DEOS-FY2011-WP-03E

JST-CREST

Research Area

"Dependable Operating Systems for Embedded Systems Aiming at Practical Applications"

DEOS Project

White Paper

Version 3.0a

2011/12/15

Mario Tokoro
Research Supervisor
(Sony Computer Science Laboratories, Inc.)

Japan Science and Technology Agency

(Intentionally left blank)

Preface to Version 3.0

It has been one year since we published the DEOS Project White Paper Version 2.0 on December 1, 2010. The project is progressing in accordance with the plan described in the White Paper Version 2.0, and implementations of the technology as well as software development have started. We are pleased to publish this third version of the DEOS Project White Paper to present our recent progress.

Changes to chapters of Version 3.0 from Version 2.0

Version 2.0	
Chapter 1	Background
Chapter 2	Dependability
Chapter 3	Phases of DEOS Process
Chapter 4	Project Direction
Chapter 5	Major Research and Development Status
Chapter 6	Research and Development Organization
Chapter 7	Roadmap
Chapter 8	Further Issues for Practical Applications
Chapter 9	References
Chapter 10	Appendix

Version 3.0	
Chapter 1	Introduction
Chapter 2	Dependability
Chapter 3	Achieving Open Systems Dependability
Chapter 4	Requirements Management in DEOS Process
Chapter 5	The DEOS Runtime Environment for Flexible Control
Chapter 6	DEOS Development Support Tools
Chapter 7	Effectiveness of Applying DEOS
Chapter 8	Steps toward Practical Use
Chapter 9	References
Chapter 10	Appendix

Contributors

Shigeru Matsubara	Dependable Embedded OS R&D Center, JST
Tomohiro Miyahira	Dependable Embedded OS R&D Center, JST
Kiyoshi Ono	Dependable Embedded OS R&D Center, JST
Hiroki Takamura	Dependable Embedded OS R&D Center, JST
Makoto Yashiro	Dependable Embedded OS R&D Center, JST
Kenji Kono	Keio University
Jin Nakazawa	Keio University
Hideyuki Tokuda	Keio University
Hiroshi Yamada	Keio University
Shuichiro Yamamoto	Nagoya University
Yoichi Ishiwata	National Institute of Advanced Industrial Science and Technology
Satoshi Kagami	National Institute of Advanced Industrial Science and Technology
Yoshiki Kinoshita	National Institute of Advanced Industrial Science and Technology
Kenji Taguchi	National Institute of Advanced Industrial Science and Technology
Toshinori Takai	National Institute of Advanced Industrial Science and Technology
Makoto Takeyama	National Institute of Advanced Industrial Science and Technology
Mario Tokoro	Sony Computer Science Laboratories, Inc.
Hajime Fujita	University of Tokyo
Yutaka Ishikawa	University of Tokyo
Toshiyuki Maeda	University of Tokyo
Yutaka Matsuno	University of Tokyo
Yasuhiko Yokote	University of Tokyo
Taisuke Boku	University of Tsukuba
Toshihiro Hanawa	University of Tsukuba
Shuichi Oikawa	University of Tsukuba
Mitsuhisa Sato	University of Tsukuba
Tatsuo Nakajima	Waseda University
Kimio Kuramitsu	Yokohama National University
Midori Sugaya	Yokohama National University

(Members are listed in alphabetical order by family name for each organization.
Organizations are listed in alphabetical order by organization name.)

This project is supported by CREST, Japan Science and Technology Agency

Table of Contents

1. Introduction

1.1. Background and Objectives

Information systems today have key roles in our daily life. We receive tremendous benefit from information systems such as mobile phones, services available through websites in the internet, weather forecast systems, public services such as traffic control, automatic train and airplane turnstiles, ticket reservation systems, vehicle operation control systems, distribution management systems, and business systems such as financial systems. In many cases, those information systems are connected to one another directly or indirectly, forming large scale information systems, which work as infrastructure to support our daily life. The stronger the dependency of our daily life on information systems is, the more important their reliability and robustness are. In this White Paper, the attribute of the information systems that it provides services continuously is called dependability. This involves both reliability and safety.

The typical information systems today have a wide range of devices, such as PCs, tablets and smart phones, connected to servers. Services are provided by the servers through a network to the end-users. Up until now, most development processes of information systems have adopted the common practice of creating a reliable development plan in advance, determining in detail the product or the system specifications, and going through the cycle of design, implementation and verification before the products or the systems come into use. This process is quite effective for the development of systems, which have specifications and behavior that are well defined and predictable at the beginning of development. However, for the large systems described above, it is almost impossible to provide system specifications at the beginning of system development, which anticipate and deal with all of the future changes in objectives and environments. Also, in large systems development, it is common today to utilize software such as existing software developed in the past, software provided by other companies, and software or its services available through networks. This means that parts of these systems may not have proper specifications or development documents.

Large systems are usually in use for a long time. Hence, these systems need to change adapting to changes in users' requirements, while keeping up operations for needed services. Adaptations to changes in hardware or networks may also be necessary to keep up these operations, and to accommodate to changes in requirements resulting from evolution of technology or business needs, or to an increase or decrease in the number of users. These will make it more difficult to maintain the dependability of today's systems. It is also required to take appropriate action in the event of a system crash caused by attacks with viruses, etc., or an information leakage caused by unauthorized access. Unfortunately, failures of important information systems have been reported from all over the world. The failure of information systems causes not only considerable loss of benefits to users, but also the loss of business chances, involving the payment of compensation for damage in some cases the loss of brand image, and even to the discontinuation of the business. We should assume the service provider is obliged to do its best to avoid system failures and to be accountable if failure occurs.

The analysis of those failures shows that major causes include the system being developed without sufficient understanding of the behavior of all of the components, the usage of the system exceeding the initial design limit, and inconsistency within the system caused by a system change to accommodate change in users' requirements while the system is in operation. These cases indicate that modern computer systems should not be assumed to be a system whose function, structure, and system boundaries are fixed. Their design, their implementation, and the way they are used should be handled accordingly, as systems which grow and change over time.

The objectives of this project are to develop a method to continuously provide services meeting users' requirements by accommodating the system to ever-changing objectives and environments.

Described in this paper are the insufficiency of current dependability attainment technology for today's systems which grow and change overtime, new concepts of "Open Systems Dependability" as a basis of a new development method, a new iterative method called the DEOS process, the architecture to realize this iterative process, and the required technologies to support the process and architecture [1, 4, 5, 6 and 12].

1.2. DEOS Project: Outline, Goal, and Deliverables

The DEOS project to develop "Dependable Operating Systems for Embedded Systems Aiming at Practical Applications", started in October, 2006 as one of the research areas supported by JST/CREST (Japan Science and Technology/Core Research for Evolutional Science and Technology). Recently, most embedded systems are connected to servers through the Internet, and the services to end users are provided by the whole network system. We defined embedded systems as special-purpose systems (as opposed to general-purpose systems) and set the project goal to be development of dependable operating systems for these special-purpose systems. "Operating systems" here refer to the system software as well as the operating systems, and includes software tools used to build the systems.

As a result of deep discussions by project members on dependability, we have concluded that because today's large and complicated systems need to keep changing to accommodate ever-changing objectives and environments, it is appropriate to understand the systems as "Open Systems" rather than "Closed Systems" which most of the traditional development methods assume, and that dependability of open systems should be achieved by an iterative approach. The concept of "Open Systems Dependability" was proposed, and will be described in detail in this paper. We redefined the goals of our project as follows.

