Regular Paper

Performance Evaluation of A Testing Framework using QuickCheck and Hadoop

Yusuke Wada[†1] and Shigeru Kusakabe[†2]

Formal methods are mathematically-based techniques for specifying, developing and verifying a component or system, in order to increase the confidence regarding the reliability and robustness of the target. Formal methods can be used at different levels with different techniques, and one approach is to use model-oriented formal languages such as VDM languages in writing specifications. During model development, we can test executable specifications in VDM-SL and VDM++. In a lightweight formal approach, we test formal specifications to increase our confidence as we do in implementing software code with conventional programming languages. While the specific level of rigor depends on the aim of the project, millions of tests may be conducted in developing highly reliable mission-critical software in a lightweight formal approach. In this paper, we introduce our approach to supporting large volume of testing for executable formal specifications using Hadoop, an implementation of MapReduce programming model. We are able to automatically distribute interpretation of specifications in VDM languages by using Hadoop. We also apply a property-based data-driven testing tool, QuickCheck, over MapReduce so that specification can be checked with thousands of tests that would be infeasible to write by hand, often uncovering subtle corner cases that wouldn't be found otherwise. We observed effect to coverage and evaluate scalability in testing large amount of data for executable specifications in our approaches.

1. INTRODUCTION

Formal methods are mathematically-based techniques for the specification, development and verification of the target systems. Performing appropriate mathematical analysis of methods is effective in increasing the confidence regarding to the reliability and robustness of a design of the target system. We can choose a specific technique from various formal methods, such as model checking techniques.

†1 Graduate School of Information Science and Electical Engineering, Kyushu University
†2 Faculty of Information Science and Electrical Engineering, Kyushu University

Instead, in order to increase confidence in our specifications, we use adequately less rigorous means, such as testing executable specifications. We test executable specifications to increase our confidence in the specifications as we do in implementing software systems with conventional programming languages. While the specific level of rigor depends on the aim of the project, millions of tests may be conducted in developing highly reliable mission-critical software. For example, in an industrial project using VDM++, a model-oriented formal specification language[1)], they developed formal specifications of 100,000 steps including test cases (about 60,000 steps) and comments written in the natural language, and they carried out about 7,000 black-box tests and 100 million random tests[2)].

The attitude of coverage which represents the degree to which the source code has been tested, can be applied to executable specifications. We can get higher confidence in executable specifications if we have higher test coverage for the specifications. We expect higher coverage rate when we increase the number of test cases in lightweight formal approach.

In this paper, we discuss our approach to testing executable formal specifications, whose naive execution is rather expensive, for large volume of test data in an elastic way. We try to automatically distribute thousands of test runs of executable specifications in VDM languages over elastic computing resources by using Hadoop, an implementation of MapReduce programming model. Generating so large number of test data seems difficult to perform by hand, which can provide higher confidence by effectively uncovering subtle corner cases that might be overlooked by small test data. We apply a property-based data-driven testing tool over MapReduce so that we can generate large volume of test data, satisfying the pre-condition for the specification under test, in an efficient way. We use a property-based testing tool QuickCheck[3)], which was originally developed to support a high-level approach to testing Haskell programs by automatically generating random input data. We can distribute generation of large number of test data, as well as test-runs of executable specification for the large number of test data, and collect coverage information as well as test results from the distributed environment while observing scalable performance.

The rest of this paper is organized as follows. We briefly explain the VDM languages in Section 2. In Section 3, we discuss several issues to reduce the cost

of large amount of testing of VDM formal specification using emerging cloud technology. We outline our framework in Section 4 and evaluate our framework in Section 5. Finally we conclude in Section 6.

2. VDM: VIENNA DEVELOPMENT METHOD

VDM (The Vienna Development Method) is one of the model based formal methods, a collection of techniques for developing computer system from formally expressed models. While VDM was originally developed in the middle of 1970s at the institution of IBM in Vienna, its support tools, VDM Tools, are currently maintained by CSK Corporation in Japan. In order to allow machine-supported analysis, models have to be formulated in a well-defined notation. The formal specification language, VDM-SL, has been used in VDM, which became the ISO standard language (ISO/IEC 13817-1) in 1996, and VDM++ is its object-oriented extension version. The VDM Tools provide functionality of dynamic check of formal specifications in the formal specification languages VDM-SL and VDM++, such as interpretation of executable specifications, in addition to static checks such as syntax check and type check of formal specifications. By using the interpretor of VDM tools, we can test executable specifications in VDM-SL and VDM++ to increase our confidence in the specifications as we do in implementing software systems with conventional programming languages. There are no definitive usage pattern. This paper contains the example of a guideline to developing a formal model in VDM++[1)].

