ISSN 1673-9418 CODEN JKYTA8
Journal of Frontiers of Computer Science and Technology
1673-9418/2008/02(04)-0418-13
DOI:10.3778/j.issn.1673-9418.2008.04.008

E-mail:fcst@public2.bta.net.cn
http://www.ceaj.org
Tel:+86-10-51616056

A Testing and Evaluation Approach for Discovering and Ordering of Software Entities for Internetware*

CAI Shubin[1,2], MING Zhong[2+], LI Shixian[1]

1. Department of Computer Science, SUN Yat-Sen University, Guangzhou 510275, China
2. Faculty of Information Engineering, Shenzhen University, Shenzhen, Guangdong 518060, China

+ Corresponding author: E-mail: mingz@szu.edu.cn

网构软件中实体发现和排序的TEA方法*

蔡树彬[1,2],明　仲[2+],李师贤[1]

1. 中山大学 计算机科学系,广州 510275
2. 深圳大学 信息工程学院,广东 深圳 518060

摘　要:自治软件实体分布在开放、动态和多变的互联网中,它们的协作构成网构软件的基础。动态连接模式是网构软件重要的协作方式。在该模式下,如何选择具有所需质量的软件实体,是一个非常困难的任务。介绍了运行时测试和自动化测试技术,提出网构软件的自动化运行时测试方法。总结网构软件中用于软件实体的发现和排序的解决方法,提出了测试和评估方法TEA。网构软件不知道候选软件实体的质量,通过进行自动化的运行时测试,TEA可以选择具有更高质量的实体。如同网构软件一样,自治软件实体也可以在运行时改变自身结构。通过使用确认断言来评估每次实体调用的结果,TEA可以尽早发现实体变化引起的质量问题。TEA中的软件实体注册机收集每个不同的网构软件反馈的测试和评估结果信息,可以产生按估计质量排序的实体列表。在模拟实验中,TEA产生的有序列表获得最好的评估分数。

关键词:网构软件;自动化测试;运行时测试;正确性;可靠性

文献标识码:A　　**中图分类号:**TP311

CAI Shubin, MING Zhong, LI Shixian. A testing and evaluation approach for discovering and ordering of software entities for internetware. Journal of Frontiers of Computer Science and Technology, 2008,2(4):418-430.

* the National Natural Science Foundation of China under Grant No.60673122 (国家自然科学基金); the Technology Plan Project of Shenzhen City under Grant No.200731 (深圳市科技计划).

Received 2008-03, Accepted 2008-06.

Abstract: Internetware is built upon the collaboration of autonomous software entities distributed in the open, dynamic and ever-changing internet. It is very difficult to find out software entities with needed quality in the dynamic connection pattern of internetware. Automated runtime testing is proposed after runtime testing and automated testing techniques are investigated. Several approaches to discover and order software entities for internetware are discussed before the Testing and Evaluation Approach (TEA) is proposed. Since qualities of candidate entities of internetware remain unknown, automated runtime testing is performed in TEA to select entity with higher quality. The autonomous composing entities may change their structure at runtime as well as internetware. Each invocation result is evaluated in TEA by validation assertions, to find out quality problems caused by entity changes as soon as possible. Software entity registry in TEA gathers feedback information of testing and evaluation results from each distinct internetware, and generates the ordered-list of entities by estimated quality. The ordered-list generated by TEA has the best scores in the simulation experiment.

Key words: internetware; automated testing; runtime testing; correctness; reliability

1 Introduction

As the internet keeps rapid and continuous development, it becomes more and more popular and important in persons, organizations and societies' daily life. Usually speaking, nomads are a nation on horse back, U.S.A is a country on wheel. Now, the Earth is becoming a globe on internet. Facing the open, dynamic and ever-changing internet, a new software paradigm named internetware, is proposed[1-2]. Frontier researches[3-6] have been carried out and internetware is coming into being.

From the point of view of software development, basic entities, which compose a software system, change from functions to objects, from objects to components and from components to services. The granularity grows bigger and bigger. Better open and dynamic features of a system often result in lower computational performance. The bigger granularity enables a worthy preference for open and dynamic features. Compared with traditional software system, internetware has a self-adaptive ability to change itself dynamically according to environment changes. The composing entities of internetware, mainly individual services, are openly published and autonomously maintained at different internet nodes. Internetware coordinates these entities to form a Software Web, similar with the current Information Web, to satisfy user's requirements. These lead to new features of internetware mentioned as Collaboration, Autonomy, Context-awareness, Evolution and Polymorphism[1-2].

(1)Collaboration. Traditional software has a static connection pattern among its composing entities in a closed and centralized environment. In addition to this pattern, internetware can have a dynamic connection pattern to accommodate the open and dynamic internet environment. It is the dynamic connection pattern that enables the autonomy feature.