> To develop a methodology and methods for dependable systems software, which enable satisfactory services to be provided to users continuously by adapting the systems to accommodate ever-changing objectives and environments of the systems. "Open Systems Dependability (OSD)" is the base concept of the methodology, and the methods consist of an iterative process to build and operate dependable systems (the DEOS Process), an architecture to enable the process (DEOS Architecture), and technologies to implement the architecture.

The concept of Open Systems Dependability (OSD) and the DEOS process should be applicable not only to building and operating dependable systems software but also as a method that builds and operates large and complicated systems that can accommodate change throughout their lifespan. That is, the DEOS process is applicable to open systems in general. In this context, we use DEOS as the abbreviation for Dependability Engineering for Open Systems. On the other hand, the DEOS architecture described in this whitepaper is for huge and complex software systems including modern embedded systems. In referring to this architecture, we use DEOS as the abbreviation of Dependable Embedded Operating Systems.

The planned deliverables of the project are;

- Concept of Open Systems Dependability (OSD Concept)
  - DEOS Process
  - DEOS Architecture
- Requirements Management Process
  - Process Specifications
    - Requirements Elicitation / Risk Analysis
    - Stakeholders' Agreement
  - Process Support Tools
    - Agreement Support Tools (D-Case Editor, D-Case Viewer)

    - Agreement Description Language (D-Case, D-Case Patterns)
    - Execution Script Language (D-Script)
- DEOS Runtime Environment (D-RE)
    - D-RE Specifications
    - D-RE Reference Code
- DEOS Development Support Tools (D-DST)
    - DS-Bench/Test-Env
    - Model/Type Checker
- DEOS Elemental Technologies (Specifications of each technology, API definition documents, Software, Implementation guidelines)
- Standards, Guidelines
- Establishment of DEOS Consortium

2. Dependability

2.1. Brief Historical Review

In the 1960's, the construction of a Fault Tolerant Computer was proposed to support real-time computing and mission critical applications. Active discussion of this topic has been ongoing since then [15, 19]. As a result of this discussion along with the increase in scale of hardware and software and the spread of online services, a concept called RAS has been developed which integrates resistance to failures (Reliability), maintenance of a high operating ratio (Availability), and quick restoration during a malfunction (Serviceability or Maintainability), with an emphasis on error detection and system recovery [8, 14]. In the latter half of the 1970's, to this concept was added the preservation of data consistency (Integrity) and the prevention of unauthorized access to confidential materials (Security), for RASIS, an extension of RAS that has served as a standard for evaluation. In 2000, the idea of Autonomic Computing was proposed to ensure dependability in complex systems connected by networks with autonomic action, in the same way that the autonomic nervous system works in the human body [9, 10, 11, and 16]. A more exhaustive review is found in [51].

There has been progress in methods of software development, which needed to realize the dependability of information systems. Software development methods such as Structured Programming (Dijkstra [29]) and Object Oriented Programming (SIMULA [30], Smalltalk [54]) were developed, and then Project Management Methodologies for software development were introduced, which improved software development process management (CMM [31, 32], CMMI [33]). Projects dealing with development methods for complicated and large scale systems started (System of Systems, Ultra-Large-Scale Systems [34]). IT governance and service management frameworks are studied in CoBIT and ITIL.

Changes in approaches taken to ensure reliability are reflected in international standards. The international standards of IEC 60300 series are known as standards for dependability management. Those series of standards were established by IEC TC56. This originally was a technical committee dealing with reliability of electronic components, but IEC 60300-1 (2003 edition), which is a core standard of the IEC 60300 series, does not fully include what is required for today's software. The work is in progress for the next edition, which will set standards for extended areas of products, systems, services, and processes to be included as targets of dependability management. International safety standards ISO 13849-1 (EN954-1) and IEC 60204-1 can handle simple systems, subsystems, and parts, but are not sufficient to deal with systems that include software. Functionality safety standard IEC 61508 was established in 2000 out of necessity for a safety standard for systems that include software. In IEC 61508, a system malfunction is divided into "random hardware failure" and "systematic failure". The probability of random hardware failure is calculated by monitoring malfunctions due to the deterioration of parts; while systematic failures, caused by incorrect system design, development, production, maintenance, and operation, are to be kept from exceeding allowed target values through a verification process such as the V-model and the documentation of all operations based on the safety lifecycle. Systems are categorized according to mode of operation: low demand mode or high demand/continuous mode. The target failure limit for each mode is defined and managed as the Safety Integrity Level (SIL). The requirements of 4 stages from SIL1 to SIL4 are also defined (with SIL4 requiring the highest safety integrity). With IEC 61508 as the base standard, machinery-related IEC 62061, process-related IEC 61511, nuclear-related IEC 61513, railway-related IEC 62278, etc. were established. For automotive systems, a DIS (Draft International Standard) of ISO 26262 was issued in June 2009 and the final version is expected to be issued in 2011.

Efforts are continuing to produce a single definition of dependability, which integrates different conceptions. In 1980, a joint committee of IFIP WG10.4 studying Dependable Computing and Fault Tolerance and IEEE TC studying Fault Tolerant Computing was formed, and they initiated a study on "The Fundamental Concepts and Terminologies of Dependability". The details and results of the subsequent investigation were compiled in a technical paper that was published in 2004 [2, 3].

Despite the progress of research on dependability and the development of related technologies, failures of large scale software systems keep occurring frequently. Some examples are listed in Appendix A.4. The major causes of the failures are: the system which developed without sufficient understanding of the behavior of all of the components, the usage of the system being exceeded the initial design limit, and the inconsistency of the system caused by system change to accommodate users' requirements changes while the system was in operation. There are cases where careless design or operations by programmers or operators caused a chain of failures to lead to very critical system failures.

The concept of dependability has changed significantly to meet the needs of the times. The concepts in the past could not cover all of the aspects we have discussed. It is not appropriate to assume that the functions, structure, and boundaries of the systems of today are fixed and definable; but rather. It must be assumed at the very beginning of system design that the above change over time. In the following paragraphs, the characteristics of today's systems and the major causes of failures will be clarified, and a new concept of dependability for today's systems will be proposed.

2.2. Characteristics of Modern Systems and Causes of Failures

Today's large software systems have become much more sophisticated and complex in order to meet the various needs of users. To shorten the development period and to lower development costs, the practice of using "black box" software, such as existing software or software provided by other companies, has increased. Moreover, specification updates and change for function improvement occur while the system is in operation. In some cases, amendments of the software are downloaded and new functions are added through the network without discontinuation of services of the systems. It is becoming exceedingly difficult for designers and developers to know each and every detail of the system throughout its lifecycle (Fig. 1).

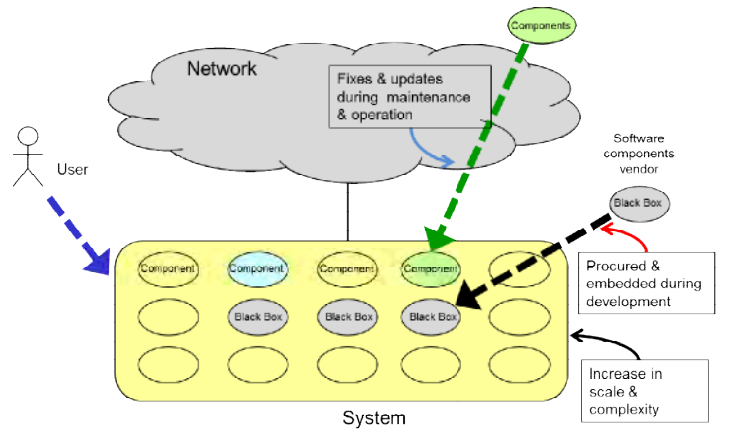

Fig.1. Configuration of System and Services

Many of the modern software systems provide services together with other systems to which they are connected via a network. In this case, users of the systems directly utilize services provided by a single domain, which may indirectly utilize services provided by other service domains through the network. In many cases those service domains are owned and operated by different owners. There is possibility that the items or contents of services, transaction performance, and interfaces may change without appropriate advance notice, that unknown services may be added, or that services used in other domains may be terminated. In addition to the change of services, changes of network itself such as service items or contents of the network, transaction performance, or interface specifications of the network itself may change. A network may be suspended temporarily. The boundaries of a system or the service domain are becoming

increasingly unclear to users. Likewise, there exists the concern that the system may be attacked on purpose with malicious intent. For these reasons, the advent of networking has made predictability much more difficult to attain (Fig. 2).

