Continuous Reactability of Persistent Computing Systems

Yuichi Goto, Takumi Endo, and Jingde Cheng
Department of Information and Computer Sciences, Saitama University
Saitama,338-8570, Japan
{gotoh, endo, cheng}@aise.ics.saitama-u.ac.jp

Abstract
Persistent computing systems are an infrastructure of computing anticipatory systems. The reactability of a persistent computing system, which is how many reactions of the system are active at a certain time, is the most important property to characterize the system. On the other hand, to be anticipatory, the reactability of a computing anticipatory system must be continuous. This paper proposes the first method to measure the continuous reactability of a persistent computing system in a unified way. The continuous reactability of a persistent computing system is a new concept of computing systems, so that it will be raise new research problems of computing anticipatory systems as well as persistent computing systems.
Keywords : Computing anticipatory system, Persistent computing system, Component-based system, Reactability, Continuous reactability

1 Introduction

The notion of anticipatory system [12], in particular, computing anticipatory system [6, 7, 8], implies a fundamental assumption or requirement, i.e., to be anticipatory, a computing system must behave continuously and persistently without stopping its running, because (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 [5]. Cheng and Shang have showed that persistent computing systems should be as an infrastructure of computing anticipatory systems [5]. A persistent computing system is a reactive system that functions continuously anytime without stopping its reactions even when it needs to be maintained, upgraded, or reconfigured, it has some trouble, or it is attacked [2, 3, 4].

We proposed the *reactability* of a persistent computing system, which is how many reactions of the system are active at a certain time, as one of the most important properties to characterize the system [9]. The most fundamental issue towards implementation of a persistent computing system is how to measure and maintain the reactability of the system. However, our definition of the reactability in [9] is not appropriate because our definition of a reaction of a persistent computing system is not appropriate.

On the other hand, to be anticipatory, the reactability of a computing anticipatory system must be continuous. However, the requirement that a computing system should

Edited by D. M. Dubois, CHAOS, Liège, Belgium, ISSN 1373-5411 ISBN 2-930396-07-5

run continuously and persistently is never taken into account as an essential and/or general requirement by traditional system design and development methodologies. As a result, there is no clearly defined standard to be used for measuring the continuous reactability of a computing system is also not clearly defined.

This paper proposes the first method to measure the continuous reactability of a persistent computing system in a unified way. At first, we re-define the reactability and define the continuous reactability of a persistent computing system. We then propose a method to measure the continuous reactability of a persistent computing system.

The rest of this paper is organized as follows: Section 2 explains what is a persistent computing system briefly. Section 3 presents definitions of the reactability and continuous reactability of a persistent computing system. Section 4 presents a method to measure the continuous reactability. Section 5 gives discussions. Some concluding remarks are given in Section 6.

2 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 [2, 3, 4]. 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 [10, 11]. 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 [2, 3, 4]. 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 state, partially functional state, and disfunctional state. In the functional state, the system can function completely; in a partially functional state, the system can function partially but not completely; in the 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 state, partially reactive state, and dead state. In the reactive state, the system can react completely; in a partially reactive state, the system can react partially but not completely; in the dead state, the system cannot react at all. Therefore, a system in the functional state must be also in the reactive state; a system in a partially functional may be in either the reactive state or a partially reactive state; a system in the disfunctional state may be in either the reactive state, a partially reactive state, or the dead state.

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

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 Reactability and Continuous Reactability

Now, let us discuss *reactions* of a persistent computing system in more detail. We consider that a reaction of a persistent computing system to a stimulus from its outside environment may be any one of the following three kinds of actions: outputting data to the outside environment, changing current state of the system, and ignoring the stimulus. There are some unexpected phenomena in a persistent computing system because of hardware troubles, software bugs, deadlock, livelock, mistakes of specification of the system, and so on, if the system cannot react to a stimulus. Note that there may be some unexpected phenomena in a persistent computing system although the system can react to all stimuli.