(2)Autonomy. Entities of internetware can be developed, published and maintained independently and autonomously at different internet nodes. Traditionally, a software composing entity has already been well recognized before the whole system comes into being. However, internetware can dynamically

choose a newly-published entity owned by third party to satisfy user's new requirements, or to adapt the ever-changing internet environment.

(3)Context-awareness. This feature makes it possible for internetware to evolve according to environment changes. When runtime environment, such as network condition, or autonomous composing entity, or predicted and permitted user requirement changes, internetware shall perceive these changes and provide appropriate information for evolution.

(4)Evolution. It is this feature that gives internetware the ability to accommodate the open, dynamic and ever-changing internet environment. Internetware evolves according to requirement and environment changes. It is incarnated at changeable composing entities, adjustable architecture relationships and configurable architecture shapes.

(5)Polymorphism. Polymorphism in OOP means a function with different implementations. Polymorphism of internetware means an internetware with different runtime implementations. The "same" internetware can have different entities at different internet nodes to satisfy different user requirements. The implementation differences are probably caused by evolutions, or in other word, self-adaptability.

To sum up, the dynamic connection pattern of collaboration enables the autonomy feature of internetware, collaboration and autonomy together lead to evolution according to context changes, and self-adaptive evolution brings polymorphism. With these features, internetware solidly grounds itself in the open, dynamic and ever-changing internet.

As a new software paradigm, internetware brings many challenges for software development methods and techniques. Architecture-based component composition (ABC) approach[3] is introduced to support the engineering of internetware. ABC is mainly concerned with the dynamic connection pattern and component composition. It has the ability to structure the chaotic software entities to ordered internetware in a bottom-up style and enable the development of self-adaptive internetware. Testing is the most important part of software quality assurance, but it is very difficult to test the ever-evolving internetware with traditional testing techniques[7]. Thus a very important problem arising for internetware is the correctness and reliability problem. In order to develop internetware with high correctness, reliability and customer satisfaction, pioneer researches have been carried out. An application semantic based relaxed transaction model[4] is proposed to make the composition more reliable. The candidate software entities for internetware form a chaos. It is not easy to choose entity with needed quality from such a chaos. Employing a trust measurement and evolution model[5] to select more trustable software entities, will improve the reliability of internetware at lower level. Furthermore, new fault tolerance technologies for internetware[6] can be adopted to improve the reliability at higher level.

How to judge an invocation success as expected at runtime, is the fundamental part of the trust model[5] but has not been addressed. In this paper, a Testing and Evaluation Approach (TEA) is proposed. Automated runtime testing is employed in TEA, and testing and evaluation information from every internetware is gathered by entity registry to generate a recommendation ordered-list of entities. Being the most fundamental approach to select software entity with needed quality, TEA can be integrated with all the above solutions to solve the correctness and reliability problem of internetware.

The rest of the paper is organized as follows. Section 2 introduces several runtime testing and au-

tomated testing techniques, which can be combined as automated runtime testing for internetware. Despite of technical details, Section 3 and 4 discuss conceptual approaches to discover and order candidate software entities. Section 5 gives simulation experiment design and the experiment results. Section 6 concludes the paper.

2 Automated Runtime Testing

Testing is often performed at development-time in traditional software development, since the make up of a system is more or less fixed at that time. With the progress of component-based software development, however, the notion of testing "the system" as an integrated whole at development-time is no longer applied in the traditional sense, because the structure of a component-based system can change even in runtime. Development-time testing alone is not enough to make sure that the system can run as expected. Runtime testing is proposed to deal with this problem. One of the most promising ways in runtime testing is the notion of Built-In Testing (BIT)[8-11], first suggested by Wang[8], later refined in Component+ project[9], and now evolved in MORABIT project[11]. The basic idea behind this approach is to give components the ability to test their environments at runtime so that they can perform much of the required system validation work "themselves." For internetware, which can be regarded as an ever-changing component-based system, runtime testing is very beneficial. For example, suppose an entity A is going to be replaced by entity B or C, when internetware evolves automatically according to environment change, by performing runtime testing on B and C, internetware can choose entity with higher success-rate and provide more reliable service. However, without support from automated testing, runtime testing will ask for human interaction and is not suitable for self-adaptive internetware.

Traditionally human engineers are responsible for putting the system into an appropriate state, judging when tests can and should be executed, analyzing the results to identify unexpected behavior and working out how to respond. In order to perform automated runtime testing for internetware, highly automated testing techniques should be adopted. Recent years progress has been seen with the advent of automated testing[12-14], which can be roughly classified into two aspects of automation: automated test cases and oracles generation, where oracles are the criteria to determine whether a test run has succeeded, and automated test execution and management.

Automated generation of test cases and oracles[12,13] is often enabled by the design by contract approach. Contracts use pre-condition, post-condition and invariant to state what conditions the software must meet at certain points of the execution. In practice, pre-conditions tend to be exhaustive, while post-conditions and invariants are more or less extensive depending on the developer's style, i.e., contracts are partial. There may be cases, which satisfy the con tracts but are incorrect from the users' point of view.

