ソフトウェアテストシンポジウム 2005
JaSST'05: Japan Symposium on Software Testing 2005

永続的な反応システムのテスティングとデバッギング －ソフトウェア工学における新しい挑戦－

程 京徳†

†埼玉大学大学院理工学研究科　〒338-8570 さいたま市桜区下大久保 255

E-mail:　†cheng@aise.ics.saitama-u.ac.jp

あらまし　ユビキタスコンピューティングの究極の目的は、いつでもどこでも利用可能な計算環境と情報サービスを提供することである。ユビキタスコンピューティングを実現するためには、その必要条件として、まず、いつでも止まらずに動作し、なおかつ、どこでも利用可能なサービスを提供し続ける永続的なシステムを実現しなければならない。本論文は、永続的な反応システムのテスティングとデバッギングという全く新しい技術課題を提示する。永続的な反応システムとは、一旦起動されると、廃棄されるまでに、故障が生じたとき、攻撃を受けたとき、あるいは保守や更新などが行われるときでさえシステム全体の稼働を止めずに動作しサービスを提供し続ける反応システムである。永続的な反応システムの保守、更新、再構成は、システムが稼働し運用されている状態で行われなければならない。しかし、従来のテスティングとデバッギング技法のほとんどは、プログラムのソースコードを対象としており、稼働しているシステムを対象としていない。また、テストとデバッグのために、プログラムやシステムを実行したり停止したりすることが自由に許されることを前提としている。永続的な反応システムを実現し、運用し、保守するためには、稼働し運用されている状態のシステムに対するテスティングとデバッギングを行う技法を新たに開発しなければならない。これはソフトウェア工学における全く新しい挑戦である。

キーワード　Ubiquitous Computing, Persistently Reactive Systems, Soft System Buses, Highly Reliability, Highly Security

Testing and Debugging Persistently Reactive Systems – A New Challenge in Software Engineering –

Jingde CHENG†

† Department of Information and Computer Sciences, Saitama University, Saitama, 338-8570 Japan

E-mail:　† cheng@aise.ics.saitama-u.ac.jp

Abstract　The ultimate goal of ubiquitous computing is to provide users with the way of computing anytime and anywhere A necessary condition and/or fundamental assumption to underlie ubiquitous computing is that there certainly are computing (information) systems working anytime available anywhere throughout the physical world.　This paper raises a completely new technical issue in the age of ubiquitous computing: how to test and debug a persistently reactive system running continuously?　A persistently reactive system is a reactive system that can run continuously anytime without stopping its service even then it had some trouble, it is being attacked, or it is being maintained or reconfigured.　The maintenance, upgrade, and reconfiguration of a persistently reactive system have to be performed during its continuous running.　However, almost all the existing testing and debugging technologies take programs of a system rather than the running system itself as the objects and/or targets, and assume that any program can be executed repeatedly with various input data only for testing and debugging without regard to stopping the task that program has to perform.　In order to develop, use, and maintain persistently reactive systems, we have to find a new way to test and debug a system running continuously and persistently. This is a completely new challenge in software engineering.

Keyword　Ubiquitous Computing, Persistently Reactive Systems, Soft System Buses, Highly Reliability, Highly Security

1. Introduction

A reactive system is a computing system that maintains an ongoing interaction with its environment, as opposed to computing some final value on termination [12, 17]. Modern society is more and more dependent on various reactive systems such as computer operating systems, computer networks, air traffic control systems, train traffic control systems, nuclear reactor control systems, various embedded and process control systems, digital libraries, and web service systems, and, of course, dependent on the continuous, reliable, and secure functioning of the systems. Since a breakdown of a large-scale reactive system running world-wide may be disastrous to our lives, how to design, develop, and maintain highly reliable, highly secure, and large-scale reactive systems has become a very important issue in modern software engineering.