In this paper, we especially focus on the step, in which we validate the specification using systematic testing and rapid prototyping. In light-weight formal methods, we do not rely on highly rigorous means such as theorem proofs, and we use testing of executable specification in order to increase confidence in our specifications. While the specific level of rigor depends on the aim of the project, thousands of tests may be conducted in developing highly reliable mission-critical software. When we consider with a performance, it is time-consuming to execute the specification for large number of test data, and the performance degradation seems accelerated as the number of tests increases in this case.

3. LARGE AMOUNT OF TESTING FOR EXECUTABLE SPECIFICATION

Software testing plays an important role in gaining confidence for quality, robustness, and correctness of software. In this section, first we discuss software testing. Testing executable specifications share issues with software testing, such as cost and running time.

A Reducing the cost of testing

As the size and complexity of software increase, its test suite becomes larger and its execution time becomes a problem in software development. Several approaches have been used to reduce the cost of time consuming test phases. Selecting a representative subset of the existing test suite reduces the cost of testing[4)5)]. In prioritizing tests we execute test cases with higher priority earlier than lower priority ones[5)6)].

Large software projects may have large test suites. There are industry reports showing that a complete regression test session of thousands lines of software could take weeks of continuous execution[7)]. While each test is independent with each other, the very high level of parallelism provided by a computational grid can be used to speed up the test execution[8)]. Distributed tests over a set of machines aims at speeding up the test stage by simultaneously executing a test suite[9)10)]. A tool aims at executing software tests on Grid by distributing the execution of JUnit[11)] test suites over Grid, without requiring modification in the application and hiding the grid complexity from the user[8)].

VDMUnit is a framework for unit testing for VDM++ specifications, and it seems possible to distribute the execution of VDMUnit over Grid like JUnit over GridUnit. However, we consider an approach to leveraging the power of testing framework by using MapReduce[12)] to perform large amount of testing on elastic cloud computing platform rather than on Grid platform. In our approach, we will be able to automatically distribute the execution of testing specifications by using Hadoop over an elastic cloud computing platform. Using a cloud computing platform may also lower the cost of acquisition and maintenance cost of the test environment.

B Elastic platform

We consider an approach to leveraging the power of testing by using elastic cloud platforms to perform large scale testing. Increasing the number of tests can be effective in obtaining higher confidence, and increasing the number of machines can be effective in reducing the testing time. The cloud computing paradigm seems to bring a lot of changes in many fields. We believe it also has impact on the field of software engineering and consider an approach to leveraging light-weight formal methods by using cloud computing which has the following aspects[13]:

(1) The illusion of infinite computing resources available on demand, thereby eliminating the need for cloud computing users to plan far ahead for provisioning;
(2) The elimination of an up-front commitment by cloud users, thereby allowing organizations to start small and increase hardware resources only when there is an increase in their needs; and
(3) The ability to pay for use of computing resources on a short-term basis as needed and release them as needed, thereby rewarding conservation by letting machines and storage go when they are no longer useful.

We can prepare a platform of arbitrary number of machines and desired configuration depending on the needs of the project.

C Property-based data-driven testing

Since increasing the number of tests is effective in obtaining higher confidence, huge number of tests may be performed especially in developing mission-critical software. Conceptually, we can increase the number of test cases on elastic cloud computing platforms. However, generating test cases by hands can be a bottleneck in software development.

In this paper, we use a property-based testing tool QuickCheck[3], which supports a high-level approach to testing Haskell programs by automatically generating random input data. Property-based data-driven testing encourages a high level approach to testing in the form of abstract invariants functions should satisfy universally, with the actual test data. Code can be checked with thousands of tests that would be infeasible to write by hand, often uncovering subtle corner cases that would not be found otherwise. We try to automatically distribute the generation of test data for formal specification in addition to the execution of formal specification.