On the other hand, the definition of a component proposed by Szyperski is "A software *component* is a unit of composition with contractually specified interfaces and explicit context dependencies only" [13]. An *interface* is an abstraction of the behavior of a component that consists of a subset of interactions that component together with a set of constraints describing when they may occur. A component can have one or more interfaces, and performs its operations via their interfaces. An *operation* performed via its interface is most primitive facility in persistent computing systems. A persistent computing system is constructed by components and binding their interfaces appropriately at the configuration level, and overall behavior of the system can be defined as sets of operations with partial order. A *reaction* to stimuli from the outside environment of a persistent computing system is constructed by performing one or more operations in partial order. From the viewpoint of software systems, any stimulus which comes from outside environment of a system can be represented as character strings. Therefore, a reaction starts when character strings represented as the stimuli are inputted into an interface of an operation in the reaction. The reaction ends when all operations in the reaction have been finished. In addition, a reaction in a certain state of a system may be different from a reaction in an other state of the system although a given stimulus is same.

Under the above considerations, we define the notion of reaction as follows. A reac-

tion r of a persistent computing system is a 4-tuple (q, s, e, Θ). q is the current state of the system. $s \in \Theta$ is a start operation which receives character strings represented as stimuli from the outside environment. $e \in \Theta$ is an end operation: outputting character strings to the outside environment, changing the system's current state q to $q' \in Q$ where Q is the set of all states of the system, or not doing anything. Θ is a set of operations which the reaction r consists of. In a well-designed and well-implemented persistent computing system, it is possible to decide a set of character strings represented as stimuli from the outside environment which make a reaction r start if s, e, and Θ of r can be decided. Thus, we do not have to consider stimuli from outside environment of a system. Note that we do not consider how to design and implement a persistent computing system well in this paper. Our previous definition of a reaction of a persistent computing system in [9] does not take into account states of the system, so that we cannot enumerate all reactions in a persistent computing system according to the previous definition.

The definitions of an active and inactive reaction are as follows;

> a reaction $r = (q, s, e, \Theta)$ is *active* if and only if all operations of Θ are executable,
>
> a reaction $r = (q, s, e, \Theta)$ is *inactive* if and only if at least one operation of Θ is unexecutable.

We re-define the reactability of a persistent computing system by using the above definition of a reaction. *The reactability of a system is the ratio the number of all active reactions to that of all reactions in the system at a certain time.* The reactability is represented as follows;

$$\text{The reactability} = \frac{\text{The number of active reactions}}{\text{The number of all reactions}}. \tag{1}$$

We also represent states of a running persistent computing system at a certain time as follows:

> a system is in the reactive state if all reactions are active in the system, that is, the reactability of the system is 1,
>
> a system is in a partially reactive state if both active reactions and inactive reactions exist in the system, that is, the reactability of the system is more than 0 and less than 1,
>
> a system is in the dead state if all reactions are inactive in the system, that is the reactability of the system is 0.

We introduce a new concept, "*continuous reactability of a system*". *The continuous reactability of a system is how high the reactability of the system is continuously for a certain time interval.* The continuous reactability of a system is represented 2-tuple (m, σ^2) where m is the mean value of observed reactabilities of the system during a certain time interval and σ^2 is the variance of the observed reactabilities. The variance shows

how much the reactability changed during a certain time interval. For computing anticipatory systems, it is important to know how many reactions of a computing anticipatory system are active for a certain time interval. To be anticipatory, a computing anticipatory system should be able to not only predict future events but also react to the future events. However, not all computing anticipatory systems can predict future events accurately, so that such a system should be able to react to the future events at any time during a period when the events may occur. For example, there is a computing anticipatory system which put up an umbrella when it starts raining. If the system can predict "it will start raining at 7:00am tomorrow," then it is enough for the system to be able to put up its umbrella at 7:00am although the system cannot do so before 7:00am. By contrast, if the system can predict only "it will start raining tomorrow," then the system should be able to put up its umbrella anytime tomorrow. Furthermore, there may be some kinds of future events which any computing anticipatory system cannot accurately predict.

4 A Measurement Method

In this paper, we propose a method to measure the continuous reactability of a system in the past. The continuous reactability of a system can be estimated from observed reactabilities of the system for a period when we want to measure it. It is easy to know the continuous reactability of a system in the past by using observed reactabilities of the system for a certain period of time in the past. By contrast, it is difficult to know the continuous reactability of a system in the future because it is necessary to predict the reactabilities of the system for a certain period of time in the future. To propose a method to measure the continuous reactability of a system in the future is a future work.