The analysis and classification of the causes of system failures discussed above, considering systems and services from both the development and operation standpoints, must address the following key factors which are inherent characteristics of the development of open systems.

(1) Incompleteness

It often happens that the initial requirement specifications become inadequate, mismatch exists between specifications and implementation, or the system's behavior is difficult to fully understand, at the time of shipment or service-in. Specifically, there is likely to be:

- An error or omission in specifications, design implementation, or testing, caused by the difficulty of understanding the whole system, particularly its software, due to its complexity and size
- An error or omission in the specifications, design implementation, or testing, caused by discrepancies in characterizations of the system during the requirements phase, specification phase, design phase, implementation phase, or testing phase, or by an error in the documentation
- An error in updating procedures in administration, operation, or maintenance, or an error caused by expiration of license
- Inconsistency between external specifications and actual operation of software components such as "black box" components or legacy codes

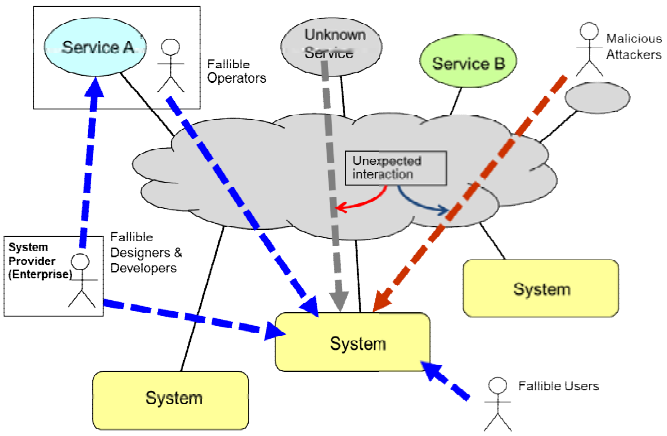

Fig.2. Services through Network with Human Interactions

(2) Uncertainty

Changes of users' requirements or the usage environments throughout the lifecycle of the system make it difficult to completely predict the behavior of the system in the design or operation phases. Examples of such changes include:

- Changes in system requirements caused by the changes of business goals
- Changes in user's requirements or expectations for the systems, or changes of operators' skill or capability during the maintenance and operation phases
- Unexpected usage changes, such as those brought about by the significant increase in users or units, and by changes in the economy
- Update or alteration of a component's function or of the system configuration while the system is in operation by direct manual operation or through a network
- Unexpected network connections and increase of interactions; intentional malicious attacks and intrusions from external entities

Today's large systems must deal with inherent incompleteness and uncertainty. We cannot create a flawless system that can already handle all possible scenarios that could take place in the future.

Contemplation of this situation has led people to make various definitions of "dependability". The examples of such definitions are: "The continuing state where no failures or malfunctions occur, or where the situation is grasped immediately when abnormalities do occur, the subsequent situation is predicted and social panic and catastrophic breakdown is prevented, at reasonable

cost." [7], and "The capacity for the services offered by the system to be maintained at a level acceptable to the users even if various accidents occur." [17]

Even if we are unable to avoid system failures completely, we think we can develop methods or technologies to continue the business by minimizing the occurrence of fatal failures, minimizing damage caused by the failure, preventing similar failure from happening again, and achieving accountability. We set our objectives to be the achievement of the above, by redefining dependability as described in the next section, and developing the methods and technologies.

2.3. Open Systems Dependability

As we have discussed so far, we need to deal with the systems which inherently have incompleteness in specifications and implementation and uncertainty caused by ever-changing users' requirements or systems environments. Systems with these characteristics are called open systems. The differences between open systems and closed systems are listed below to help in understanding the characteristics of open systems (Fig. 3).

The characteristics of closed systems are;

- The boundary of the system is definable.
- The interaction with the outer world is limited, and the system functions are fixed.
- The subsystems or components of the system are fixed and their relationship does not change over time.
- The system is observable from outside of the system.
- Reductionism is applicable.

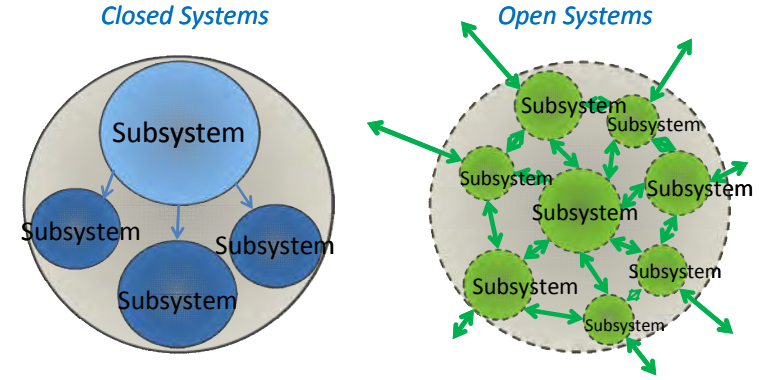

Fig.3. Closed Systems and Open Systems

On the other hand, the characteristics of open systems are;

- The boundaries of the systems change over time.
- There is interaction with the outer world, and the system functions change over time.
- The subsystems or components of the system and their relationship change over time.
- An observer of a system is inherently a part of the system, and the system is not observable from outside of the system.
- Therefore, reductionism is not applicable.

The computer systems we need to manage today have the characteristics of open systems; that is functions, structures, and boundaries keep changing over time. Because it keeps changing, it is reasonable to conclude that reductionism is not applicable, and that observers of a system, such as owners, designers, developers, operators, and users are inherently parts of the system through their interaction with the system.

It may be possible to assume a system to be a closed system at a specific time, which means assuming that there will be no change for a certain period of time, and then consider the lifecycle of the system by concatenating these periods of time. In this case, the function, the structure, boundary, and specifications of the system need to be defined at each time, and the design, verification, and the testing of the system are done based on these specifications, repeating this process for each period of time in the lifecycle. The traditional dependable systems have been developed and maintained in this way. However, it is extremely difficult to separate the phases where the system is fixed and in operation, and the phases where the system is in the process of modification.

Usually it is most important that the service and operation of the system is continued, even with changes to the system that are on-going. To deal with this situation, we should focus on "ever-changing systems" and establish the concept of dependability focusing on realizing the continuity of services and business by managing the ever-changing systems properly. Our approach is to consider the systems as open systems and to focus on how we should improve dependability throughout their lifecycles. Based on our discussion thus far on the characteristics of modern software systems, we define **Open Systems Dependability** (OSD) with the following description:

> **Functions, structures, and boundaries of modern software systems change over time. Hence incompleteness and uncertainty may result in failures in the future (factors in open systems failure). Open Systems Dependability is a property of a system such that it has the ability to continuously prevent these problem factors from causing failure, to take appropriate and quick action when failures occur, to minimize damage, to safely and continuously provide the services expected by users as much as possible, and to maintain accountability for the system operations and processes.**

"Open Systems Dependability" does not conflict with the "dependability" that has been studied, discussed and classified by many researchers. Until now, technologies for improving the safety and security of systems have been researched, discussed and developed with a focus on incidental and intentional faults. Our approach is to improve the dependability of systems by minimizing the factors that specifically cause open systems failures and minimizing the damage due to open systems failures, concentrating on open systems failures resulting from incompleteness and uncertainty. Indeed, "Open Systems Dependability" further enhances traditional dependability.