Automated testing frameworks[13,14] focus on test management and execution. Given a testing description, which contains test cases and oracles, the frameworks perform the testing process automatically. Failure recovery, which allows testing to continue even if a test case causes execution to fail, is an important issue in the framework. Automated regression testing, i.e., to rerun an appropriate subset of earlier recorded test cases after a change happened, is very useful to deal with software evolution and maintenance.

Combining runtime testing[8-11] and automated testing[12-14] together, the automated runtime testing for internetware becomes possible. Because of the limitation of time and resources, and the large amount of candidate entities, it's not practical and valuable to test every entity extensively. If an entity or its provider is trustable[5], i.e., internetware believes that the entity has undergone extensive development-time testing in which the normal coverage and other criteria were used to define test cases, typical test cases can be designed with a view to uncovering the most likely causes of misunderstandings between entity provider and consumer. Since the goal of this testing is to check whether entity meets internetware's expectations rather than its specification, the testing is driven more from the perspective of validation than of verification, and can be called as automated runtime validation testing. It is easy for internetware developer or automated technique to design typical test cases from requirement specifications, scenarios or use cases of the internetware. Otherwise, if trust of an entity or its provider is unknown, automated runtime verification testing, where time and resources cost should be carefully considered, should be carried out. Fault injection and other testing techniques may be adopted in automated runtime verification testing.

With the support from Built-In Testing[8-11] or transaction[4], automated runtime testing framework can be designed straightforwardly and integrated into internetware's runtime environment. It should be noted that however, if E-Business related software entity, such as hotel reservation, flight-ticket and book purchase etc, supports neither BIT nor transaction, the automated runtime testing can't be performed.

3 Discovering and Ordering Software Entities for Internetware

Since software entities of internetware are distributed in the internet, there is an entity discover and retrieval problem. Firstly, an entity description language EDL should be designed to represent entity, such as WSDL for web service. But the design of an expressive EDL is far beyond the scope of this paper, we simply assume that EDL exists. The self-adaptive internetware asks its runtime environment for software entities described in EDL, and the runtime environment will use the description to discover matching entity in the internet. As there will be many matching entities with the progress of internetware, the concentration is paid to the generation of an ordered-list of matching entities by estimated quality.

Despite of technical details, existing approaches to deal with the discovering and ordering (or selecting) problem can be roughly summarized as below.

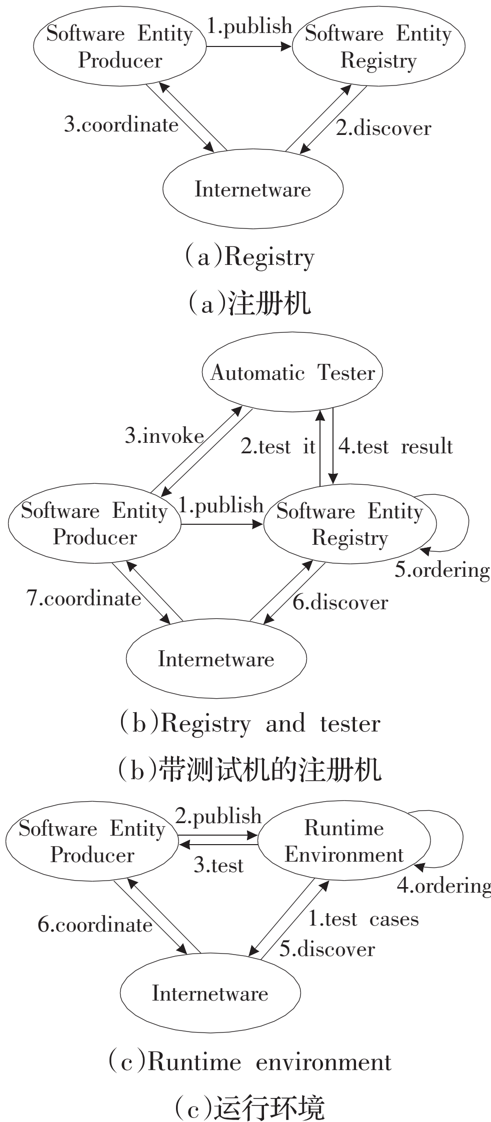
3.1 Registry

Using software entities registry, as well as service registry, is the most direct way to discover software entities in the internet. Software entity is published and indexed at software entity registry. As shown in Fig.1(a), when internetware need some software entities, it communicates with the registry to find matching ones. Then it binds and invokes the needed one to accomplish the task. This approach is widely adopted in Service-Oriented Architecture. But the problem of how to select the "right" one from the matching entities remains unsolved.

3.2 Registry and Tester

By performing testing on registered entity, entity quality can be estimated and ordered-list can be

generated. Generally speaking, the third party registry or tester has little specific knowledge about the entities, only automated assertion testing[12-14] can be carried out. Though the given assertions are partial, they are still very helpful to estimate and order entity quality. As shown in Fig.1(b), the automated assertion testing is performed before internetware queries the registry for matching entities. Internetware retrieves the ordered-list of matching entities from registry, and chooses the needed one according to its requirement of reliability and cost etc. This approach can be improved with "real" runtime environment, where the entities are going to settle in, and "real" user requirement, where the entities are going to satisfy.



