Persistent Computing Systems as an Infrastructure of Computing Anticipatory Systems

Jingde Cheng and Feng Shang
Department of Information and Computer Sciences, Saitama University
Saitama, 338-8570, Japan
{cheng, frank}@aise.ics.saitama-u.ac.jp

Abstract
The notion of anticipatory system, in particular, computing anticipatory system, implies a fundamental assumption or requirement, i.e., to be anticipatory, a computing system must behave continuously and persistently without stopping its running. However, the requirement that a computing system should run continuously and persistently is never taken into account as an essential or general requirement by traditional system design and development methodologies. As a result, a traditional computing system often has to stop its running and service when it needs to be maintained, upgraded, or reconfigured, it has some trouble, or it is attacked. From the viewpoints of anticipatory computing and persistent computing, this paper shows that a new type of computing systems, named "persistent computing systems," is indispensable to design and development of true computing anticipatory systems. The paper also discusses how persistent computing systems can be constructed by soft system bus technology, and present some new scientific and technical challenges on anticipatory computing and persistent computing.
Keywords: Anticipatory computing, Persistent computing, Anticipatory reasoning-reacting systems, Soft system bus, Design and development methodology.

1 Introduction

The concept of an anticipatory system first proposed by Rosen in 1980s [33]. Rosen considered that "an anticipatory system is one in which present change of state depends upon future circumstance, rather than merely on the present or past" and defined an anticipatory system as "a system containing a predictive model of itself and/or its environment, which allows it to change state at an instant in accord with the model's prediction to a latter instant." Rosen considered that the ability to take anticipation is the intrinsic difference between organisms or living systems and non-living systems.

Since the pioneering work of Rosen, many scientists from various disciplines have given various definitions of anticipatory systems in order to understand the characteristics of anticipatory systems.

Dubois defined that "a computing anticipatory system is a system which computes its current states in taking into account its past and present states but also its potential future states." Dubois also distinguished between strong anticipation and weak anticipation such that "strong anticipation refers to an anticipation of events built by or embedded in a system, while weak anticipation refers to an anticipation of events predicted or forecasted from a model of a system." [15-17] More formally, Dubois

defined an incursive discrete weak anticipatory system as "a system which computes its current state at time t, as a function of its states as past times, ..., $t-3$, $t-2$, $t-1$, present time t, and even its predicated states at future times $t+1$, $t+2$, $t+3$, ...

$$\boldsymbol{x}(t+1) = \boldsymbol{A}(\ldots, \boldsymbol{x}(t-2), \boldsymbol{x}(t-1), \boldsymbol{x}(t), \boldsymbol{x}^*(t+1), \boldsymbol{x}^*(t+2), \ldots; \boldsymbol{p})$$

where the variable $\boldsymbol{x}^*$ at future times $t+1$, $t+2$, ... are computed in using a predictive model of the system." (Note: $\boldsymbol{x}(t)$ are the vector states at time t and $\boldsymbol{p}$ a set of parameters to be adjusted.) Dubois also defined an incursive discrete strong anticipatory system as "a system which computes its current state at time t, as a function of its states as past times, ..., $t-3$, $t-2$, $t-1$, present time t, and even its predicated states at future times $t+1$, $t+2$, $t+3$, ...

$$\boldsymbol{x}(t+1) = \boldsymbol{A}(\ldots, \boldsymbol{x}(t-2), \boldsymbol{x}(t-1), \boldsymbol{x}(t), \boldsymbol{x}(t+1), \boldsymbol{x}(t+2), \ldots; \boldsymbol{p})$$

where the variable $\boldsymbol{x}$ at future times $t+1$, $t+2$, ... are computed in using the equation itself." [18]

Nadin mentioned the following operational definitions: "An anticipatory system is a system whose current state is defined by a future state." "An anticipatory system is a system containing a predictive model of itself and/or of its environment, which allows it to change state at an instant in accord with the model's predictions pertaining to a later instant, in faster than real time." [28-30]

Butz, Sigaud, and Gerard defined anticipatory behavior as "a process, or behavior, that does not only depend on past and present but also on predictions, expectations, or beliefs about the future." [2]

From these definitions of anticipatory systems, we can see that the following two characteristics are common in the definitions: (1) for any anticipatory system, concerning its current state, there must be a future state referred by the current state, and (2) for any anticipatory system, its states form an infinite sequence. Therefore, we can say that the notion of anticipatory system, in particular, computing anticipatory system, implies a fundamental assumption or requirement, i.e., to be anticipatory, a commuting system must behave continuously and persistently without stopping its running.