3. Achieving Open Systems Dependability

As described in the previous chapter, Open Systems Dependability is dependability of ever-changing systems. In order to achieve Open Systems Dependability, we consider that an iterative process approach is indispensable. We also consider that such a process must incorporate two cycles: one to adapt the system according to requirement changes caused by changes in objectives and environment, and the other to take immediate actions and fix failures when failures occur while the system is in operation. In addition, each cycle is composed of component processes and states, such as a requirements management process, development process, ordinary operation state, failure response process, and accountability achievement process, which are organically united. Thus, it is a process of processes. We propose that the DEOS process have such a composite makeup.

In order to perform the DEOS process, an architecture that effectively supports the process is essential. We consider that such an architecture should include 1) tools to support a requirements management process, 2) a database that preserves and maintains agreement descriptions, 3) software development support tools for such functions as program verification, benchmarking, and fault injection testing, and 4) a program execution environment that provides a function that dynamically responds to failures so as to minimize damage, by monitoring, recording, and reporting the state of the system. We propose that the DEOS architecture to include the above.

3.1. DEOS Process

The concerned parties in a system are called the stakeholders. We consider the following parties to be possible stakeholders of a system:

- Users of services or products (the whole society in the case of systems for social infrastructure),
- Providers of services or products,
- Providers of systems:
  - Designers and developers,
  - Maintainers,
  - Providers of hardware, and
- Certifiers (Authorizers) of services or products.

The stakeholders may change their intentions as time passes and/or according to environmental changes, and therefore may change their requirements for functions and services of the system. We call such a change “objective/environment change”. After the stakeholders have deliberated and mutually agreed upon an objective/environment change, they request a system change at an appropriate time. This request initiates the development process, and a new program will be put in operation. The DEOS process provides a cycle to accommodate such changes, called a “change accommodation cycle”

A system cannot fully get rid of its failures due to its incompleteness and uncertainty. Therefore, when we predict a failure before it has occurred, we need to prevent it from occurring, but it may not be preventable in some cases. In the case where a failure has unfortunately occurred, immediate actions to minimize the damage, and to analyze causes must be taken. In addition, it is crucial that there is accountability for the failure. The DEOS process provides a cycle for this, and is called the “failure response cycle.”

When newly developing a system or revising a system in accordance with changes in objectives and environment, we need to record the reasons for actions, the process of discussion among the stakeholders, and the resulting agreements. This is essential to continuously improve the system and to achieve accountability. For this purpose, a database to preserve and maintain

such descriptions, called the agreement database, is provided in the DEOS process. This database contains D-Case that describes the process of discussions and agreements for achieving dependability and D-Script which describes executable procedures to detect signs of failures, prevent failures, and respond to failures in case they occur to minimize damages. Based on these descriptions, the development process, failure response process, and ordinary operating process are executed and accountability is achieved. Thus, the agreement database plays an important role in organically uniting the component processes.

In summary, the DEOS process has the following characteristics (Fig. 4):

(1) It consists of two cycles, the change accommodation cycle and the failure response cycle, both initiated from the ordinary operation state.
(2) It has two important phases, the stakeholders' agreement phase for system requirement changes and the accountability achievement phase for system revisions and for failures.
(3) It provides the agreement description database containing D-Case which describes reasons for actions, progress of discussions, and resulting agreements, and D-Script which describes executable procedures to respond quickly to failures.

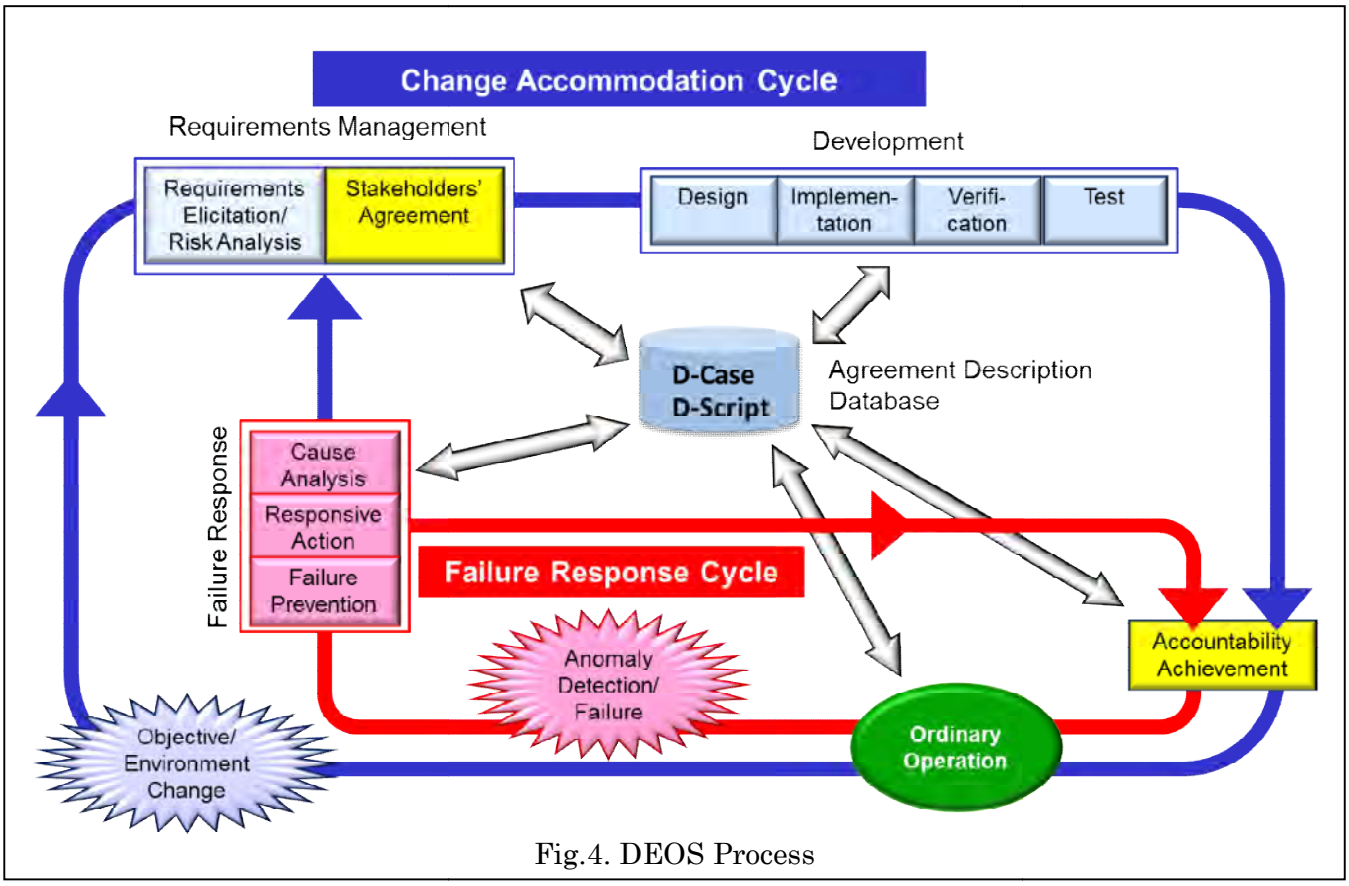

Fig.4. DEOS Process

3.1.1. Ordinary Operation

Ordinary operation is the state where the system is operating with values within the ranges agreed upon by the stakeholders. We refer to these ranges as "in-operation ranges". The change accommodation cycle should run in parallel with ordinary operation, so that improvement of the system is performed while providing services. Also, the failure response cycle preferably runs in parallel with ordinary operation. In case the system detects a sign of a failure before it occurs, it might be able to prevent it from occurring. Even in the case where a failure has occurred, the system may run in a degraded mode. There also are cases where the system must be fully stopped after a severe failure.