To measure the continuous reactability of a system in the past, a method to measure the present reactability of a system is necessary. To measure the present reactability of a system, it should be able to enumerate all reactions in the system and to count active reactions.

First, we explain a method to enumerate all reactions in a system. It is possible to enumerate all reactions in a system by using control flow relations among operations. O is a set of all operations of a system. $O_s \subseteq O$ is a set of all source operations. $O_e \subseteq O$ is a set of all sink operations. From the viewpoint of control flow, a source operation is an operation which no operation calls or no operation sends a message to; a sink operation is an operation which does not call or send message to any operation. A start operation of a reaction is an element of O_s, and an end operation of a reaction is an element of O_e. $Q =_{def} \{q_i\}$ $(1 \quad i \quad n)$ is a set of all states of a system where n is a natural number. $\rho_i =_{def} \{r_1, \ldots, r_k\}$ is a set of all reactions in a certain state q_i $(1 \quad i \quad n)$ where r_j $(1 \quad j \quad k)$ is a reaction in q_i and k is a natural number. $\rho_i \cap \rho_j = \emptyset$ $(1 \quad i, j \quad n, i \neq j)$. $R =_{def} \bigcup_{i=1}^{n} \rho_i$ is a set of all reactions of the system. A procedure to enumerate all reactions in a system is as follows;

1. in a state $q_i \in Q$ (i is a natural number), draw a nondeterministic parallel control-flow net [1], CFN for short, where $s \in O_s$ is as the start vertex and $e \in O_e$ is as the

termination vertex,

2. enumerate all reactions in the CFN,

3. enumerate all reactions whose start operations are s in q_i by repeating above processes for all CFNs whose start vertices are s in q_i,

4. enumerate all reactions of ρ_i by repeating above processes for all source operations of O_s,

5. enumerate all reactions of R by repeating above processes for all states of Q.

We can assume that it is possible to know Q, O, O_s, and O_e from configuration files of a system.

A CFN is a directed graph which is proposed to represent control flow in concurrent programs as well as sequential programs [1]. A CFN is a 10-tuple $(V, N, P_F, P_J, A_C, A_N, A_{PF}, A_{PJ}, s_v, t_v)$. V, N, P_F, and P_J are finite sets of vertices. $N \subset V$ is a set of elements, called nondeterministic selection vertices. $P_F \subset V$ ($N \cap P_F = \emptyset$) is a set of elements, called parallel execution fork vertices. $P_J \subset V$ ($N \cap P_J = \emptyset$ and $P_F \cap P_J = \emptyset$) is a set of elements, called parallel execution join vertices. $s_v \in V$ is a unique vertex, called a start vertex, such that the in-degree of s_v is 0. $t_v \in V$ is a unique vertex, called a termination vertex, such that the out-degree of t_v is 0. A_C, A_N, A_{PF}, and A_{PJ} are sets of arcs, such that $A_C \subseteq V \times V$, $A_N \subseteq N \times V$, $A_{PF} \subseteq P_F \times V$, and $A_{PJ} \subseteq V \times P_J$. For any $v \in V$ ($v \neq s_v$, $v \neq t_v$), there exists at least one path from s_v to v and at least one path from v to t_v. Any arc $(v_1, v_2) \in A_C$ is called a sequential control arc, any arc $(v_1, v_2) \in A_N$ is called a nondeterministic selection arc, and any arc $(v_1, v_2) \in A_{PF} \cup A_{PJ}$ is called a parallel execution arc. For any arc $(v_1, v_2) \in A_C \cup A_N \cup A_{PF} \cup A_{PJ}$, v_1 is said to be adjacent to v_2, and v_2 is said to be adjacent from v_1. A predecessor of a vertex v is a vertex adjacent to v, and a successor of v is a vertex adjacent from v. A dominator of a vertex v is a vertex dominates v. v_1 dominates v_2 if every path from the entry that reaches v_2 has to pass through v_1. A sub-CFN of a CFN $(V, N, P_F, P_J, A_C, A_N, A_{PF}, A_{PJ}, s_v, t_v)$ is a 10-tuple $(V', N', P'_F, P'_J, A'_C, A'_N, A'_{PF}, A'_{PJ}, s_v, t_v)$ where V', N', P'_F, P'_J, A'_C, A'_N, A'_{PF}, and A'_{PJ} are subsets of V, N, P_F, P_J, A_C, A_N, A_{PF}, and A_{PJ}, respectively. For any $v \in V'$ ($v \neq s_v$, $v \neq t_v$), there exists at least one path from s_v to v and at least one path from v to t_v. A CFN or its sub-CFN has choice structure if there is at least one vertex which more than one sequential control arc departs from, and has merge structure if there is at least one vertex which more than one sequential control arc arrives at. A CFN or its sub-CFN has iteration structure if there is at least one vertex whose successor is its dominator.