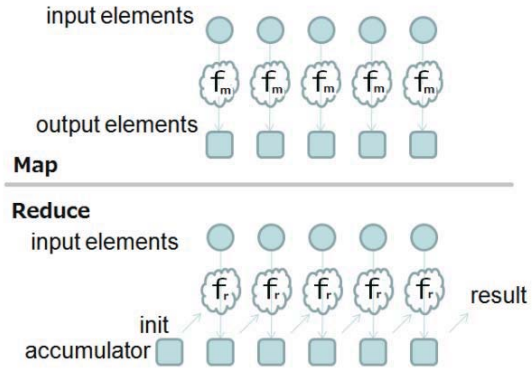

Fig. 1 Concept of map/reduce programming model.

D MapReduce

While we can prepare a platform of an arbitrary number of computing nodes and generate an arbitrary number of test cases, we need to reduce the cost of managing and administrating of the platform and runtime environment.

MapReduce programming model is proposed in order for processing and generating large data sets on a cluster of machines[12]. Input data-set is split into independent elements, and each mapper task processes the corresponding element in a parallel manner as shown in **Fig. 1**. Data elements are typically data chunks when processing huge volume of data. The outputs of the mappers are sorted and sent to the reducer tasks as their inputs. The combination of map/reduce phase has flexibility, thus, for example, we can align multiple map phases in front of a reduce phase.

MapReduce programs are automatically parallelized and executed on a large cluster of machines. The runtime system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. Its implementation allows programmers to easily utilize the resources of a large

distributed system without expert skills for parallel and distributed systems.

When using this map/reduce framework, input elements can be test cases, `f` can be an executable specification in VDM languages or actual code fragment under test, and output elements test results, contains test coverage of an executable specification.

4. OUR APPROACH

Increasing the number of tests can be effective in obtaining higher confidence, and increasing the number of machines can be effective in reducing the testing time. Regarding the number of test cases, preparing an arbitrary large number of test cases by hand is possible but impractical. Among many tools for testing, a property-based testing tool, QuickCheck, supports a high-level approach toward testing Haskell programs by automatically generating random input data as described later. We modify QuickCheck to fit to our approach for testing formal specification with Hadoop. As formal specification languages, such as VDM-SL for example, share features with functional programming languages, we can obtain formal description necessary to use QuickCheck to generate test data in VDM-SL. There have been work focusing on their relations[14)15)].

We consider an approach to use QuickCheck on elastic cloud platforms. We can perform testing of arbitrary scale by exploiting such a combination. We can automatically distribute the generation of test data and the execution of tests in a scalable manner with Hadoop. We employ Hadoop framework to easily execute tests in a data-parallel way.

A QuickCheck

QuickCheck is an automatic testing tool for Haskell programs. It defines a formal specification language to state properties. Properties are universally quantified over their arguments implicitly. The function `quickCheck` checks whether the properties hold for randomly generated test cases when they are passed as its arguments. QuickCheck has been widely used and inspired related studies[16)17)18)].

For the explanation, we use a simple `qsort` example from a book[19)].

```
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort lhs ++ [x] ++ qsort rhs
      where lhs = filter  (< x) xs
            rhs = filter (>= x) xs
```

We use idempotency as an example invariant to check that the function obeys the basic rules a sort program should follow. Applying the function twice has the same result as applying it only once. This invariant can be encoded as a simple property. The QuickCheck convention in writing test properties is prefixing with `prop_` to distinguish them from normal code. This idempotency property is written as a following Haskell function. The function states equality that must hold for any input data that is sorted.

```
prop_idempotent xs = qsort (qsort xs) == qsort xs
```

QuickCheck generates input data for this `prop_idempotent` and passes it to the property via the `quickCheck` function. Following example shows the property holds for the 100 lists generated.

```
> quickCheck (prop_idempotent :: [Integer] -> Bool)
OK, passed 100 tests.
```

While the sort itself is polymorphic, we must specify a fixed type at which the property is to be tested. The type of the property itself determines which data generator is used. The `quickCheck` function checks whether the property is satisfied or not for all the test input data generated. QuickCheck has convenient features such as quantifiers, conditionals, and test data monitors. it provides an embedded language for specifying custom test data generators.

Conceptually, we can evaluate property expressions in property-based random testing in a data-parallel style by using MapReduce framework. Each mapper evaluates the property for one of the test data and reducer combines the results from mappers. By applying an automatic testing tool such as QuickCheck on MapReduce framework, we expect we can greatly reduce the cost of a large scale testing.