In ordinary operation, the following actions are taken:

- recording the system's operation states and inspecting this record periodically to detect failure signs,
- cleaning systems to prevent system aging,
- rehearsing failures to improve detectability of failure signs and preventive actions,
- reviewing processes periodically to improve them,
- educating and training staff,

etc.

3.1.2. Failure Response Cycle

The failure response cycle is the cycle to quickly respond to failures to minimize damage. In the DEOS process, failures are defined as the deviation of services and functions from the acceptable operation ranges that have been agreed upon among the stakeholders. The failure response cycle starts from the ordinary operation state, and consists of the failure prevention phase, the responsive action phase, the cause analysis phase, and the accountability achievement phase. The first three phases are not always executed in series in this order, and often are executed collaboratively in parallel.

The failure prevention phase is a phase for the following: when a failure is predicted before it occurs, or the possibility of a failure increases, an action is taken to prevent such a failure from occurring. If such situations have been detected well in advance of the time when the failure is expected to occur, an effective preventive action can be taken, for example temporarily rejecting particular service requests or lowering the throughput. If such situations have been detected just before the failure is to occur, effort will be made to minimize damage. In any case, the system tries its best to collect as much log data as possible so that causes of the failure can be analyzed quickly and easily. A practical approach for failure prevention is to detect anomaly patterns that have caused a failure in the past. When such an anomaly pattern is detected and values go beyond their in-operation ranges, this cycle is initiated. Failure prevention scenarios, which are described in D-Script are executed automatically or in cooperation with operators and system administrators.

The responsive action phase is the phase for minimizing damage when a failure has occurred. When a failure, defined as a deviation of a value from its in-operation range, is detected, the system enters this phase, and a responsive action scenario, which is described in D-Script, is initiated. This process is preferably performed automatically, but there are cases where this is impossible. For such cases, the responsive action scenarios should include descriptions of how to designate responsible persons or groups and rules concerning escalation. Automatically or with the help of designated persons or groups, the system isolates the failure, aborts and restarts some programs, and then goes back to the ordinary operation.

The cause analysis phase is the phase for analyzing causes of failures. It analyzes which responses can be achieved as a quick recovery, and the root causes of the failure in preparation for a long term solution. For both purposes, accurate and appropriate records of system behavior should be preserved as log data. Analyzing root causes for long term solution will often been done off-line by human intervention and the results of this analysis initiate the requirements management process in the change accommodation cycle.

In the accountability achievement phase, the service and/or product providers disclose and explain to the users and other stakeholders the nature and causes of the failures, current status of the failures, the actions taken, expected recovery time, and plans to amend their development and operation processes and procedures in ordinary operations so as to prevent such failures from occurring again in the future. The agreement description database, especially the D-Case

description, used together with log data and various tools, helps the providers to achieve accountability.

3.1.3. Change Accommodation Cycle

The change accommodation cycle is the cycle adapts the system to new requirements caused by changes in objectives and environments. It starts from the ordinary operation state, and consists of the requirements elicitation/risk analysis phase, the stakeholders' agreement phase, the development process, and the accountability achievement phase. This cycle is initiated by changes in objectives and environments as described above and also as a result of cause analysis in the failure response cycle, calling for a significant system change.

In the requirements elicitation/risk analysis phase, requirements are elicited from service objectives of service/product providers, users' needs, other stakeholders' intentions, various system environments, related regulations and standards, and so forth. At the same time service continuity scenarios are created. The result of this phase is a set of dependability requirements.

The stakeholders' agreement phase is the phase where stakeholders discuss what to changes to make in the system and why, and how to make these changes. The process of discussions and the resulting agreement are described with supporting reasons in D-Case. This phase also generates executable procedures from the service continuity scenarios. Such executable procedures are described in D-Script to respond quickly to failures.

The development process consists of design, implementation, verification, and test phases. There are various excellent tools available for program development that we can use for the DEOS process. In order to enhance the dependability of programs, we consider that program verification, benchmarking, and fault injection testing are important. We are developing tools for these.

In the accountability achievement phase in the change accommodation cycle, the service/product providers need to explain to the users and other stakeholders why the system should be revised and how the services and functions will be changed. For this purpose, the D-Case description helps the providers to achieve accountability.

3.2. DEOS Architecture

The DEOS process provides an iterative process to achieve dependability of open systems in general. In order to apply the DEOS process to systems for a specific field or application, we can construct an architecture that can effectively support the DEOS process for that field or application. In this section, we describe a DEOS architecture for huge and complex software systems including modern embedded systems.

This DEOS architecture consists of the following components (Fig 5):

- A set of tools to support the requirements elicitation/risk analysis phase,
- D-Case Editor and D-Case Viewer, tools to support the stakeholders' agreement phase,
- Agreement Description Database (ADD) which contains D-Case and D-Script,
- The DEOS Runtime Environment (D-RE), and
- A set of tools for program verification, benchmarking, and fault injection test (DEOS Development Support Tools, D-DST).