(a)Registry

(a)注册机

(b)Registry and tester

(b)带测试机的注册机

(c)Runtime environment

(c)运行环境

Fig.1 Registry, registry and tester, runtime environment approaches

图1 注册机、带测试机的注册机和运行环境方法

3.3 Runtime Environment

Let runtime environment manage all the candidate software entities is another approach[15]. In this case, software entities are submitted to runtime environment, control on software entities to some extent is earned[3]. Thus internetware using this approach can be called as half-autonomy internetware. As shown in Fig.1(c), internetware gives test cases and oracles to its runtime environment. Automated testing on matching entities is carried out by the environment. The very shortcoming of this approach is the lack of a registry. Though every internetware can minimize the testing cost themselves[15], since they don't share information of testing result, an entity can be tested many times by different internetware, the overall cost is too high. By employing a registry to gather and share feedback information of testing result, the overall testing cost can be greatly reduced.

4 Two New Approaches

In product-purchasing market, consumers check the product before purchasing and evaluate the product after using. By gathering consumers' feedback information, the market can estimate the quality of the product and finally form an ordered-list of products of the same kind by estimated quality. Consumers can benefit greatly from the ordered-list when they are going to purchase new products. Feedback information is very important for the market to form the ordered-list and beneficial for consumers' product-purchasing. The same things will

probably happen in the inter netware and entity "market". Applying the product-purchasing idea in the generation of ordered-list of entities by estimated quality of registry, approaches in Section 3.2 and 3.3, and feedback information are combined to form a hybrid testing approach below.

4.1 Hybrid Testing Approach

As shown in Fig.2(a), registry and tester are adopted to produce the original ordered-list by estimated quality of software entities at first. Internetware still gives test cases and oracles to its runtime environment. When internetware asks the environment for a certain entity, the environment will discover ordered-list of matching entities from registry firstly. Then the environment will use test cases to test and reorder the entities by the newly-tested success-rate of some entities in the ordered-list, e.g., the top 5 entities. The overall success rate of entity e may be calculated as

$$SR_i(e)=\alpha\times tested\text{-}success\text{-}rate(e)+(1-\alpha)\times SR_r(e)$$

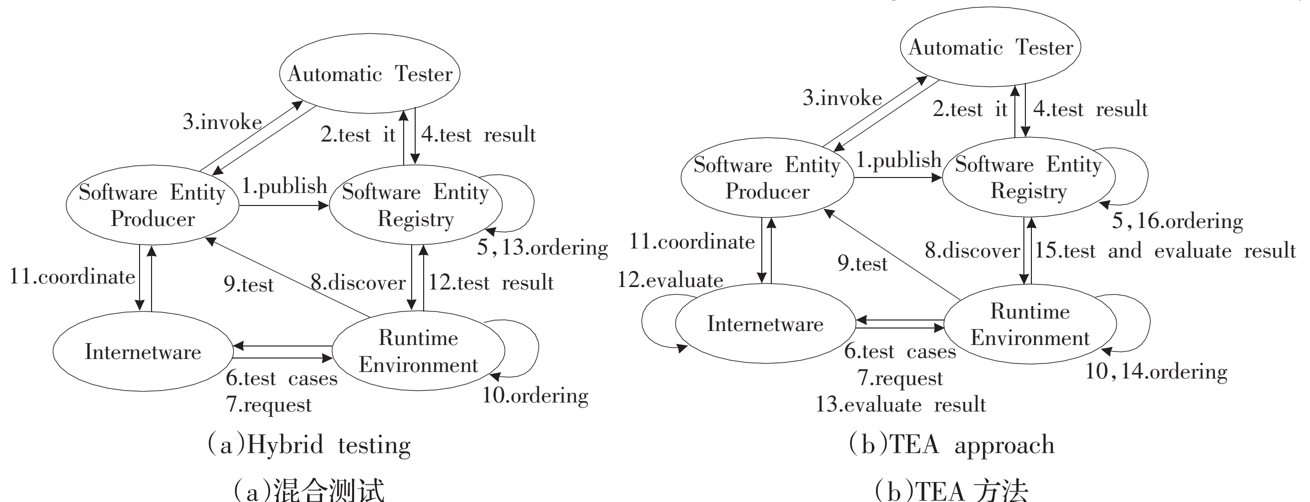
where $SR_r(e)$ is success-rate from registry, $\alpha\in[0,1]$ and usually $\alpha\geqslant 0.5$. Thirdly, entity with needed quality will be forwarded to internetware. Sometimes the top 1 entity won't be forwarded when other factors, such as cost are considered in the entity selection policy. Finally, the environment will report test results to registry to update the ordered-list. As for registry, the overall success rate may be calculated as

$$newSR_r(e)=\beta\times tested\text{-}success\text{-}rate(e)+(1-\beta)\times oldSR_r(e)$$

where $\beta=1/N$, N is the number of distinct internetware feeding back test result. Different weight of *tested-success-rate* is given by runtime environment and entity registry. After gathering enough feedback result, the ordered-list of registry becomes stable and looks more like the "real one", which can only be seen in simulation experiment but not in practice. Usually, the ordered-list in registry is unstable at first. But when more feedback information is gathered from distinct internetware, N grows bigger and the ordered-list becomes stable.