Obviously, it is very desirable, if possible, that a reactive system with requirements of high reliability and high security can run continuously anytime without stopping its service even then it had some trouble, it is being attacked, or it is being maintained or reconfigured. In particular, since defining requirements of a large-scale reactive system formally and then specifying its behavior formally are very difficult, a challenging engineering problem is how to design, develop, and maintain a large-scale reactive system such that it not only satisfies initial requirements but also adapts to new changes while running continuously and persistently.

On the other hand, the Internet, which itself can be regarded as a huge reactive system, and its associated technologies have changed the way reactive systems serve for users. Today, many applications require a reactive system to respond to its users ubiquitously. The ultimate goal of ubiquitous computing is to provide users with the way of computing anytime and anywhere [20]. Obviously, a necessary condition and/or fundamental assumption to underlie ubiquitous computing is that there certainly are computing (information) systems working anytime available anywhere throughout the physical world. Therefore, ubiquitous computing must lead to requiring reactive systems running continuously and persistently.

However, the requirement that a reactive system should run continuously and persistently is not taken into account as an essential and/or general requirement by traditional reactive systems. From the viewpoints of ubiquitous computing, the present author has proposed a new type of reactive systems, named "***persistently reactive systems***." A persistently reactive system is a reactive system that can run continuously anytime without stopping its service even then it had some trouble, it is being attacked, or it is being maintained or reconfigured. The maintenance, upgrade, and reconfiguration of a persistently reactive system have to be performed during its continuous running. However, almost all the existing testing and debugging technologies take programs of a system rather than the running system itself as the objects and/or targets, and assume that any program can be executed repeatedly with various input data only for testing and debugging without regard to stopping the task that program has to perform. In order to develop, use, and maintain persistently reactive systems, we have to find a new way to test and debug a system without stopping its running. This is a completely new challenge in software engineering.

2. Designing, Developing and Maintaining Persistently Reactive Systems

A good design, development, and maintenance methodology for reactive systems must be based on a deep recognition and understanding of the intrinsic characteristics of the systems. The present author has proposed the following general principles in concurrent systems engineering [6, 7]:

The ***dependence principle in measuring, monitoring, and controlling***: "Any system cannot control what it cannot measure and monitor."

The ***wholeness principle of concurrent systems***: "The behavior of a concurrent system is not simply the mechanical putting together of its parts that act concurrently but a whole such that one cannot find some way to resolve it into parts mechanically and then simply compose the sum of

its parts as the same as its original behavior."

The ***uncertainty principle in measuring and monitoring concurrent systems***: "The behavior of an observer such as a run-time measurer or monitor cannot be separated from what is being observed."

The ***self-measurement principle in designing, developing, and maintaining concurrent systems***: "A large-scale, long-lived, and highly reliable concurrent system should be constructed by some function components and some (maybe only one) permanent self-measuring components that act concurrently with the function components, measure and monitor the system itself according to some requirements, and pass run-time information about the system's behavior to the outside world of the system."

Based on the above principles, the present author considered that a persistently reactive system can be constructed by a group of control components including self-measuring, self-monitoring, and self-controlling components with general-purpose which are independent of systems, a group of functional components to carry out special takes of the system, some data/instruction buffers, and some data/instruction buses. The buses are used for connecting all components and buffers such that all data/instructions are sent to target components or buffers only through the buses and there is no direct interaction which does not invoke the buses between any two components and buffers.

Conceptually, a ***soft system bus***, SSB for short, is simply a communication channel with the facilities of data/instruction transmission and preservation to connect components in a component-based system. It may consist of some ***data-instruction stations***, which have the facility of data/instruction preservation, connected sequentially by ***transmission channels***, both of which are implemented in software techniques, such that over the channels data/instructions can flow among data-instruction stations, and a component tapping to a data-instruction station can send data/instructions to and receive data/instructions from the data-instruction station [9, 10].