B Implementation

We developed our testing environment by customizing QuickCheck and developing glues to connect components for testing executable specifications on Hadoop framework. In this section, we discuss implementation issues.

Fig. 2 outlines this approach. Our approach to implementing property-based testing on Hadoop is to separate the testing process into two phases. We generate

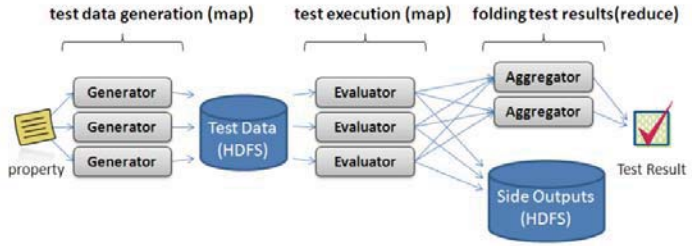

Fig. 2 Outline of our approach to propeerty-based data-driven testing.

test data by using mappers, and store the data into a file in the first phase. Then we can read and split the file, and distribute the data to mappers, where the property function written in Haskell is evaluated.

Hadoop streaming: Hadoop, open source software written in Java, is a software framework implementing MapReduce programming model[20]. We write mapper and reducer functions in Java by default in this Hadoop framework. However, Hadoop distribution contains a utility, Hadoop streaming, which allows us to create and run jobs with any executable or script as the mapper and/or the reducer. The utility will create a mapper/reducer job, submit the job to an appropriate cluster, and monitor the progress of the job until it completes. When an executable is specified for mappers, each mapper task will launch the executable as a separate process when the mapper is initialized. When an executable is specified for reducers, each reducer task will launch the executable as a separate process when the reducer is initialized. This Hadoop streaming is useful in implementing our testing framework.

Fig. 3 shows an overview of our property-based data-driven testing. The overview of Property-based data-driven testing is as follows: First, we generate test data according to the specified property. Next, we store the generated data in a file on HDFS. Finally, in the evaluation phase, we pass the test data to mappers through the standard input, and the mappers output the results to the standard output.

Distribution of test data generation: We generate the specified number of random test data with mappers in Hadoop framework in a distributed way. However,

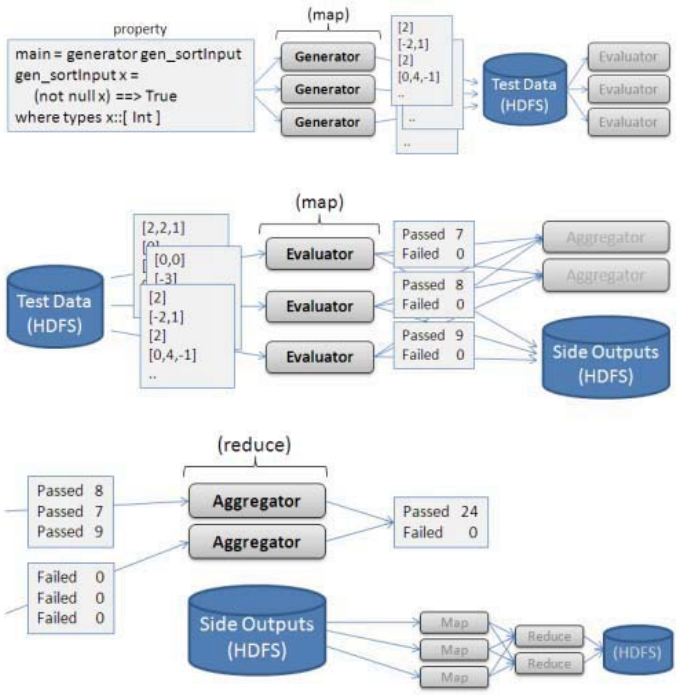

Fig. 3 Progress of property-based data-driven testing.

naively splitting the number and assigning the sub-numbers to mappers lead to useless computing due to overlap of test data generated among different mappers. We need to avoid increasing the number of redundant test data from the view point of efficiency and coverage. We modified the generator in QuickCheck to avoid this problem. We add one extra parameter to `check` function in QuickCheck module. The parameter represents the start index of test data and is passed to `test` function. After the total number of tests is determined, each mapper is given different start index according to the number of tests assigned to mappers.