4.2 Testing and Evaluation Approach

Keeping in mind that composing entities are autonomous, they can also change their own structure at runtime as well as internetware. Though the entities have successfully passed the testing before, there is no guarantee that they will not change and will always run as expected. Performing an automated runtime testing from time to time seems a way



(a)Hybrid testing

(a)混合测试

(b)TEA approach

(b)TEA 方法

Fig.2 Hybrid testing and TEA approaches

图 2 混合测试和 TEA 方法

out. However, the choice of time interval is a dilemma. If time interval is set too long, changes of entity may cause a lot of problems before next testing. Otherwise, resource consumed in testing is insufferable. Of course there may be a trade-off, but a better way out is proposed by TEA: a normal invocation can be treated as a test case, i.e., invocation result is evaluated with validation assertions.

Though assertions in contracts can serve as validation assertions, we don't recommend it. Validation assertion concerns more from specific requirements and users' point of view than the contracts, e.g. an entity sums integers in an input Excel file, internetware invokes this entity to sum Asian population amount from population amount of each country in Asian stored in an Excel file, which may assert that global population amount is greater than a billion. If the entity implements the integers as java.lang.Integer, though the contract assertion "result==int_1+int_2+…" still holds, the validation assertion is violated.

The design of validation assertions has a "simple and useful" principle. Simple means the assertion should be easily calculated, i.e., the assertion is not resource and time costly. Useful means the assertion should be sensitive to perceive an error. For example, an execution time assertion can be designed for many entities. Though a room reservation entity keeps on sending "keepALive" message, to indicate that it is reserving a hotel room in the past 5 minutes, it is probably that the entity has failed. If an entity is invoked to solve an equation, the result is easily evaluated by calculating the equation substituted with the result. Business validating rules are other good sources to produce validation assertions. However, designing effective evaluation assertions is not an easy job in many cases.

As shown in Fig.2(b), evaluation result will finally pass to registry. Registry can generate better ordered-list with feedback testing and evaluation results. Steps needed in TEA are sketched as below:

Step 1-5 is performed when new software entity is published.

Step 1 Software Entity Producer publishes entities to Software Entity Registry.

Step 2 Software Entity Registry asks the Automatic Tester to test the newly-registered entity.

Step 3 Automatic Tester invokes the entity to perform a contract-based assertion testing.

Step 4 Automatic Tester returns test result to Software Entity Registry.

Step 5 Software Entity uses test result to order the entity by estimated quality.

Step 6~16 is performed when internetware begins to run.

Step 6 Internetware gives test cases and oracles to its Runtime Environment.

Step 7 Internetware requests its Runtime Environment for a certain entity described in EDL.

Step 8 Runtime Environment discovers ordered-list of matching entities in Software Entity Registry.

Step 9 Runtime Environment uses test cases given in step 6 to test candidate entities in the ordered-list.

Step 10 Runtime Environment calculates and orders overall testing success rate of candidate entities.

Step 11 Internetware selects needed entity according to the overall success rates, and then uses it.

Step 12 Internetware evaluates each normal invocation result.

Step 13 Internetware sends evaluation results to Runtime Environment, when a validation assertion is

violated or all invocations to the entity are finished. In the former case, jump back to step 9.

Step 14 Runtime Environment reorders the ordered-list of matching entities.

Step 15 Runtime Environment passes testing and evaluation results to Software Entity Registry.

Step 16 Software Entity Registry reorders the ordered-list of software entities.

5 Simulation Experiment

5.1 Design

For all the approaches proposed above, it is the different strategies that registry uses have significant effect on the generation of the ordered-list of software entities. Simulation experiment to evaluate these strategies is designed and described below. The results show that TEA is the best among the strategies to order the entities.

5.1.1 Simulation of software entity, tester and invoker

The correctness of software entity is the essential part of the problem. But unfortunately, real correctness can't be measured directly. We simply assume that the distribution of the correctness of software entities is a uniform distribution. This means a random $c_e \in [0,1]$ is adopted as the real correctness for a software entity e. If an input $in \in [0,1]$ for entity e satisfying $in \leqslant c_e$, it is assumed that e will execute correctly as expected. But since c_e is unknown in practice, this assumption is only used directly in the testing where test cases and oracles are provided, i.e., the testing is performed by runtime environment.