However, the requirement that a computing system should run continuously and persistently is never taken into account as an essential and/or general requirement by traditional system design and development methodologies. A fact to support this proposition is that we cannot find 'persistence' and/or 'persistent' related technical terms defined or listed in various computer dictionaries, computer glossaries, encyclopedia of software engineering, and handbooks of software reliability such as [23-27, 32, 41]. As a result, although there are some individual computing systems that are designed and developed with considerations on fault tolerance [1, 22], in general a traditional computing system often has to stop its running and service when it needs to be maintained, upgraded, or reconfigured, it has some trouble, or it is attacked, because at the first stage of design and development of a system it is impossible to completely specify all possible troubles, attacks, and end user requests that appear in the stage of its

everyday use. Therefore, traditional computing systems cannot provide an infrastructure for computing anticipatory systems.

From the viewpoint of anticipatory computing and persistent computing, this paper shows that a new type of computing systems, named "persistent computing systems," is indispensable to design and development of true computing anticipatory systems. The paper also discusses how persistent computing systems can be constructed by soft system bus techniques, and present some new scientific and technical challenges on anticipatory computing and persistent computing.

2 Persistent Computing and Persistent Computing Systems

Persistent Computing is proposed by Cheng as a new methodology that aims to develop continuously dependable and dynamically adaptive reactive-systems, called "***persistent computing systems***," in order to build more tough, useful, and human-friendly reactive systems [11, 12, 14]. Conceptually, a reactive system is a computing system that maintains an ongoing interaction with its environment, as opposed to computing some final value on termination [21, 31]. A ***persistent computing system*** is a reactive system that functions continuously anytime without stopping its reactions even when it is being maintained, upgraded, or reconfigured, it had some trouble, or it is being attacked [11, 12, 14]. Persistent computing systems have the two key characteristics and/or fundamental features: (1) persistently continuous functioning, i.e., the systems can function continuously and persistently without stopping its reactions, and (2) dynamically adaptive functioning, i.e., the systems can be dynamically maintained, upgraded, or reconfigured during its continuous functioning.

From the viewpoint of function (here we use "function" to mean "provide correct computing service to end users"), all states of a computing system can be divided into three classes: ***functional states***, ***partially functional states***, and ***disfunctional states***. In a functional state, the system can function completely; in a partially functional state, the system can function partially but not completely; in a disfunctional state, the system cannot function at all. While, from the viewpoint of reaction (here we use "reaction" to mean "react to the outside environment"), all states of a computing system can be divided into three classes: ***reactive states***, ***partially reactive states***, and ***dead states***. In a reactive state, the system can react completely; in a partially reactive state, the system can react partially but not completely; in a dead state, the system cannot react at all. Therefore, a system in functional state must be also in reactive state; a system in partially functional may be in either reactive state or partially reactive state; a system in disfunctional state may be in either reactive state, partially reactive state, or dead state.

Based on the above definitions, we can also define a persistent computing system as a reactive system which will never be in a dead state such that it can evolve into a new functional state in some (autonomous or controlled) way, or can be recovered into a functional state from a partially functional or disfunctional state by some (autonomous or controlled) way. While, a traditional computing system is different from the persistent computing system in that it must eventually be in a dead state.

Persistent computing and persistent computing systems are motivated by the following problems: (1) to solve the problem of automated theorem finding [4, 5, 39, 40], (2) to develop autonomous evolutionary information systems [8, 13], (3) to build anticipatory reasoning-reacting systems [9, 10, 20, 34], and (4) to provide users with the way of computing anytime anywhere, i.e., ubiquitous computing [11, 12, 14]. Refer to [14] for some more detailed explanations. Besides the third problem that is directly related to computing anticipatory systems, the other three problems are also some how related to computing anticipatory systems. For example, anticipatory reasoning can be regarded as the problem of automated theorem finding in a formal theory based on temporal relevant logics; Torres-Carbonell, Parets-Llorca, and Dubois have discussed the relationship between software systems evolution and incursive discrete strong anticipatory systems [36]; if we consider computing anticipatory systems from not only the aspect of time but also the aspect of space, then they certainly concern ubiquitous computing.