An ***SSB-based system*** is a component-based system consisting a group of control components including self-measuring, self-monitoring, and self-controlling components with general-purpose which are independent of systems, and a group of functional components to carry out special takes of the system such that all components are connected by one or more SSBs and there is no direct interaction which does not invoke the SSBs between any two components [9, 10].

The most intrinsic characteristic or most important requirement of SSBs is that an SSB must provide the facility of data/instruction preservation such that when a component in a system cannot work well temporarily all data/instructions sent to the component should be preserved in some data-instruction station(s) until the component works well to get these data/instructions. Therefore, other components in the system should work continuously without interruption, except those components that waiting for receiving new data/instructions sent from the component in question.

From the viewpoint of structure, an SSB may be either linear or circular. On the other hand, from the viewpoint of information flow direction, data/instruction flows along an SSB may be either one-way or bidirectional. Therefore, there may be four types of SSBs: ***linear one-way***, ***linear bidirectional***, ***circular one-way***, and ***circular bidirectional*** SSBs. It is obvious that different types of SSBs will provide system designers and developers a variety of technical benefits and functional advantages to make target systems more flexible and powerful. On the other hand, different types of SSBs will have different difficult to implement.

As an example, Fig. 1 shows a circular SSB architecture of SSB-based system. The group of central control components includes a central measurer (Me), a central recorder (R), a central monitor (Mo), and a central controller/scheduler (C/S), all of which are permanent components of the system, and are independent of any application. These central control components are connected by a circular SSB such that all data and instructions are sent to or received by components only through the SSB and there is no direct interaction which does not invoke the buses between any two components. The functional

components are measured, recorded, monitored, and controlled by the central control components. All measurement data, instructions issued by the central control components, and communicating data between components flow along the SSB.

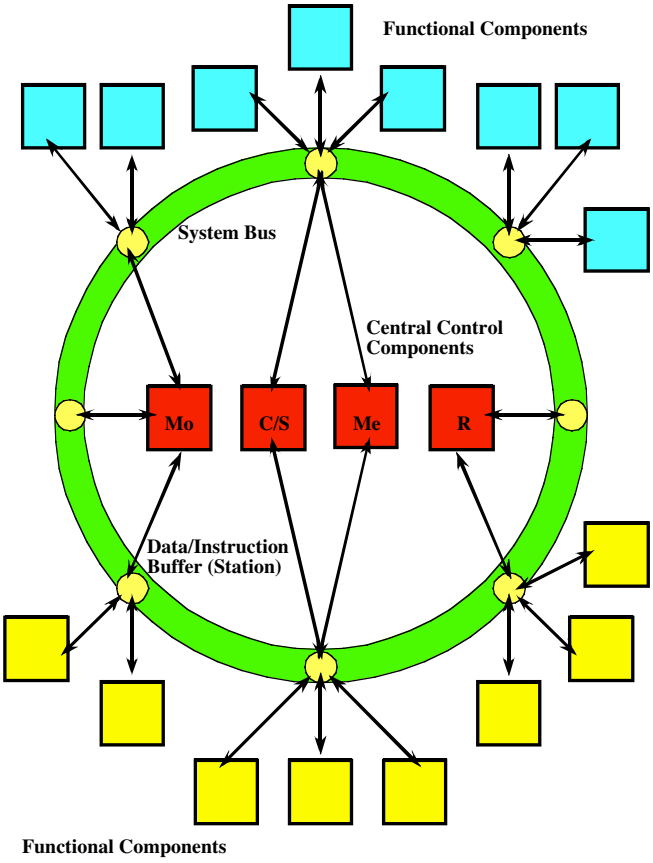

Fig. 1 A circular SSB architecture

As this example shows, in an SSB-based system, the group of central control components can be regarded as the 'heart' and/or 'brain' of the system, while the SSBs can be regarded as 'nerves' and/or 'blood vessels' of the system. This is a completely new, control-oriented design and development methodology quite different from the traditional design and development methodology that is function-oriented [2, 3].