Control flow relations among operations in a persistent computing system are classified into 3 kinds: sequential, fork, join. Let $o_1, \ldots, o_i \in O$ be operations. An ordered pair (o_1, o_2) is sequential relation if and only if o_2 is called and executed after o_1 finished, or o_1 sends a message (or event) to o_2. A tuple $(o_1, \ldots, o_i)$ is fork relation if and only if $o_2, \ldots$, and o_i are called and executed concurrently after o_1 finished, or o_1 multicasts a message (or event) to $o_2, \ldots$, and o_i. A tuple $(o_1, \ldots, o_i)$ is join relation if and only if o_i

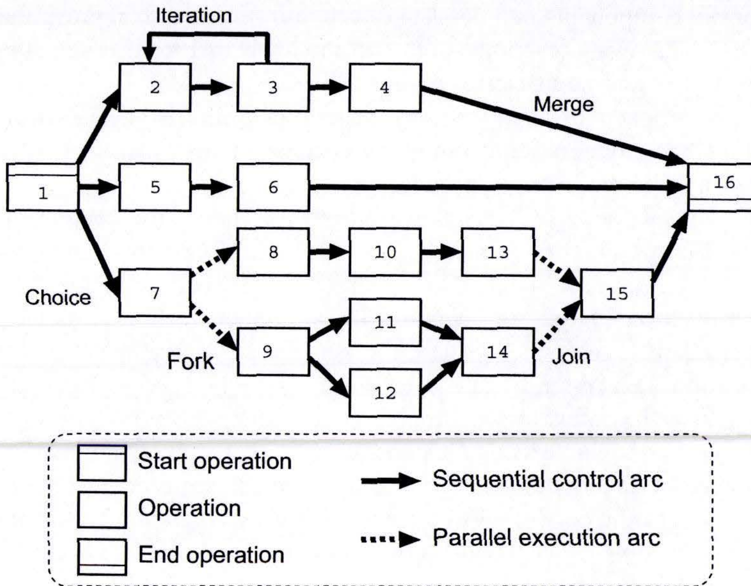

Fig. 1: A CFN to represent control flow relation among operations

is called and executed after o_1, ..., and o_{i-1} finished, or o_i is executed after it received all messages (or events) from $o_1,...,o_{i-1}$. In a CFN to represent the control flow relations among operations, the start vertex is a source operation $s \in O_s$ and the termination vertex is a sink operation $e \in O_e$; if a ordered pair (o_1, o_2) is sequential relation then $o_1, o_2 \in V$; if a tuple $(o_1, \ldots, o_i)$ is fork relation then $o_1 \in P_F$ and $o_1, \ldots, o_i \in V$; if a tuple $(o_1, \ldots, o_i)$ is join relation then $o_1, \ldots, o_i \in V$ and $o_i \in P_J$. Note both of N and A_{PJ} are $\emptyset$ because the control flow relations among operations are deterministic. In a state $q \in Q$, there is at least one reaction whose start operation is $s \in O_s$ and end operation is $e \in O_e$ if and only if it is possible to draw a CFN where s is as the start vertex and e is as the termination vertex.

To represent a reaction in the framework of CFN, we define a reaction processing net, RPN for short, of a CFN. It is a sub-CFN of the CFN which does not have both choice structure and merge structure. All operations which constitute a reaction are all operations of a RPN. The number of reactions in a CFN is the number of RPN in the CFN. Fig. 1 shows a CFN. In the CFN, there are 4 RPNs. Sets of operations of the RPNs are {1, 2, 3, 4, 16}, {1, 5, 6, 16}, {1, 7, 8, 9, 10, 11, 13, 14, 15, 16}, and {1, 7, 8, 9, 10, 12, 13, 14, 15, 16}. By finding all of RPNs in a CFN, we can enumerate all reactions in a CFN.