As has been argued, contract-based assertions of a software entity are partial. Thus for a software entity e, the probability to satisfy its partial assertions, denoted as a_e, is assumed to uniformly distribute in $[c_e,1]$. Similarly, if an input $in \in [0,1]$ for entity e satisfying $in \leqslant a_e$, it is assumed that output for in will satisfy the assertions. Since whether output data satisfies the assertions or not can be calculated directly, this assumption is used by automated assertion tester. Assertion tester randomly generates input in uniformly distributed in $[0,1]$. If $in \leqslant a_e$, the output for in is assumed to satisfy the post-assertions. Obviously, more inputs will result in a closer assertion success rate to a_e.

Several parameters are simulated for an invoker. The number of test cases is randomly generated $in \in [0, MaxTestNum]$. For each test case, an input $in \in [0,1]$ is randomly generated. Since different entities have different correctness, a test case which fails in one does not necessarily to fail in another. The number of invocations is randomly generated in $[0, MaxInvokeNum]$. For each invocation, a random input $in \in [0,1]$ is also generated. But whether the invocation is successful or not remains unknown. The evaluation assertions are adopted to tell the success of the invocation. The number of evaluation assertions is randomly generated in $[0, MaxEvaNum]$. For each evaluation assertion, an estimation magnified factor $1/s$ is randomly generated, where $s \in (MinEvaFactor, 1]$. The correctness of an entity c_e is magnified to c_e/s, which means when input in satisfies $in \leqslant c_e/s$, the output will be evaluated as correct. If all the evaluation assertions evaluate input in as correct, in is called evaluated success. It can be seen that if s is closer to 0, c_e/s becomes bigger enough to evaluate most input as correct. In the simulation experiment, a $MinEvaFactor$ of 0 and 0.5 is compared.

Entity with higher estimated quality is easier to

be invoked in practice. This is simulated by calculating $Proportion(e)$ for entity e, and multiply $Proportion(e)$ with the total number of invokers, which is set to 1 000 in the simulation experiments, as the number of invokers for e.

$$Proportion(e)=\frac{estimatedQuality(e)}{\sum_{e_i\in E} estimatedQuality(e_i)}$$

5.1.2 Ordering strategies of registry

As stated above, registry can adopt different ordering strategies to generate ordered-list of matching entities. The strategies are summarized in Table 1.

Equally steady means after the execution of a number of invoker, though the success rate may change, the ordering position remains unchanged.

5.1.3 Evaluating the strategies

Actually, it is the ordering position, but not the success rate, that has more importance. Since front positions in the matching list are more important than back positions, a simple position calculation function, named $PosScore(s)$, is proposed to evaluate the performance of the strategies. Strategy has smaller $PosScore(s)$ is better than those have bigger ones.

$$PosScore(s)=\sum_{e_i\in E}\left|Pos(e_i)-calcPos_s(e_i)\right|\times(cardinality(E)-Pos(e_i))$$

where $Pos(e)$ is the ordering position of e by "real" quality, and $calcPos(e)$ is the position of e by estimated quality. The reverse ordering of "real" quality has the largest $PosScore$, and is named as worst score. Standardized ordering score is given as follow:

$$StdScore(s)=(1-\frac{score(s)}{worstScore(cardinality(E))})\times 100\%$$

5.2 Experiment Results

Table 2 is the average score and average deviation of score of 100 runs of $MaxTestNum=10$, $MaxEvaNum=10$, $MaxInvokeNum=1000$, $MinEvaFactor=0.5$ for 10 software entities. Table 3 increases the number of entities to 20.

R Registry coordinated each entity in the first 0 position, thus $score=165$ for 10 entities and $score=1\,330$ for 20 entities at each simulation. Average score of P Registry should be similar with R Registry as can be proved in probability theory, and comfirmed by the simulation results. Partial assertion tests improve the ordering result, which has about 1.6 times $stdScore$ than the random strategy.

Table 1 Ordering strategies

表 1 排序策略

Name of Registry	Ordering Strategy
R Registry	Randomly ordering, which means all entities are coordinated at the first position
P Registry	By Publish time of entities
A Registry	By Assertion success Rate (AR) of Automatic Tester
AT Registry	By AR firstly, then by test case success rate
AE Registry	By AR firstly, then by evaluation success rate
ATE Registry	By AR firstly, then combining TR and ER to an overall success rate
AU(T) Registry	As AT Registry firstly, keep on updating TR with more test cases until become equally steady
AU(E) Registry	As AE Registry firstly, keep on updating ER with more invocations until become equally steady
AU(TE) Registry	As ATE Registry firstly, keep on updating TR and ER with more test cases and invocations until become equally steady

Table 2 Average results of 100 simulations for 10 candidate software entities

表 2 对 10 个候选软件实体进行 100 次模拟的平均结果

	R Reg	P Reg	A Reg	AT Reg	AE Reg	ATE	AU(T)	AU(E)	AU(TE)
Avg	165	164.40	97.96	67.28	51.06	55.01	36.78	32.80	25.32
AvgDev	0	32.92	28.76	27.98	33.20	23.83	20.80	17.56	15.06
StdScore/%	40	40	64	76	81	80	87	88	91