Consequently, SSBs can provide system designers and developers a variety of technical benefits and functional advantages to make target systems more reliable, secure, adaptive, and flexible. Using SSBs, system designers and developers can build persistently reactive systems such that they can be easily maintained and upgraded, without changing the basic system architecture, by adding some new components for satisfying new requirements, replacing an old or problematic component with a newer or sounder one, and removing some useless components. The maintenance and reconfiguration of a persistently reactive system built by SSBs even can be done without stopping the running of the whole system, if those components added, replaced, or removed are functional components but not permanent control components.

3. Testing and Debugging Persistently Reactive Systems

A persistently reactive system has to be maintained, upgraded, and reconfigured during its continuous and persistent running. This raises a new challenging issue: how to test and debug a persistently reactive system running continuously?

Because any testing and debugging of a program and/or system must based on its basic requirements, first of all we enumerate some essential requirements for persistently reactive systems (SSB-based systems) as follows:

R1: The running of any control component in an SSB-based system must not be stopped.

R2: Any control component in an SSB-based system must not be dependent on any special functional component in the system.

R3: Any functional component in an SSB-based system must be able to be added to, upgraded, replaced, or moved from the system without stopping the running of the whole system.

R4: The stop of running of any functional component in an SSB-based system must not lead to the stop of running of the whole system.

R5: All data/instructions sent to a functional component in an SSB-based system must be preserved, if the functional component does not work, until the functional component works well and must be resented to it.

R6: The SSBs must not be dependent on any special computing environment including computers, operating systems, and programming

languages.

R7: All data/instructions flowing over SSBs must have the unified form.

R8: The SSBs must be able to be implemented in distributed way as well as centralized way.

R9: Any control component in an SSB-based system must be invisible and inaccessible to the outside world of the system.

R10: Any instruction to and any operation on any functional component in an SSB-based system must be authenticated.

R11: All data/instructions flowing over SSBs must be able to be enciphered in different degree of security according to different security policies of application systems.

R12: The running of the whole system must be stopped when its continuous running will lead to a disaster.

Testing is an indispensable step in software development and maintenance. A program error is a difference between the actual behavior of a program and the behavior required by the specification of the program. The purpose and/or goal of testing is to find errors in a target program/system. Traditionally, testing is defined as the process of executing the target program/system to determine whether it matches its specification and executes in its intended environment [13, 18, 21].

Debugging is another indispensable step in software development and maintenance. A program "bug" relative to a program error is a cause of the error. A bug may cause more than one error, and also, an error may be caused by more than one bug. Traditionally, debugging is defined as the process of locating, analyzing, and ultimately correcting bugs in the target program/system [1, 4, 13]. In general, debugging is performed by reasoning about causal relationships between bugs and the errors which have been detected in program/system by testing. It begins with some indication of the existence of an error, repeats the process of developing, verifying, and modifying hypotheses about the bug(s) causing the error until the location of the bug(s) is determined and the nature of the bug(s) is understood, then corrects the bug(s), and ends in a verification of the removal of the error [4].

Testing and debugging a concurrent program/system is more difficult than testing and debugging a sequential program/system because a concurrent program/system has multiple control flows, multiple data flows, and interprocess synchronization, communication, and nondeterministic selection. An intrinsic characteristic of concurrent programs is the so-called "unreproducibility of behavior," i.e., for a concurrent program, two different executions with the same input may produce different behavior and histories because of unpredictable rates of processes and existence of nondeterministic selection statements in the program [4, 15, 19].

Almost all the existing testing and debugging technologies take programs of a system rather than the running system itself as the objects and/or targets [11, 14, 15, 16]. A fundamental assumption underlying the existing testing and debugging technologies is that any program can be executed repeatedly with various input data only for testing and debugging without regard to stopping the task that program has to perform. However, for persistently reactive systems this fundamental assumption does not hold no longer. Therefore, we have to find a new way to test and debug a system running continuously and persistently.