Persistent computing and persistent computing systems are proposed with the considerations based on the following fundamental principles [6, 7]:

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."

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

Based on the above fundamental principles, we considered that a persistent computing 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.

3 Design and Development of Persistent Computing Systems by Soft System Bus Technology

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 [11].

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. It is this facility of data/instruction preservation that supports persistent computing in the sense that the maintenance, upgrade, and reconfiguration of a persistent computing system can be done without stopping the running of the whole system.

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.

An SSB may be implemented as a distributed one as well as a centralized one. The transmission channels of a distributed SSB may be implemented in a wired network, a wireless network, or a hybrid one by remote message-passing procedure calls, while the transmission channels of a centralized SSB may be implemented in the host computer by usual message-passing procedure calls.

Conceptually, 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 [11, 14].

Similar to SSBs, the connection between components and data-instruction stations in a distributed SSB-based system may be implemented in a wired network, a wireless network, or a hybrid one by remote message-passing procedure calls, while connection between components and data-instruction stations in a centralized SSB-based system may be implemented in the host computer by usual message-passing procedure calls. The most simple, centralized SSB-based system may include only one central control component as the self-measuring, self-monitoring, and self-controlling component,

while a large-scale, distributed SSB-based system may include many self-measuring, self-monitoring, and self-controlling components respectively. All control components of an SSB-based system should be invisible and inaccessible to any end-user. On the other hand, a functional component in an SSB-based system may be a one with complex internal structure to perform some difficult task as well as a very simple one only with the ability of communication with a data-instruction station.

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.

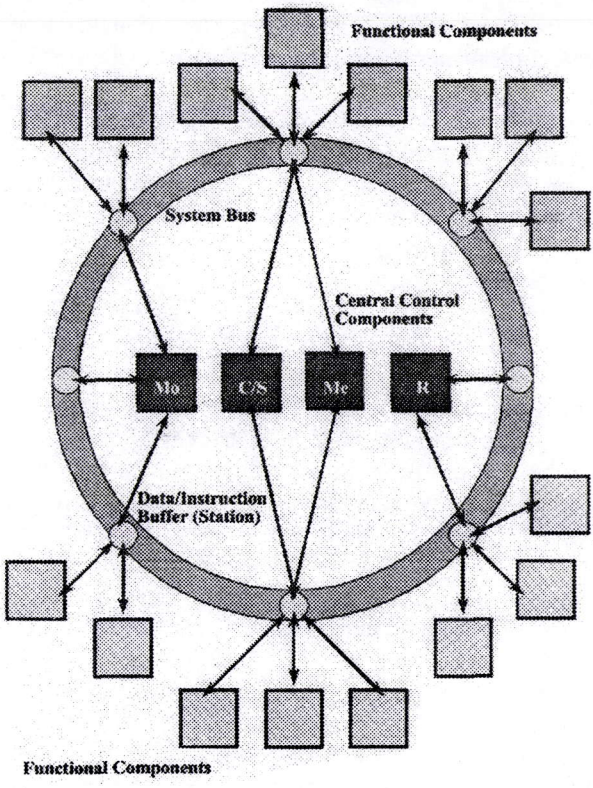

Fig. 1 A circular SSB architecture

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, the SSBs can be regarded as 'nerves' and/or 'blood vessels' of the system, while the functional components can be regarded as the 'mouth', 'eyes', 'nose', 'hands', and 'feet' 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. Refer to [11] for some detailed discussions on SSB and SSB-based systems from the viewpoint of software engineering.

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 persistent computing 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 persistent computing 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.

To achieve the goals, a persistent computing system built as an SSB-based system must at least satisfy the following basic requirements:

R1: 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 maintained, upgraded, replaced, added to, 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 resent 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 interaction between an SSB-based system and its outside world must be able to be stopped when the interaction will lead to a disaster.

In order to satisfy the above basic requirements and to provide system designers, developers, and maintainers with enough technical benefits and functional advantages, the SSBs and its associated technologies must at least provide the following functions and/or facilities:

F1: The way for system designers, developers, and maintainers to identify and specify any component in an SSB-based system.

F2: The way of self-measuring, self-monitoring, and self-recording of system states.

F3: The way for components to send data/instructions to and receive data/instructions from data-instruction stations.

F4: The way for components to specify partners in cooperation and communication.

F5: The way for control components to manage functional components including, adding, upgrading, replacing, and removing the functional components and starting or stopping the running of them.