(the worst score is 275)

Table 3 Average results of 100 simulations for 20 candidate software entities

表 3 对 20 个候选软件实体 100 次模拟的平均结果

	R Reg	P Reg	A Reg	AT Reg	AE Reg	ATE	AU(T)	AU(E)	AU(TE)
Avg	1 330	1 354.10	699	536.11	445.47	397.36	181.75	225.24	137.27
AvgDev	0	195.70	161.14	176.68	271.17	148.80	79.11	104.84	54.01
StdScore/%	37	36	67	74	79	81	91	89	93

(the worst score is 2 100)

AT, AE and ATE keep on establishing better ordering. AU(T), AU(E) and AU(TE) refine the ordering when invokers keep on coming. Providing average 5 test cases for an invocation is an easy job, while the outcome is notable when there are a number of invokers. AU(T), though a litter worse than AU(TE), has very effective ordering with *stdScore* around 80 out of 100. Providing average 5 useful evaluation factor, where *MinEvaFactor* =0.5, may not as easily as providing 5 test cases, but as can be seen, the outcome of AU(TE) is remarkable. AU(TE) registry has *stdScore* more than 90, which means it has established the ordered-list very similar with the "correct" one.

Fig.3 is 10 runs of the simulation experiment of Table 2. A, AT and AE are in the upper class, while AU(T), AU(E) and AU(TE) compose the lower class and are separated by ATE. Fig.4 increases *MaxTestNum* to 20. It shows that AU(T) becomes much better. On the other hand, loosen the *MinEvaFactor* to 0 is a disaster for AE and AU(E), in the *MaxEvaNum* =10 experiment, almost all evaluations are successful and AE, AU(E) registry becomes very similar to R registry.

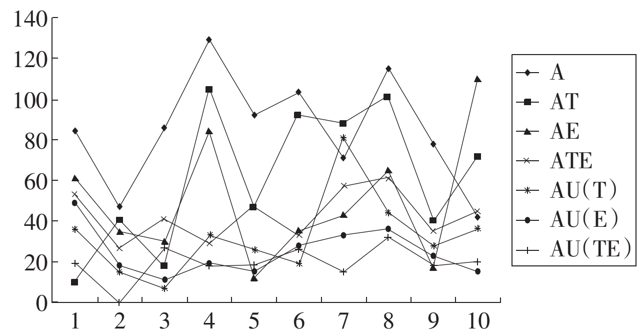


Fig.3 10 runs of *MaxTestNum*=10

图 3 最大测试数为 10 的 10 次运行结果

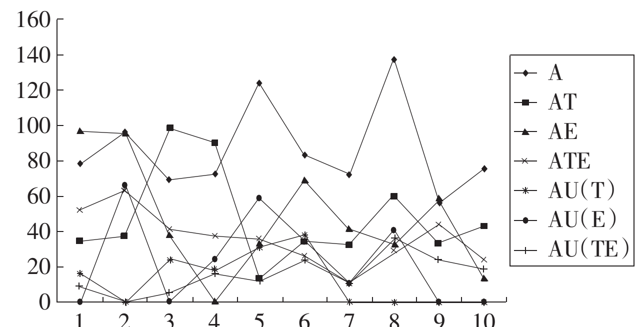


Fig.4 10 runs of *MaxTestNum*=20

图 4 最大测试数为 20 的 10 次运行结果

6 Conclusion

With the progress of internetware[1-3], the correctness and reliability problem of internetware are drawing more and more attentions among researchers. Several pioneer researches[4-6] have been carried out to improve the correctness and reliability of internetware.

In this paper, we propose a Testing and Evaluation Approach (TEA) to further improve the correctness and reliability of internetware. Testing is the most important activity in software quality assurance. Since the qualities of candidate software entities are unknown at development time, it is natural for us to propose an automated runtime testing based on Built-In Testing and contract[12-14] for internetware to select entity with higher quality. Furthermore, composing entities are autonomous and can change their structure at runtime as well as internetware, normal invocation result should be evaluated by validation assertions to find out quality change of the invoked entity as soon as possible with acceptable cost. As in product-purchasing market, feedback information of every single consumer is valuable for the market to form an ordered-list of product of the same kind by estimated quality, feedback information of testing and evaluation results are gathered by software entity registry to generate an ordered-list of entities. The ordered-list is beneficial to the selection of candidate entities for internetware. When the number of internetware grows bigger, the generated ordered-list by estimated quality will become stable and very similar with the ordered-list by "real" quality. Being the most fundamental approach to the correctness and reliability problem of internetware, TEA can be integrated with ABC[3], transaction model[4], Trust Model[5] and fault-tolerate technologies[6] to form a solution to the correctness and reliability problem of internetware.