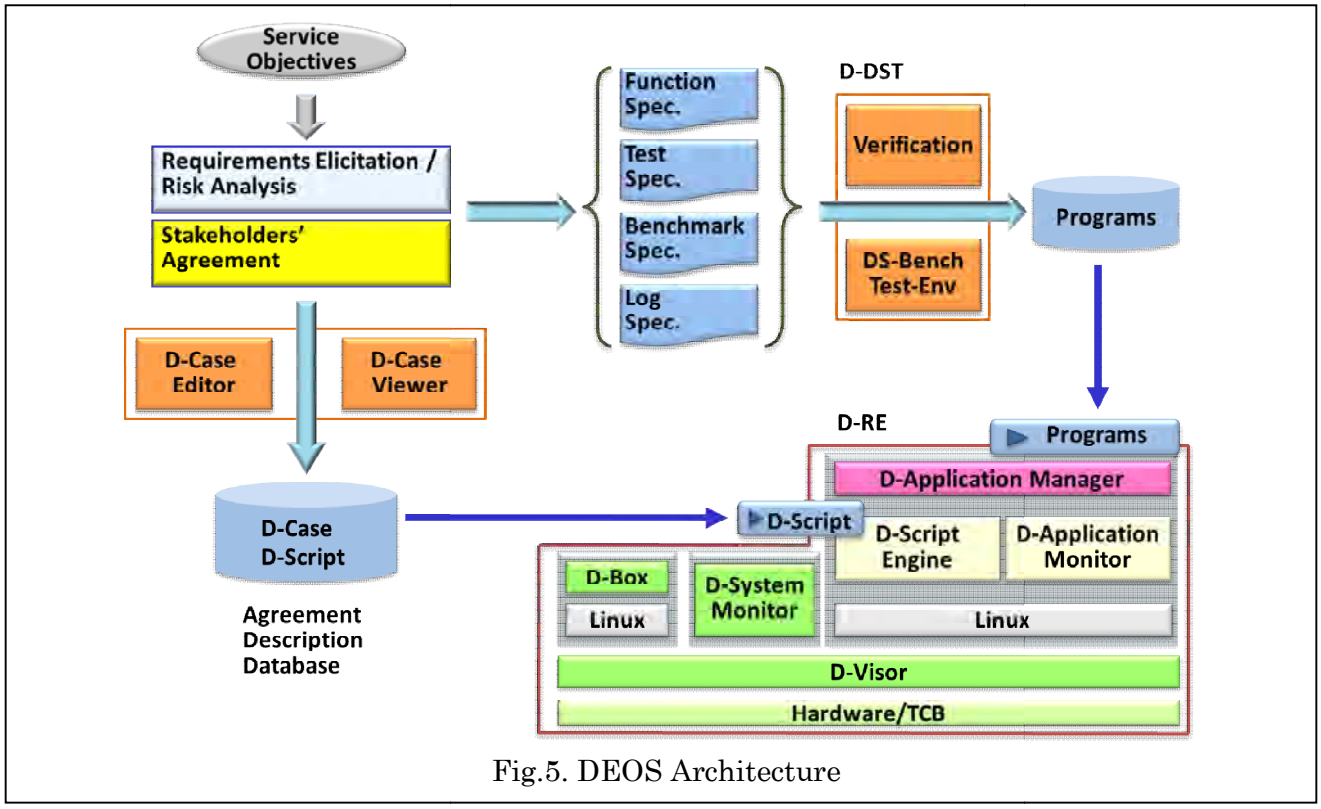

Fig.5. DEOS Architecture

The tools to support the requirements elicitation/risk analysis phase facilitate elicitation of requirements, generation of service continuation scenarios, and analysis of risks. Such a set of tools is being designed and implemented.

D-Case Editor facilitates description of stakeholders' agreement following a specific description method and notation. D-Case Viewer helps stakeholders to retrieve and examine related agreements. A tool to support generation of D-Case descriptions is under development. These tools are also used to help service/product providers achieving accountability.

The D-Script plays a very important role in the DEOS architecture. It dynamically combines the D-Case description and the execution of application programs. That is, it contains executable scenarios which instruct the D-Script Engine when and what log data should be collected and how D-RE behaves in case of failures. Such scenarios may call for human intervention in case of severe failures. This flexibility contributes to achieve Open Systems Dependability.

D-RE is an execution environment with functions that help to achieve dependability in a system. It consists of the following subsystems: 1) D-Visor has a function which abstracts the hardware and isolates the functions of the subsystems. This function creates what are called system containers, and prevents a failure, which has occurred in one system container from propagating to the other system containers. 2) D-Application Manager has a function, which isolates application programs. This function creates what are called application containers; this insures the independence of each application program and manages and controls the lifecycle of application programs. 3) D-Application Monitor monitors application programs in the application containers, collects log data according to the descriptions in D-Case monitor nodes, and stores this in the D-Box. 4) The D-Box preserves log data even in the case of a system crash, functioning like a flight recorder. 5) The D-System Monitor monitors the system, collects log data according to the descriptions in D-Case monitor nodes, and stores this in the D-Box. 6) The D-Script Engine executes D-Script safely, and controls the D-Application Manager, D-Application Monitor, and D-System Monitor.

DEOS Development Support Tools (D-DST) we are developing include a program verification tool, which consists of two parts, one using a type-theoretic approach and the other adopting

model-based approaches. We also are developing a dependability test support tool that consists of benchmarking tool and test environment with fault injection function that can utilize a cloud environment.

3.3. Expected Benefits of DEOS

The largest benefit which we will enjoy by using the DEOS process is that it will enable enough discussion among the stakeholders to reach agreement on the requirements changes and then enable description of reasons, progress of discussions, and resulting agreements in D-Case. The D-Case description will be referred to in designing the system with the help of the DEOS architecture so that the system can achieve proper and quick response in case of a failure. Causes of the failure will be analyzed and accountability will be achieved, both based on the D-Case descriptions.

Another benefit derived by using the DEOS process will be that the whole process is managed based on the requirements which are elicited. Each stakeholder can know the state of the system at any time from its own viewpoint. This enables simple but effective management of the system during operation. On the other hand, the number of requirements will become huge. Necessary tools including D-Case Editor and D-Case Viewer are provided to reduce tedious work in requirements management.

The DEOS architecture provides D-RE which records and reports all the necessary states of execution of the system and application programs (based on the D-case description). This enables quick response in case of a failure, using D-Script. This also gives supporting evidence to stakeholders for analyzing causes and for achieving accountability in accordance with the D-Case description. D-Script together with D-Script Engine provides the function of interfacing the D-Case and D-RE. This enables flexible response, automatically or with human assistance if necessary (again, in accordance with the D-Case description).

The DEOS architecture provides the function of isolating a failure so that it does not propagate to other programs. It also provides the function of intrusion detection to maintain the security of the system. Various tools provided will also help achieve dependability at the development phase; to verify programs to be executed, to check performance, to examine the behavior of a system against injected faults, and so forth.

By exploiting the facilities and functions described above, the DEOS process and architecture support the provision of the ability to continuously prevent failures from occurring, to take appropriate and quick actions when failures do occur to minimize damage, to provide continuous service, and to maintain accountability. The DEOS process and architecture is the first attempt to achieve Open Systems Dependability.

4. Requirements Management in DEOS Process

Requirements are the most fundamental elements in the DEOS process. In this chapter, we explain the DEOS process used with requirements management. The requirements management process in the DEOS process consists of the following four phases: 1) requirements elicitation and risk analysis, 2) stakeholders' agreement on requirements, 3) verification of realized requirements and achieving accountability, and 4) changing requirements and updating agreement. In the course of consensus building, elicited requirements may possibly be revised. Fig. 6 shows that each agreed-upon requirement is implemented in two forms: in the form of programs or of D-Script. The runtime environment D-RE offers an infrastructure for accountability achievement by logging execution of D-Script.

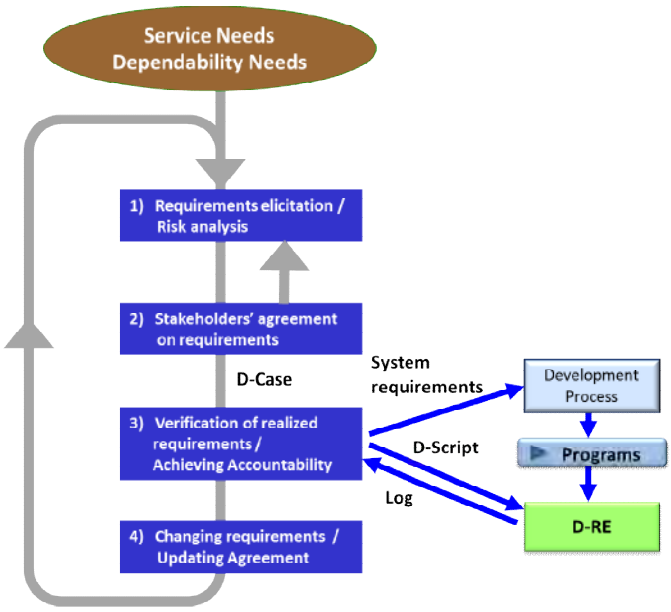

Fig.6. Requirements Management Process

4.1. Requirements Elicitation and Risk Analysis

The requirements elicitation starts with the service objectives. Stakeholders can be defined according to their service objectives. Requirements are generated from each stakeholder's objectives and needs. Here, requirements include service requirements and dependability requirements. Regulations made by regulatory agencies can be considered as a kind of requirements. The activities for requirements elicitation must include identification of various levels of requirements in order for this task to be manageable.