F6: The facility of data/instruction preservation and retransmission.

F7: The facility of authentication.

F8: The facility of encipherment.

F9: The way to stop the interaction between an SSB-based system and its outside world.

4 Scientific and Technical Challenges

Until now, there is no persistent computing system that has been implemented to satisfy all requirements for persistent computing systems. Probably, even some implementation issues have not been identified. To implement a true persistent computing system useful in practices, we have to solve many scientific and technical challenging problems.

A scientific and theoretical fundamental problem is how to define a persistent computing model formally. Traditionally, the notion of computability in computer science is intrinsically a finite concept such that any computable problem must be able to be computed within finite steps. However, anticipatory computing itself as well as persistent computing intrinsically concerns an infinite sequence of states to be computed. Thus, from the viewpoint of computation, we have some fundamental questions as follows: What is "computable" by a persistent computing system as well as an anticipatory computing system? Is there some intrinsic difference between the notion of "computability" by persistent computing as well as anticipatory computing and the notion of Turing-computability? Is there some intrinsic difference between the notion of "computability" by persistent computing and that by anticipatory computing? Is any Turing-incomputable problem "computable" by a persistent computing system as well as an anticipatory computing system?

Another scientific and theoretical fundamental problem is how to define the notion of function and the notion of reaction of a computing system formally such that their

difference can be used for distinguishing functional states, partially functional states, and disfunctional states from reactive states, partially reactive states, and dead states.

The concept of autonomous and continuous evolution of a persistent computing system also should be clarified philosophically and theoretically.

The biggest technical challenge offered by persistent computing is how to protect, maintain, upgrade, and reconfigure control components of a persistent computing system. Because the control components are the pivot of a persistent computing system, a trouble of any control component may lead to a dead state of the whole system.

Any of reliability and security policies, requirements, functions, and facilities of a persistent computing system must be able to be updated, exchanged, added, or deleted while running of the whole system without stopping service. How to satisfy this requirement is a completely open problem.

In order to implement a persistent computing system, the methodology and technology for self-measuring, self-monitoring, and self-controlling are indispensable.

To implement a true persistent computing system, at first we need to have some technical ways previously to test and debug it. A persistent computing system has to be maintained, upgraded, and reconfigured during its continuous and persistent running. This raises a new technical challenge: how to test and debug a persistent computing system running continuously without stopping?

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 are 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 [38].

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 [3]. 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 [3].

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 [3, 35].

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. 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 persistent computing 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 persistent computing systems are as follows:

First, since continuous and persistent running without stopping services is the most essential and/or general requirement for persistent computing systems, it is of course specified in the specification of any persistent computing system. Therefore, a completely new class of errors in persistent computing 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 persistent computing systems is continuous and persistent running without stopping services, this new class of errors should be most serious one to any persistent computing 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 [19]. 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 [24]. If we consider the non-stop running/serving is a function that a persistent computing 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 persistent computing 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 persistent computing 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 persistent computing 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 persistent computing 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 persistent computing system is a concurrent system, in general, its behavior is not reproducible. Moreover, because a persistent computing 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. Our consideration is that temporal relevant logic is an indispensable tool for this task.

Finally, for an SSB-based persistent computing system, when a functional component is tested or debugged, it may need a substitute to play temporarily the part for the functional component being tested or debugged. Therefore, some switchover technology is necessary. On the other hand, if the control components or data-instruction stations need to be tested and/or debugged, the task is more difficult than testing and debugging functional components, because some run-time information may be not available in these situations. It is obvious that the switchover of control components or data-instruction stations must be difficult by far.

5 Concluding Remarks

We have presented that to be anticipatory, a computing system must behave continuously and persistently without stopping its running and showed that a new type of computing systems, named "persistent computing systems," is indispensable to design and development of true computing anticipatory systems. We also discussed how persistent computing systems can be constructed by soft system bus technology, and presented some new scientific and technical challenges on anticipatory computing and persistent computing.

Having persistent computing systems as an infrastructure, we can design and develop various true anticipatory computing systems. We are working on the scientific and technical challenges presented in this paper in order to implement persistent computing systems. We are also working on design and development of a general-purpose package of soft system bus with control components and data/instruction stations such that any persistent computing system can be constructed by adding various functional components to the package.