References:

[1] Yang Fuqing, Mei Hong, Lv Jian, et al. Some thoughts on the development of software technologies[J]. Acta Electronica, 2002,30(12A):1901-1906.

[2] Yang Fuqing. Thinking on the development of software engineering technology[J]. Journal of Software, 2005,15(1):1-7.

[3] Mei Hong, Huang Gang, Zhao Haiyan, et al. A software architecture centric engineering approach for internetware[J]. Science in China Series F: Information Science, 2006,49(6):702-730.

[4] Huang Tao, Ding Xiaoning, Wei Jun. An application-semantics-based relaxed transaction model for internetware[J]. Science in China Series F: Information Science, 2006,49(6):774-791.

[5] Wang Yuan, Lv Jian, Xu Feng, et al. A trust measurement and evolution model for internetware[J]. Journal of Software, 2006,17(4):682-690.

[6] Wang Ping, Sun Changsong, Li Lijie. Primary research on internetware reliability technology[C]//Proceedings of the 1st International Multi-Symposiums on Computer and Computational Sciences (IMSCCS'06), 2006.

[7] Mao Chengying, Lu Yansheng. Research progress in testing techniques of component-based software[J]. Journal of Computer Research and Development, 2006,43(8):1375-1382.

[8] Wang Yingxu, Graham K, Hakan W. A method for built-in tests in component-based software maintenance[C]//Proceedings of the 3rd European Conference on Software Maintenance and Reengineering (CSMR'99). [S.l.]: IEEE Computer Society Press, 1999:186-189.

[9] Hörnstein J, Edler H. Test reuse in CBSE using built-in tests[C]//Proceedings of the Workshop on Component-Based Software Engineering, Composing System from Components. [S.l.]: IEEE Computer Society Press, 2002:11-14.

[10] Vincent J, King G, Lay P, et al. Principles of built-in-test for run-time-testability in component-based software systems[J]. Software Quality Journal, 2002,10:115-133.

[11] Brenner D, Atkinson C, Malaka R, et al. Reducing verification effort in component-based software engineering through built-in testing[J]. Information System Front, 2007, 9:151-162.

[12] Jiang Ying, Xin Guomao, Shan Jinhui, et al. A method of automated test data generation for web service[J]. Chinese Journal of Computers, 2005,28(4):568-577.

[13] Meyer B, Ciupa I, Leitner A, et al. Automatic testing of object-oriented software[C]//van Leeuwen J. LNCS 4362: SOFSEM 2007, 2007:114-129.

[14] Haddox J M, Kapfhammer G M, Michael C C. An approach for understanding and testing third party software components[C]//Proceedings of Annal Reliability and Maintainability Symposium, 2002.

[15] Zhang Jia. An approach to facilitate reliability testing of Web services components[C]//Proceeding of the 15th Int'l Symposium on Software Reliability Engineering (ISSRE'04). [S.l.]: IEEE Computer Society Press, 2004:210-218.

[16] Jiang Ying, Xin Guomao, Shan Jinhui, et al. A method of automated test data generation for web service[J]. Chinese Journal of Computers, 2005,28(4):568-577.

附中文参考文献:

[1] 杨芙清,梅宏,吕建,等.浅论软件技术发展[J].电子学报,2002,30(12A):1901-1906.

[2] 杨芙清.软件工程技术发展思索[J].软件学报,2005,15(1):1-7.

[5] 王远,吕建,徐锋,等.一个适用于网构软件的信任度量及演化模型[J].软件学报,2006,17(4):682-690.

[7] 毛澄映,卢炎生.构件软件测试技术研究进展[J].计算机研究与发展,2006,43(8):1375-1382.

[16] 姜瑛,辛国茂,单锦辉,等.一种Web服务的测试数据自动生成方法[J].计算机学报,2005,28(4):568-577.

CAI Shubin was born in 1979. He is a Ph.D. candidate at SUN Yat-Sen University. His research interests include software engineering and ontology engineering, etc.

蔡树彬(1979-),男,广东汕头人,中山大学博士研究生,主要研究领域为软件工程和本体工程等,发表论文被SCI、EI检索4篇。

MING Zhong was born in 1967. He received the Ph.D. degree in Computer Software and Theory from SUN Yet-Sen University. He is a professor at Shenzhen University. His research interests include software engineering, ontology engineering, semantic web and information retrieval, etc.

明仲(1967-),男,江西宁都人,中山大学博士,深圳大学教授,CCF理事,主要研究方向为本体在软件工程、语义Web等方面的应用,主持国家自然科学基金2项、广东省自然科学基金1项。

LI Shixian was born in 1944. He is a professor and doctoral supervisor at SUN Yet-Sen University. His research interests include software engineering and formal semantics, etc.

李师贤(1944-),男,江西于都人,中山大学教授,博士生导师,CCF专委,主要研究方向为软件工程及形式语义学,主持国家、省部委科研项目多项,发表论文近百篇,译著近20部。