In requirements engineering, various requirements elicitation methods have been proposed: Ethno-Methodology, Trolling, Business Modeling, Goal Oriented Analysis, Use Case Analysis, Misuse Case Analysis, Triage, etc. [52].

The DEOS process focuses on dependability requirements in its requirements elicitation and requirements analysis. First, needs are extracted from stakeholders who describe them informally and verbally, and from these, dependability needs are obtained. Second, "dependability requirements" are identified through the analysis of dependability needs. Next, "service continuity scenarios" are created based on risk analysis and service requirements. More precisely, service continuity scenarios are developed by considering and determining countermeasures for each factor causing deviations. Finally, D-Case and D-Script documents are created through consensus building among stakeholders based on the service continuity scenarios.

The requirements management in the DEOS project involves management of change in requirements to satisfy the following three conditions:
(1) Dependability despite change in requirements can be assured (See 4.2).
(2) Agreement can be made successively regarding each changed requirement (See 4.2).
(3) Accountability can be achieved for each changed requirement (See 4.3).

4.2. Consensus Building

Consensus building is a series of activities for achieving agreement on the requirements elicited from stakeholders. D-Case is used for consensus building. The characteristics of D-Case are:

(a) Structural notation for agreement based on arguments and evidence
(b) Support for managing stakeholders and their agreements
(c) Support for monitoring the achievement of an agreement in ordinary operation (monitor node)
(d) Support for verifying consistency of agreement descriptions

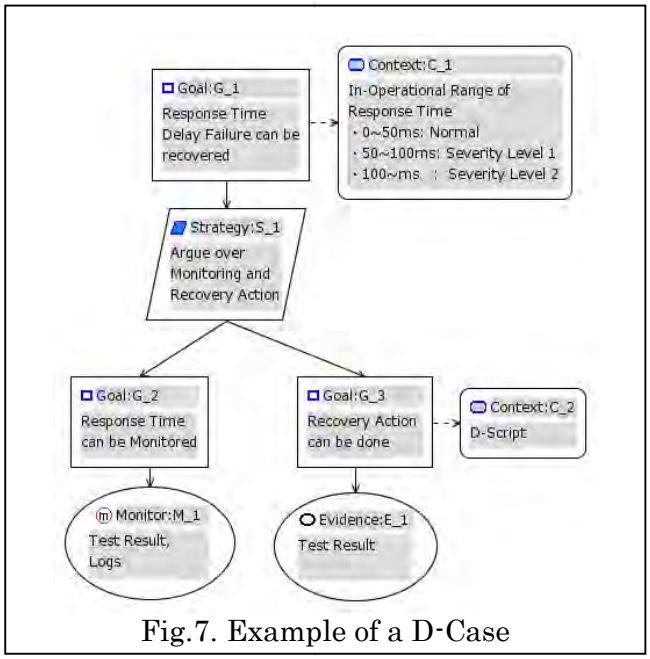

Fig.7. Example of a D-Case

This structural notation is based on a goal-structuring notation (GSN) [37, 38] (a). For safety critical systems, GSN already is widely used in "safety cases" for describing how safety is assured in various cases [39, 40, 41, and 42]. In addition to GSN, D-Case supports a node type that represents the relationship between stakeholders and their agreements (b). We are developing a D-Case tool to visualize consensus building.

Also, to support achievement of accountability in operation of a system, we introduce monitoring nodes. These nodes designate system parameters to be monitored at run time by D-System Monitor or D-Application Monitor.

Since it is time consuming to develop agreement documents, we will provide D-Case patterns for various application domains. Since it is also a cumbersome work to check the consistency of D-Case agreements, we are providing a proof assistant tool as a backend interface (d).

The main part of D-Cases is structured with a few types of nodes including goal nodes for claims to be argued, strategy nodes for reasoning steps that decompose a goal into sub goals, evidence nodes for references to evidence that support the goals to which they are linked, context nodes for environmental information, and monitoring nodes for runtime evidence. Fig. 7 shows an example of D-Case, which argues about response time delay failures of servers. The in-operation range of response time is defined in Context C_1 with severity levels 1 and 2. This example shows how the failures can be monitored and recovered by the D-Script according to the monitored severity level. The argument is supported by test results and logs of monitoring.

4.3. Requirements Implementation Confirmation and Accountability Achievement

Requirements need substantial agreement among stakeholders to be established. However, agreed-upon requirements in D-Case are not necessarily implemented in the real system and maintained in ordinary operation. The achievement of agreed requirements is done by proper implementation and maintenance of agreed-upon requirements. This must be objective and confirmable by stakeholders or even third parties when external judgments are needed.

Agreed-upon requirements should be tested and verified in the development process until the implemented system reaches the agreed-upon level. When deviation from stakeholders' agreements occurs in ordinary operation, recovery efforts including D-Script execution and human intervention must be performed to maintain the agreements. The content of such recovery efforts should have been agreed upon in D-Case.

Every record of actions for achieving the agreements is essential for achieving accountability. Whenever an agreement is not implemented, all the information including supporting evidence is disclosed to stakeholders. The DEOS process is designed to allow automated collection of necessary evidence to fulfill such accountability requirements.

4.4. Requirements Change and Agreement Update

In requirements management, as mentioned above, states of requirements are managed. There are four kinds of the states; elicited, agreed, ordinarily operated, and deviated (Fig. 8). First, requirements are elicited from stakeholders. These elicited requirements may conflict. By consensus-building, requirements are agreed upon among the stakeholders. Agreed-upon requirements are then implemented in ordinary operations. When objectives and environments change, some ordinarily operated requirements may become obsolete and new requirements must be elicited again. This is referred to as the change accommodation cycle.

If a requirement is not fulfilled, i.e., there is deviation from the corresponding in-operation range, it moves to the deviated state. When a responsive action is possible, it moves back to the ordinarily operated state. This is referred to as the failure reaction cycle. If the service continuity scenarios cannot work for some requirements in the deviated state, these requirements should be modified and move to the elicited state. If deviations came from the implementation problems, the corresponding elicited requirements do not need any change. But it is necessary to agree on other requirements to revise the faulty implementation. This is done by consensus-building.

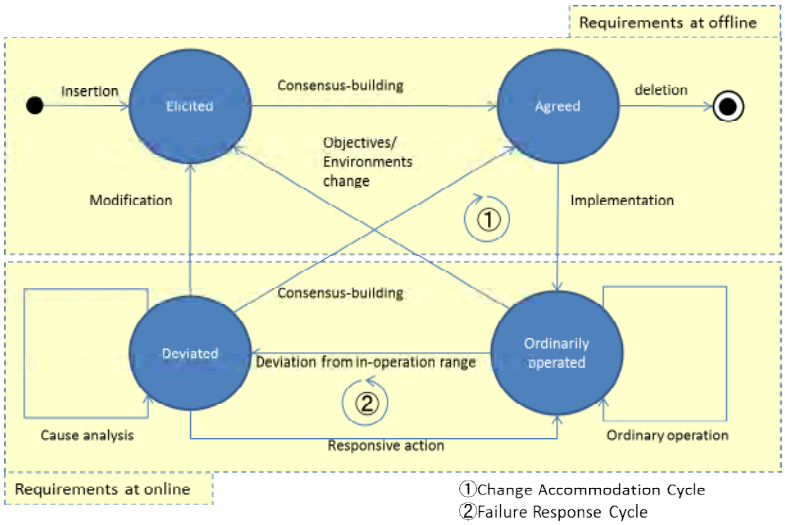

Fig.8. Requirements State Management Model

The elicited and agreed states of requirements are managed at offline, whereas ordinarily operated and deviated states are managed online. The state of the system is represented by a set of these requirements states.