Acknowledgements

The work presented in this paper was supported in part by a grant from The Telecommunications Advancement Foundation, Japan, a grant from CASIO Science Promotion Foundation, Japan, a grant from Support Center for Advanced Telecommunications Technology Research, Japan, and a grant from Japan Society for the Promotion of Science under Grant-in-Aid for Scientific Research (B) No. 18300005.

References

[1] R. J. Abbott, "Resourceful Systems for Fault Tolerance, Reliability, and Safety," ACM Computing Surveys, Vol. 22, No. 1, pp. 35-68, 1990.

[2] M. V. Butz, O. Sigaud, and P. Gerard, "Anticipatory Behavior: Exploiting Knowledge About the Future to Improve Current Behavior," in M. V. Butz, O. Sigaud, and P. Gerard (Eds.), "Anticipatory Behavior in Adaptive Learning Systems: Foundations, Theories, and Systems," Lecture Notes in Artificial Intelligence, Vol. 2684, pp. 1-10, Springer-Verlag, 2003.

[3] 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, 1993.

[4] J. Cheng, "Entailment Calculus as the Logical Basis of Automated Theorem Finding in Scientific Discovery," in "Systematic Methods of Scientific Discovery - Papers from the 1995 Spring Symposium," AAAI Technical Report SS-95-03, pp. 105-110, 1995.

[5] J. Cheng, "EnCal: An Automated Forward Deduction System for General-Purpose Entailment Calculus," in Advanced IT Tools, IFIP World Conference on IT Tools, IFIP96 - 14th World Computer Congress, edited by N. Terashima and E. Altman, Chapman & Hall, pp. 507-514, 1996.

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

[7] J. Cheng, "Wholeness, Uncertainty, and Self-Measurement: Three Fundamental Principles in Concurrent Systems Engineering," in Proc. 13th International Conference on Systems Engineering, pp. CS-7-CS-12, 1999.

[8] J. Cheng, "Autonomous Evolutionary Information Systems," Wuhan University Journal of Natural Sciences, Vol. 6, No. 1-2, Special Issue: Proceedings of the International Software Engineering Symposium 2001, pp. 333-339, 2001.

[9] J. Cheng, "Anticipatory Reasoning-Reacting Systems," in Proc. International Conference on Systems, Development and Self-organization, pp. 161-165, 2002.

[10] J. Cheng, "Temporal Relevant Logic as the Logical Basis of Anticipatory Reasoning-Reacting Systems," in D. M. Dubois (Ed.), "Computing Anticipatory Systems: CASYS 2003 - Sixth International Conference," AIP Conference

Proceedings 718, pp. 362-375, The American Institute of Physics, 2004.
[11] J. Cheng, "Connecting Components with Soft System Buses: A New Methodology for Design, Development, and Maintenance of Reconfigurable, Ubiquitous, and Persistent Reactive Systems," in Proc. 19th International Conference on Advanced Information Networking and Applications, Vol. 1, pp. 667-672, IEEE Computer Society, 2005.
[12] J. Cheng, "Comparing Persistent Computing with Autonomic Computing," in Proc. 11th International Conference on Parallel and Distributed Systems, Vol. II, pp. 428-432, IEEE Computer Society, 2005.
[13] J. Cheng, "Autonomous and Continuous Evolution of Information Systems," in Knowledge-Based Intelligent Information & Engineering Systems, 9th International Conference, edited by R. Khosla, R. J. Howlett, and L. C. Jain, Lecture Notes in Artificial Intelligence, Vol. 3681, pp. 758-767, Springer-Verlag, 2005.
[14] J. Cheng, "Persistent Computing Systems as Continuously Available, Reliable, and Secure Systems," in Proc. 1st International Conference on Availability, Reliability and Security, pp. 631-638, IEEE Computer Society, 2006.
[15] D. M. Dubois, "Computing Anticipatory Systems with Incursion and Hyperincursion," in D. M. Dubois (Ed.), "Computing Anticipatory Systems: CASYS - First International Conference," AIP Conference Proceedings 437, pp. 3-29, The American Institute of Physics, 1998.
[16] D. M. Dubois, "Introduction to Computing Anticipatory Systems," International Journal of Computing Anticipatory Systems, Vol. 2, pp. 3-14, 1998.
[17] D. M. Dubois, "Review of Incursive, Hyperincursive and Anticipatory Systems - Foundation of Anticipation in Electromagnetism," in D. M. Dubois (Ed.), "Computing Anticipatory Systems: CASYS'99 - Third International Conference," AIP Conference Proceedings 517, pp. 3-30, The American Institute of Physics, 2000.
[18] D. M. Dubois, "Mathematical Foundations of Discrete and Functional Systems with Strong and Weak Anticipations," in M. V. Butz, O. Sigaud, and P. Gerard (Eds.), "Anticipatory Behavior in Adaptive Learning Systems: Foundations, Theories, and Systems," Lecture Notes in Artificial Intelligence, Vol. 2684, pp. 110-132, Springer-Verlag, 2003.
[19] E. Dustin, "Effective Software Testing: 50 specific ways to improve your testing," Addison-Wesley, 2003.
[20] Y. Goto, S. Nara, and J. Cheng, "Efficient Anticipatory Reasoning for Anticipatory Systems with Requirements of High Reliability and High Security," International Journal of Computing Anticipatory Systems, Vol. 14, pp. 156-171, 2004.
[21] D. Harel and A. Pnueli, "On the Development of Reactive Systems," in Logics and Models of Concurrent Systems, edited by K. R. Apt, Springer-Verlag, pp. 477-498, 1985.
[22] H. Hecht, "Fault-Tolerant Software for Real-Time Applications," ACM Computing Surveys, Vol. 8, No. 4, pp. 391-407, 1990.
[23] D. S. Hermann, "Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors," IEEE-CS Press, 1999.
[24] IEEE-CS, IEEE Standard 610, "IEEE Standard Computer Dictionary – A