5. PERFORMANCE EVALUATION

In order to examine the effectiveness of our approach, we measured performance

Table 1 The ratio of unique data and standard deviation in generating random data of Int and float

		Original	Native	Ours
Int	Unique(%)	34.5	8.3	34.4
	Std.Dev.	7754.1	966.3	7656.2
Float	Unique(%)	99.6	95.7	99.6
	Std.Dev.	7684.5	962.0	7729.8

in testing specification of Enigma in VDM++ written in the book[1]. The Enigma cipher machine is basically a typewriter composed of three parts: the keyboard to enter the plain text, the encryption device and a display to show the cipher text. Both the keyboard and the display consist of 26 elements, one for each letter in the alphabet.

A Distribution of test data generation

In order to see the effectiveness of our distribution of test generation, we compare the number of unique (non-redundant) test data in generating 80000 data for `Int` and `Float` type. Table 1 is the result. "Original" means using the QuickCheck original data generator on non-Hadoop environment. "Naive" means each data generator for subset starts its index from 1 in generating random test data on mappers in Hadoop environment. "Ours" means each data generator knows its own starting index, each of which is different from each other. According to the result, we can see effectiveness of our modification as we see no outstanding difference between "Original" and "Ours" while we have some degradation in "Naive".

B Coverage

Higher test coverage for a software component leads to the higher confidence in the component, while coverage rate represents the degree to which the source code has been tested. This also applies to our approach. We can get higher confidence in executable specifications if we have higher test coverage for the specifications. Getting higher coverage is important in our approach, since our approach is a kind of light-weight formal methods that rely on testing, not formal proof, in gaining confidence in a formal specification. We expect higher coverage rate when we increase the number of test cases in our property based approach.

We examine the effectiveness of our approach. VDMTools can report coverage of an executable specification, and we can gather coverage data in our approach as described later. We can observe the change in coverage rate while we change the number of test cases by changing the parameter value of QuickCheck.

We examined impact of changing test cases on coverage rate in our approach through the following steps:

(1) We used VDM++ executable specification files for enigma cipher.
(2) We performed property-based data-driven test, in which we generate test cases by using customized QuickCheck and executed specifications with VDMTools through command line interface.
(3) Observe the VDM++ class coverage while we alter amount of input test data.

We run our property-based test ten times for each configuration. We changed the number of test cases as one, ten, one hundred and one thousand. **Fig. 4** shows the results for Rotor.vpp in this experiment.

By using one of the features of VDMTools, we collected coverage information through this experiment. As we can see from the figure, the larger number of test cases we use, the higher coverage we have, and the specification file is covered completely when we use one thousand of test cases.

However, it took long time to execute the specification files for one thousand of test cases. Next, we examine the effectiveness of parallel and distributed execution of formal specifications using Hadoop.

C Speedup in Property-based Testing

The configuration of the platform is shown in Table Table 2. We show the result of elapsed time in Table 3 and in **Fig. 5**. As we can see from the results, the elapsed time of Hadoop version became shorter when the number of tests was four hundreds and over. Since Hadoop framework is designed for large scale data processing, we have no advantage in elapsed time for small set of test data. The computation node has four processor cores, and we can achieve speedup even on a single node as Hadoop can exploit thread-level parallelism on multi-core platforms. **Fig. 6** shows the scalability with the changing number of nodes. The speedup ratio is calculated against the result of the sequential execution for the tests on a single node. It means the higher the ratio, the more the platform is of advantage in our testing. As we see in Fig. 6, the increase of the number of slave

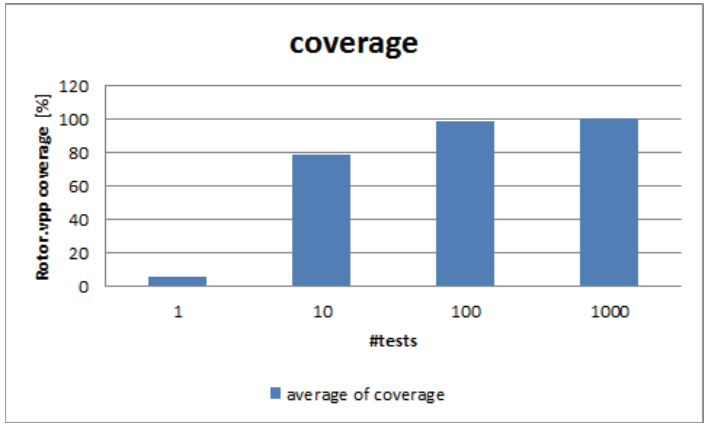

Fig. 4 Coverages of four configuration of patterns in ten trial.

