Zhang X, Song Y, Yuan X (2006) Large-scale software unit testing on the grid. In: Proceedings of IEEE international conference on granular computing, Atlanta
7. Duarte A, Cirne W, Brasileiro F, Machado P (2006) GridUnit, software testing on the grid. In: Proceedings of ICSE
8. Beck M, Dongarra J, Eijkhout V, Langston M, Moore T, Plank J (2003) Scalable, trustworthy network computing using untrusted intermediaries: a position paper. DOE/NSF workshop on new directions in cyber-security in large-scale networks: development obstacles
9. SETI@home (2002) SETI@home: the Search for Extraterrestrial Intelligence. http://setiathome.ssl.berkeley.edu
10. Folding@home (2003) Distributed computing. http://folding.stanford.edu

11. distributed.net (2006) http://www.distributed.net
12. Pressman R (2005) Software engineering: a practitioner's approach. McGraw–Hill, New York
13. Foster I, Kesselman C (1997) Globus: a metacomputing infrastructure toolkit. Int J Supercomput Appl 11(2):115–128
14. Globus website (2005) http://www.globus.org
15. JUnit (2005) http://www.junit.org/index.htm
16. NUnit (2005) http://www.nunit.org
17. DbUnit (2005) http://dbunit.sourceforge.net
18. pyUnit (2005) http://pyunit.sourceforge.net
19. Fisher L (2002) Workflow handbook. Workflow Management Coalition
20. Bonabeau E, Théraulaz G (2000) Swarm smarts. Sci Am 282:72–79
21. Bivens HP (2001) Grid workflow. Grid computing environments working group. Global grid forum
22. Denebourg JL, Pasteels JM, Verhaeghe JC (1983) Probabilistic behavior in ants: a strategy of errors? J Theor Biol 105:259–271
23. Shapiro JA (1988) Bacteria as multicellular organisms. Sci Am 256:82–89
24. Fitzgerald TD, Peterson SC (1998) Cooperative foraging and communication in caterpillars. Bioscience 38:20–25
25. Di Caro G, Dorigo M (1998) Ant colonies for adaptive routing in packet-switched communications networks. In: Proceedings of fifth international conference on parallel problem solving from nature
26. Di Garo G, Dorigo M (1998) An adaptive multi-agent routing algorithm inspired by ants behavior. In: Proceedings of 5th annual Australasian conf para & real-time sys
27. Carriero N, Gelernter D, Leichter J (1986) Distributed data structures in Linda. In: Proceedings of 13th ACM symp on principles of programming languages
28. Andrews GR (1991) Concurrent programming: principles and practice. Benjamin–Cummings, Redwood City
29. Kuang H, Bic LF, Dillencourt MB (2002) Iterative grid-based computing using mobile agents. In: Proceedings of 31st IEEE international conference on parallel processing, ICPP2002
30. Sarmenta LFG (2001) Sabotage-tolerance mechanisms for volunteer computing systems. In: Proceedings of ACM/IEEE international symposium on cluster computing and the grid (CCGrid'01)
31. Wu M, Sun X (2003) A general self-adaptive task scheduling system for non-dedicated heterogeneous computing. In: Proceedings of IEEE intl conf on cluster computing
32. Li Y, Mascagni M (2003) Analysis of large-scale grid-based Monte Carlo applications. Int J High Perform Comput Appl 17(4):369–382
33. Li Y, Mascagni M (2004) E-science on the grid: toward a dynamic E-science automation with XML and workflow techniques. In: Proceedings of the 8th world multi-conference on systemics, cybernetics, and informatics, SCI'04, Orlando, Florida
34. Anderson DP, Cobb J, Korpela E, Lebofsky M, Werthimer D (2002) SETI@home: an experiment in public-resource computing. Commun ACM 45(11):56–61
35. Li Y, Mascagni M (2003) Improving performance via computational replication on a large-scale computational grid. In: Proceedings of the 3rd IEEE/ACM international symposium on cluster computing and the grid

Yaohang Li received his B.S. in Computer Science from South China University of Technology in 1997 and M.S. and Ph.D. degree from Department of Computer Science, Florida State University in 2000 and 2003, respectively. After graduation, he worked as a research associate in the Computer Science and Mathematics Division at Oak Ridge National Laboratory, TN. His research interest is in Grid Computing, Computational Biology, and Monte Carlo Methods. Now he is an assistant professor in Computer Science at North Carolina A&T State University.

Yong-Duan Song is a tenured professor at North Carolina A&T State University. He also holds the position of Langley Distinguished Professor at National Institute of Aerospace. His expertise lies in robotics/biomimetics, micro robotic UAVs/UGVs, adaptive and autonomous systems, bio-inspired guidance and control. He is an associate editor of the International Journal of Intelligent Automation and the Soft Computing and International Journal of Structure Health Monitoring. He also served as guest editor for the Journal of Robotic Systems and the International Journal of Wind Engineering and Industrial Aerodynamics.