Some major new issues in testing and debugging persistently reactive systems are as follows:

First, since continuous and persistent running without stopping services is the most essential and/or general requirement for persistently reactive systems, it is of course specified in the specification of any persistently reactive system. Therefore, a completely new class of errors in persistently reactive systems should be "running/serving stop" errors, i.e., those system situations stopping the running of the whole system. From the viewpoint that the most intrinsic characteristic or most important requirement of persistently reactive systems is continuous and persistent running without stopping services, this new class of errors should be most serious one to any persistently reactive

system. We have to find some systematic method to test the running/serving stop errors.

In order to test any behavior of a target program/system according to a requirement, the requirement must be testable, i.e., to be precisely and unambiguously defined. Traditionally, a requirement is defined to be testable if it is possible to design a procedure in which the functionality being tested can be executed, the expected output is known, and the output can be programmatically or visually verified [9]. Obviously, this traditional definition for the testability of requirement has to be revised such that the requirement of non-stop running/serving is taken into account. On the other hand, the IEEE Standard 610 only defined the following six different types of requirements: design, functional, implementation, interface, performance, and physical requirements. If we consider the non-stop running/serving is a function that a persistently reactive system must be able to perform, then it can be classified into functional requirements; otherwise a new type of requirement has to be defined. Only after we have an explicit, precise, and unambiguous definition for the testability of requirements on non-stop running/serving, we can start on test planning, test case design, test data generation, and test result evaluation for persistently reactive systems.

Second, as we have mentioned, a fundamental assumption underlying the existing testing technologies is that any program can be executed repeatedly with various input data only for testing without regard to stopping the task that program has to perform. Therefore, almost all traditional and/or usual requirements, i.e., design, functional, implementation, interface, performance, and physical requirements, should be reconsidered. If the testability of a requirement is underlain by the fundamental assumption, then it has to be revised or redefined.

Third, in testing a persistently reactive system, any testing action must not disturb the task of any sound component in order to satisfy the requirement of non-stop running/serving. If the testing concerns not only one component but also other components, how to perform the testing well but do not disturb those sound components may be a difficult issue.

Fourth, debugging a persistently reactive system must be more difficult than testing it because debugging must make some modification to remove bugs from the system and then correct it. Similar to the case of testing, in debugging a persistently reactive system, any debugging action must not disturb the task of any sound component in order to satisfy the requirement of non-stop running/serving. This must be more difficult than testing because debugging has to modify the system.

Fifth, because any persistently reactive system is a concurrent system, in general, its behavior is not reproducible. Moreover, because a persistently reactive system being debugged is running continuously, some interaction with its outside environment may be taken place during debugging. Therefore, it must be quite difficult to establish a mapping from the programs of the system to its actual behavior at various time points. This means that the reasoning about causal relationships between bugs and the errors may be quite difficult. The present author's consideration is that temporal relevant logic is an indispensable tool for this task [5, 8].

Finally, for an SSB-based persistently reactive system, if the control components or data-instruction stations need to be tested and/or debugged, the task is more difficult than testing and debugging its functional components, because some run-time information may be not available in these situations.

4. Concluding Remarks

We have shown a new challenge in software engineering: how to test and debug a persistently reactive system running continuously? We have also shown some completely new technical issues in development and maintenance of persistently reactive systems. There are many interesting and challenging research problems in this direction. To solve these difficult problems is necessary to actualizing the ultimate goal of ubiquitous computing.

References

[1] K. Araki, Z. Furukawa, and J. Cheng, "A General Framework for Debugging," IEEE-CS Software, Vol. 8, No. 3, pp. 14-20, May 1991.

[2] J. Bacon, "Concurrent Systems: An Integrated Approach to Operating Systems, Distributed Systems and Databases (3rd Edition)," Addison-Wesley, 2002.

[3] A. W. Brown, "Large-Scale, Component-Based Development," Prentice Hall, 2000.