Second, we explain a method to count active reactions in a system. We consider that targets to be measured (tracked) should be interactions among participating components whose operations consist of a reaction, such that monitor components, which are devel-

opment as control components in a persistent computing system, can grasp the progress of operations on the relevant reaction and reason about whether the every operations from the start-point to the end-point can complete their executions or not, i.e., whether the reaction is active or inactive. Note we propose the most primitive design of reactability measurement facilities for persistent computing systems, in this paper, we do not consider reliability, security, and real-time properties.

When the start operation of a reaction starts its execution, the context information can be informed to monitor components. Here context information contains a 8-tuple: $(\mathrm{ID}_R, T_R, \mathrm{ID}_O, \mathrm{ID}_C, \mathrm{ID}_{PO}, \mathrm{ID}_{PC}, \mathrm{ID}_{SO}, \mathrm{ID}_{SC})$. They mean respectively, a reaction ID, the time when the reaction started, an operation ID, a component ID, the predecessor operation ID, the ID of a component where the predecessor operation is performed, the successor operation ID, and the ID of a component where the successor operation is performed. Note that such context information never contain payload (body) of the message. Similarly, when a component executes an operation to interact with an other component, only the tuple are sent to monitor components. Monitor components receive such context information and check the conditions of a reaction by analyzing them. To decide whether a operation in a reaction is executable or not, monitor components check two context information: one of operations in the source component (below "source component's information"), other one of operations in the destination component (below "destination component's information"). Monitor components check whether ID_R, T_R, ID_O, ID_C, ID_{SO} and ID_{SC} in source component's information is equal to ID_R, T_R, ID_{PO}, ID_{PC}, ID_O, and ID_C in destination component's information or not. After that, the monitor components calculates difference between compare time of receipt of them. An operation can be regarded as unexecutable if the check cannot be passed and/or the difference differs vastly. As a result, monitor components can grasp the number of active reactions.

However, by using such "reactive sensing", monitor components can not aware that an operation in the reaction is in unexecutable state till the start operation can be executed. Ideally, it is necessary for monitor components to awake soon after an operation is unexecutable for some reasons (e.g., failures or attacks). As a practical scheme to solve this issue, we require "proactive sensing," meaning that monitor components send query to an operation periodically and then receive the ACK from the operation in such a way the responsible operation receives and sends the messages via the contractually specified interfaces. Since the interfaces correspond operations consisting of reactions, control components can awake that the relevant operation is unexecutable if the ACK can be sent to them in an allowable time.

According to those methods, we can measure the reactability of a persistent computing system at a current time. We can also estimate the continuous reactability of the system in the past from the observed reactabilities of the system for a certain time interval in the past. The proposed measurement method does not depend on persistent computing systems, so that it is possible to measure the reactability and the continuous reactability of a component-based reactive system in the past as well as a persistent computing system by using the measurement method.

5 Discussion

We have not considered reliability, security, and real-time properties in the proposed method. From the viewpoint of reliability, we should investigate how to check whether an operation is executable or not if the operation behaves irregularly because of its bugs. In order to improve the reliability of a persistent computing system, it is natural to put more than one component which can provide same operation in a persistent computing system. In such case, we should investigate how to check whether an operation which more then one component can provide in a system is executable or not. From the viewpoint of security, even if more than one component which can provide same operation exists in a persistent computing system, it may have to use only a particular component. In such case, we should investigate how to measure the reactability and the continuous reactability of the system. Moreover, it is possible to measure the reactability and the continuous reactability of a system which has malicious components. From the viewpoint of real-time operation, we should take into account turnaround time about a reaction of a persistent computing system when the reaction must be finished until a certain time. Monitor components to measure the reactability of a system may become a bottleneck of the performance of the system, such that we should put several or more monitor components on the system. Thus, we should investigate how we keep the consistency of information about reactions of the system among those monitor components.

Most important and challenging issue is how can we measure the reactability and the continuous reactability of a persistent computing system in the future and how can we maintain enough reactability and continuous reactability in the future. We might take into account the probability of hang-up of components or operations because of specification faults and bugs, the reliability and performance of hardwares where components are working and connection channels among components, the probability of occurrence of maintain components, the probability of hang-up of components or operations because of attacks or accidents, the probability of hang-up of components or operations because of occurrences which no one cannot think about in development phase of the system. It is difficult to predict above things, so that how to measure the reactability and the continuous reactability is really a challenging problem.