Compilation of IEEE Standard Computer Glossaries," 1990.
[25] IEEE-CS, IEEE Standard 610.12-1990, "IEEE Standard Glossary of Software Engineering Terminology," 1990.
[26] M. R. Lyu (ed.), "Handbook of Software Reliability Engineering," McGraw-Hill, 1996.
[27] J. J. Marciniak (ed.), "Encyclopedia of Software Engineering," John Wiley & Sons, New York / Chichester / Brisbane / Toronto / Singapore, 1994.
[28] M. Nadin, "Anticipation - A Spooky Computation," International Journal of Computing Anticipatory Systems, Vol. 6, pp. 3-17, 2000.
[29] M. Nadin, "Anticipatory Computing," Ubiquity - The ACM IT Magazine and Forum, Views - Vol. 1, Issue 40, 2000.
[30] M. Nadin, "Not Everything We Know We Learned," in M. V. Butz, O. Sigaud, and P. Gerard (Eds.), "Anticipatory Behavior in Adaptive Learning Systems: Foundations, Theories, and Systems," Lecture Notes in Artificial Intelligence, Vol. 2684, pp. 23-43, Springer-Verlag, 2003.
[31] A. Pnueli, "Specification and Development of Reactive Systems," in Information Processing 86, edited by H.-J. Kugler, IFIP, North-Holland, pp. 845-858, 1986.
[32] P. Rook (ed.), "Software Reliability Handbook," Elsevier, London / New York, 1990.
[33] R. Rosen, "Anticipatory Systems - Philosophical, Mathematical and Methodological Foundations," Pergamon Press, Oxford, 1985.
[34] F. Shang and J. Cheng, "Towards Implementation of Anticipatory Reasoning-Reacting System," International Journal of Computing Anticipatory Systems, Vol. 14, 93-109, 2004.
[35] 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.
[36] J. J. Torres-Carbonell, J. Parets-Llorca, and D. M. Dubois, "Software Systems Evolution, Free Will and Hyperincursivity," International Journal of Computing Anticipatory Systems, Vol. 12, pp. 3-22, 2002.
[37] M. Weiser, "Some Computer Science Problems in Ubiquitous Computing," Communications of the ACM, Vol. 36, No. 7, pp. 75-84, 1993.
[38] J. A. Whittaker, "What Is Software Testing? And Why Is It So Hard?," IEEE-CS Software, Vol. 17, No. 1, pp. 70-79, 2000.
[39] L. Wos, "Automated Reasoning: 33 Basic Research Problems," Prentice-Hall, 1988.
[40] L. Wos, "The Problem of Automated Theorem Finding," Journal of Automated Reasoning, Vol. 10, No. 1, pp. 137-138, 1993.
[41] A. Y. H. Zomaya (ed.), "Parallel & Distributed Computing Handbook," McGraw-Hill, 1996.