[4] J. Cheng, "Slicing Concurrent Programs -- A Graph-Theoretical Approach," in P. A. Fritzson (Ed.), "Automated and Algorithmic Debugging, 1st International Workshop, AADEBUG'93, Linkoping, Sweden, May 1993, Proceedings," Lecture Notes in Computer Science, Vol. 749, pp. 223-240, Springer-Verlag, November 1993.

[5] J. Cheng, "Temporal Relevant Logic as the Logic Basis for Reasoning about Dynamics of Concurrent Systems," Proc. 1998 IEEE-SMC Annual International Conference on Systems, Man, and Cybernetics, Vol. 1, pp. 794-799, San Diego, USA, October 1998.

[6] J. Cheng, "The Self-Measurement Principle: A Design Principle for Large-scale, Long-lived, and Highly Reliable Concurrent Systems," Proc. 1998 IEEE-SMC Annual International Conference on Systems, Man, and Cybernetics, Vol. 4, pp. 4010-4015, 1998.

[7] J. Cheng, "Wholeness, Uncertainty, and Self-Measurement: Three Fundamental Principles in Concurrent Systems Engineering," Proc. 13th International Conference on Systems Engineering, pp. CS7-CS12, 1999.

[8] J. Cheng, "Temporal Relevant Logic as the Logical Basis of Anticipatory Reasoning-Reacting Systems" (Best Paper Award awarded at 6th CASYS), in D. M. Dubois (Ed.), "COMPUTING ANTICIPATORY SYSTEMS: CASYS 2003 - Sixth International Conference, Liege, Belgium, 11-16 August 2003," AIP Conference Proceedings, Vol. 718, pp. 362-375, American Institute of Physics, 2004.

[9] J. Cheng, "Soft System Bus as a Future Software Technology," Proc. 8th International Symposium on Future Software Technology, 2004.

[10] J. Cheng, "Connecting Components with Soft System Buses: A New Methodology for Design, Development, and Maintenance of Reconfigurable, Ubiquitous, and Persistent Reactive Systems," Proc. 19th IEEE-CS International Conference on Advanced Information Networking and Applications, March 2005.

[11] E. Dustin, "Effective Software Testing: 50 specific ways to improve your testing," Addison-Wesley, 2003.

[12] D. Harel and A. Pnueli, "On the Development of Reactive Systems," in K. R. Apt (Ed.), "Logics and Models of Concurrent Systems," pp. 477-498, Springer-Verlag, 1985.

[13] IEEE Standard 610, "Standard Computer Dictionary – A Compilation of IEEE Standard Computer Glossaries," 1990.

[14] R. Lecevicius, "Advanced Debugging Methods," Kluwer Academic, 2000.

[15] E. McDowell and D. P. Helmbold, "Debugging Concurrent Programs," ACM Computing Surveys, Vol. 21, No. 4, pp. 593-622, 1989.

[16] G. J. Myers, "The Art of Software Testing," John Wiley & Sons, 1979.

[17] A. Pnueli, "Specification and Development of Reactive Systems," in H.-J. Kugler (Ed.), "Information Processing 86," pp. 845-858, IFIP, North-Holland, 1986.

[18] S. R. Schach, "Testing: Principles and Practice," ACM Computing Surveys, Vol. 28, No. 1, pp. 277-279, 1996.

[19] K. C. Tai and R. H. Carver, "Testing of Distributed Programs," in A. Y. Zomaya (Ed.), "Parallel and Distributed Computing Handbook," pp. 955-978, McGraw-Hill, 1996.

[20] M. Weiser, "Some Computer Science Problems in Ubiquitous Computing," Communications of the ACM, Vol. 36, No. 7, 1993.

[21] J. A. Whittaker, "What Is Software Testing? And Why Is It So Hard?," IEEE-CS Software, Vol. 17, No. 1, pp. 70-79, 2000.