Table 2 Configuration of the platform in performance evaluation.

	NameNode	JobTracker	Slave
CPU	Xeon E5420[1]	Xeon E5420[1]	Xeon X3320[2]
Memory	3.2GB	8.0GB	3.2GB
Disk	2TB	1TB	140GB

Table 3 Elapsed time in tests of Enigma specification

Nodes/Tests	100	200	300	400	500	600	800	1000
Single	21.1	67.4	158.1	334.5	492.1	672.4	1392.3	2285.5
1	232.9	230.1	272.4	267.0	246.2	262.0	369.9	434.4
2	163.8	160.4	179.3	183.2	168.1	160.1	233.5	285.7
3	126.4	120.2	123.1	127.3	123.9	126.2	157.2	211.1
4	102.9	104.1	104.4	106.1	110.0	114.0	135.1	170.7
5	91.2	90.2	93.9	95.6	99.0	100.0	119.7	145.5
6	86.4	84.7	88.5	88.6	93.5	94.4	110.9	136.0
7	90.3	84.8	85.2	84.4	86.9	91.5	104.4	127.4
8	83.8	78.4	79.4	79.9	83.0	83.9	99.2	117.6

machines is effective in reducing testing time.

1 Intel(R) Xeon(R) CPU EL5410 @ 2.50GHz Quad Core
2 Intel(R) Xeon(R) CPU X3320 @ 2.50GHz Quad Core

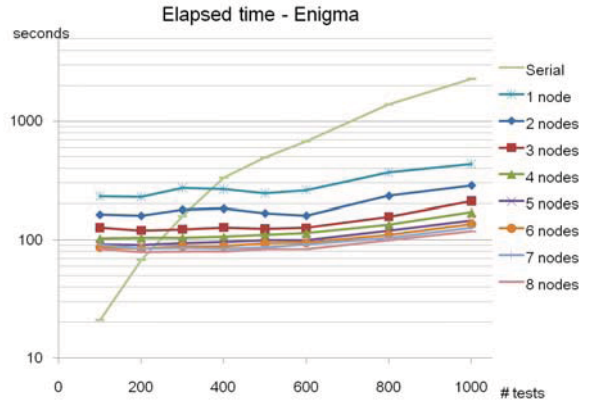

Fig. 5 Elapsed time in increasing the number of tests of VDM++ enigma specification on the various number of nodes.

6. Concluding Remarks

In this paper, we explained our approach to testing executable formal specifications in lightweight formal method framework using VDM languages. In order to increase confidence in the specification, we increase the number of test cases with a property-based data-driven approach on a cloud computing oriented programming framework. We apply a property-based data-driven testing tool, QuickCheck, so that specification can be checked with hundreds of tests that would be infeasible to write by hand. We investigated coverages of executable VDM++ model. We can increase of high coverage possibilities by multiplying test data. However, large amount of test data gains long executing time. Our framework can deal with this probrem. We observed scalable performance in conducting large amount of testing for executable specifications. Therefore, we can both of acquiring high coverage because of large amount of test data and tuning our platform so that execute the test in time.

As one of future work, we will investigate more detailed performance breakdown to achieve more efficient environment. We will also try to improve usability of our framework. VDMTools include test coverage printout tool. We will extend

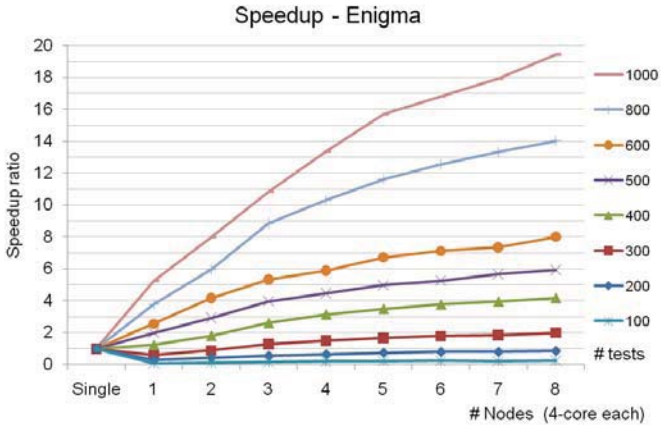

Fig. 6 Speedup ratio in increasing the number of tests of VDM++ enigma specification on the various number of nodes. Please note each node has a 4-core processor.