6 Concluding Remarks

In this paper, we re-defined the reactability and defined the continuous reactability of a persistent computing system, and presented a method to measure the continuous reactability in the past. The proposed measurement method does not depend on persistent computing systems, so that it is possible to measure the reactability and the continuous reactability of a component-based reactive systems in the past as well as a persistent computing system by using the measurement method. Soft system bus based methodology is proposed in order to build persistent computing systems [2]. A system built using this methodology is called soft system bus based system. The soft system bus based system

is expected as a persistent computing system. We will therefore apply our measurement method to the soft system bus based system and evaluate whether the method is useful or not. On the other hand, we have not considered reliability, security, and real-time properties in the proposed measurement method. It is necessary to improve the measurement method from view point of reliability, security, and real-time properties.

The continuous reactability of a system is a new concept of computing systems, so that it will be raise new research problems of computing anticipatory systems as well as persistent computing systems. For example, how to measure the reactability and the continuous reactability of a system in the future is a challenging problem.

References

[1] Cheng, J.: Nondeterministic Parallel Control-Flow / Definition-Use Nets and Their Applications. In Joubert, G. R., Trystram, D., Prters, F. J., Evans, D. J., eds.: Parallel Computing: Trends and Applications, Proceedings of the International Conference, ParCo93, Grenoble, France, 7-10 September 1993. Elsevier B.V. (1994) 589-592

[2] Cheng, J.: Connecting Components with Soft System Buses: A New Methodology for Design, Development, and Maintenance of Reconfigurable, Ubiquitous, and Persistent Reactive Systems. In: Proc. of 19th International Conference on Advanced Information Networking and Applications, IEEE Computer Society Press (2005) 667–672

[3] Cheng, J.: Comparing Persistent Computing with Autonomic Computing. In: Proc. 11th International Conference on Parallel and Distributed Systems, IEEE Computer Society Press (2005) 428–432

[4] Cheng, J.: Persistent Computing Systems as Continuously Available, Reliable, and Secure Systems. In: Proc. 1st International Conference on Availability, Reliability and Security, IEEE Computer Society Press (2006) 631–638

[5] Cheng, J., Shang, F.: Persistent Computing Systems as An Infrastructure of Computing Anticipatory Systems. International Journal of Computing Anticipatory Systems **18** (2006) 61–74

[6] Dubois, D.M.: Computing Anticipatory Systems with Incursion and Hyperincursion. In Dubois, D.M., ed.: Computing Anticipatory Systems: CASYS - First International Conference. AIP Conference Proceedings 437., The American Institute of Physics (1998) 3–29

[7] Dubois, D.M.: Introduction to Computing Anticipatory Systems. International Journal of Computing Anticipatory Systems **2** (1998) 3–14

[8] Dubois, D.M.: Review of Incursive, Hyperincursive and Anticipatory Systems - Foundation of Anticipation in Electromagnetism. In Dubois, D.M., ed.: Computing Anticipatory Systems: CASYS'99 - Third International Conference. AIP Conference Proceedings 517, The American Institute of Physics (2000) 3–30

[9] Endo, T., Goto, Y., Cheng, J.: Measuring Reactability of Persistent Computing Systems. In Lumpe, M., Vanderperre, W., eds.: Software Composition: 6th International Symposium, SC 2007, Braga, Portugal, March 24-25, 2007, Revised Selected Papers. Volume 4829 of Lecture Notes in Computer Science., Springer-Verlag (2007) 144–151

[10] Harnel, D., Pnueli, A.: On The Development of Reactive Systems. In Apt, K.R., ed.: Logic and Models of Concurrent Systems. Springer-Verlag (1989) 477–498

[11] Pnueli, A.: Specification and Development of Reactive Systems. In Kugler, H.J., ed.: Information Processing 86, North-Holland/IFIP (1986) 845–858

[12] Rosen, R.: Anticipatory Systems - Philosophical, Mathematical and Methodological Foundations. Pergamon Press (1985)

[13] Szyperski, C., Gruntz, D., Murer, S.: Component Software, Beyond Object-Oriented Programming, Second Edition. ACM Press/Addison-Wesley Publishing Co. New York (2002)