our framework to exploit this tool in a parallel and distributed way to inform detail coverage information, such as a VDM++ statement covered but another is not, for more usability.

References

1) Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M. and Fitzgerald, J.: Validated Designs For Object-oriented Systems, *Springer Verlag* (1998).
2) Kurita, T., Chiba, M. and Nakatsugawa, Y.: Application of a formal specificaton language in the development of the mobile felica IC chip firmware for embedding in mobile phone, *FM 2008: FORMAL METHODS*, pp.425–429 (2008).
3) Claessen, K. and Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs, *ACM SIGPLAN Notices*, pp.35(9):268–279 (2000).
4) Graves, T.L., Harrold, M.J., Kim, J.-M., Porter, A. and Rothermel, G.: An empirical study of regression test selection techniques, *ACM Transactions on Software Engineering and Methodology*, pp.10(2):184–208 (2001).
5) Wong, W., J.R.Horgan, London, S. and Agrawal, H.: A study of effective regression testing in practice, *Proceedings of the Eight International Symposium on Software Reliability Engineering* (1997).
6) Kim, J.-M. and Porter, A.: A history-based test prioritization technique for regression testing in resource constrained environments, *Proceedings of the 24th International Conference on Software Engineering* (2002).
7) Elbaum, S., Malishevsky, A.G. and Rothermel, G.: Prioritizing test cases for regression testing, *In Proceedings of the International Symposium on Software Testing and Analysis*, pp.102–112, ACM Press (2000).
8) Duade, A., Cirne, W., Brasileiro, F. and Macado, P.: Gridunit: Software testing on the grid, *In Proceedings of the 28th ACM/IEEE International Conference on Software Engineering*, Vol.28, pp.779, ACM (2006).
9) Kapfhammer, G.M.: Automatically and transparently distributing the execution of regression test suites, *In Proceedings of the 18th International Conference on Testing Computer Software* (2001).
10) Hughes, D., Greenwood, P. and Coulson, G.: A framework for testing distributed systems, *In Proceedings of the 4th IEEE International Conference on Peer-to-Peer computing(P2P'04)* (2004).
11) Gamma, E. and Beck, K.: Junit: A cook's tour, *Java Report*, pp.4(5):27–38 (May 1999).
12) Dean, J. and Ghemawat, S.: MapReduce: simplified data processing on large clusters, *Commun ACM*, pp.51(1):107–113 (January 2008).
13) Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I. and Zaharia, M.: Above the clouds: A berkeley view of cloud computing, *Technical report, UCB/EECS-2009-28, Reliable Adaptive Distributed Systems Laboratory* (February 2009).
14) Borba, P. and Meira, S.: From vdm specifications to functional prototypes, *J. Syst. Softw*, pp.21(3):267–278 (June 1993).
15) Visser, J., Oliveira, J.N., Barbosa, L.S., Ferreira, J.F. and Mendes, A.S.: Camila revival: VDM meets haskell, *First Overture Workshop* (2005).
16) Arts, T., Hughes, J., Johansson, J. and Wiger, U.: Testing telecoms software with quviq quickcheck, *ERLANG'06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pp.2–10, New York, NY, USA (2006).
17) Boberg, J.: Early fault detection with model-based testing, *Erlang Workshop*, pp. 9–20 (2008).
18) Claessen, K., Palka, M., Smallbone, N., Hughes, J., Svensson, H., Arts, T. and Wiger, U.: Finding race conditions in erlang with quickcheck and pulse, *ICFP*, pp. 35(9):268–279,2000 (2009).
19) O'Sullivan, B., Goerzen, J. and Stewart, D.: Real World Haskell, *Oreilly & Associates Inc* (2008).
20) Apache: Hadoop, `http://hadoop.apache.org/core/` (As of Jun.1, 2009).