4.5. Towards Evaluation of Conformance to DEOS Process

After a target system is developed using the DEOS process and a DEOS architecture for an area or application, conformance of the system to the DEOS process must be evaluated. We are developing conformance evaluation criteria. The evaluation is performed using the D-Case description which was generated during the development of the target system. Fig. 9 shows an example. Achievement of dependability is evaluated in three state and cycles: in the ordinary operation, in the failure response cycle, and in the change accommodation cycle as shown in the leftmost column. For each of these, achievement of dependability of the system is evaluated according to three life stages: the planning stage, the implementation stage, and the execution stage.

For example, let us consider the intersection of the failure response cycle and the planning stage in Fig. 9. In the D-Case description, there are sub-goals in the failure response cycle reflecting a service level agreement (SLA), which is widely used in development contracts of web services applications. Examples of criteria which this section includes are (1) whether it can be judged properly and quickly, and (2) whether an escalation process with a rule for authority delegation when the SLA fails is well-defined.

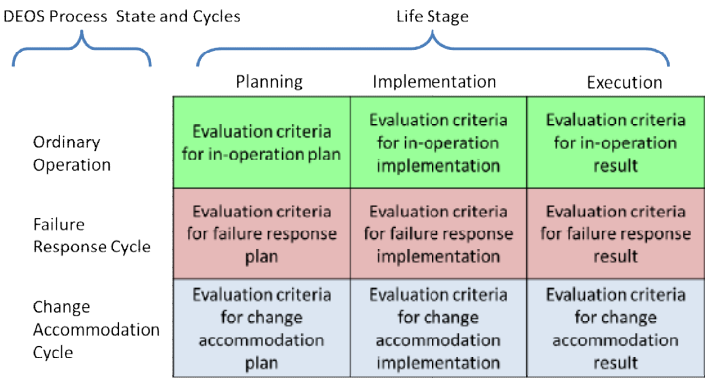

Fig.9. Conformance to DEOS Process

Detailed criteria are being developed, as shown in Appendix A.7. DEOS process conformance evaluation will further be developed in the process of standardization.

4.6. Support Tools for Stakeholders' Agreements

We have been developing "D-Case Editor" [43], which is a tool to support stakeholders' agreements (Fig. 10), and "D-Case Viewer", which is a tool to show how stakeholders' agreements are satisfied based on appropriate dependability metrics. D-Case Editor will co-work with DS-Bench/Test-Env, D-RE, D-Script, and Agda which is a proof assistant tool for testing consistency and completeness of D-Case. Furthermore, to facilitate the use of our tools, we have been trying to make a tool chain with other development support tools including Redmine and UML modeling tools.

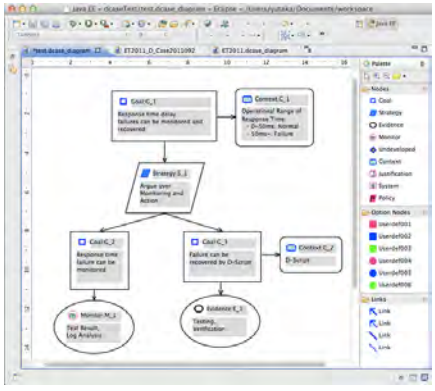
Fig.10. A Snapshot of D-Case Editor

For developing D-Case Editor, we are actively cooperating with other tool developers and standardization organizations, including Adelard LLP in UK, which provides a safety case tool called ASCE, OMG (Object Management Group) that is standardizing UML meta models, and Open Group, which is a standardization organization for enterprise architectures.

5. Runtime Environment for Monitoring and Flexible Control

The DEOS process has three important characteristics: 1) co-existence of the change accommodation cycle and the failure response cycle, 2) descriptions of dependability requirements agreed upon by stakeholders, written in D-Case, and 3) responding quickly to failures and detecting failure signs, procedures for both being governed by D-Case.

The stakeholders may change their dependability requirements for various reasons such as changes in environment or service objectives. Therefore, the D-Case representing the stakeholders' agreement at one particular time is transformed to another D-Case adapted to new stakeholders' requirements at another time. Below, five functions that should be provided by the runtime environment for implementation of the DEOS architecture are described.

Monitoring: This function monitors the activity inside the system, ascertaining whether it is operated by a procedure in conformance with the stakeholders' agreement. In order to achieve this function, data collected by the designated D-Case monitor nodes are used.

Reconfiguration: This is a function to configure the runtime environment according to D-Case. Changes in D-Case are reflected in the runtime environment, and there is a new configuration when there is a new D-Case. It is important in reconfiguration to isolate different system components. How to configure the runtime environment and how to isolate system components depend on how the systems are utilized. Therefore, it is required that the reconfiguration be flexible.

Script: This is a procedure of responsive action to be executed in response to failures, allowing flexible control. These actions include reconfiguration of the runtime environment according to D-Case change. It is also a procedure to isolate system components for reconfiguration. Such procedures are called "D-Script". The runtime environment is required to run the D-Script securely.

Recording: This is a function to record data designated by D-Case monitor nodes, a log of reconfigurations, and a log of D-Script execution. These data are called "evidence" in this article. It also records any information needed to sustain OSD.

Security: This is a function to insure the above functions to be executed securely. The runtime environment provides several security features such as access control, authentication/authorization, and system takeover protection, which are constructed from a trusted computing base.

The above functions are provided by several subsystems seen in Fig. 11, which outlines the D-RE described in this article. The next sections describe monitoring and recording, flexible system control, application lifecycle management, and security in virtual machines.

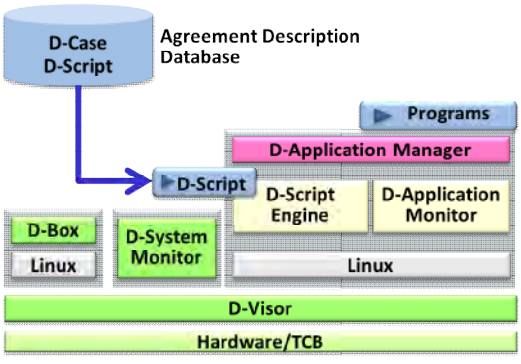

Fig.11. DEOS Runtime Environment

5.1. Monitoring of Agreement Achievements

Changes in a system occur for various reasons and may cause unexpected failures. Runtime monitoring (including online log analysis) is a key technology to grasp the operational state of a system. It is used to confirm whether the stakeholders' agreements are being achieved as described in D-Case.

The specifications that specify what parameters to monitor are described in D-Case along with their in-operation ranges and descriptions of the severity levels of deviation from those ranges. When deviation is detected, the failure response actions in D-Script are triggered.

Every D-RE monitoring starts with checking whether the parameters are in the in-operation range written in D-Case. Detection of deviation by monitoring initiates either the failure response cycle or the change accommodation cycle. The cycle is initiated by the description in D-Case. Log data is recorded in D-Box and is used to improve the system. If the detected deviation is of high severity, rapid response is required and therefore D-Script is invoked.

D-RE provides APIs for monitoring and logging at various levels of the system: they include networking interfaces, D-System Monitor, operating systems, D-Script Engine and D-Application Monitor. Log data are categorized into the following three types:

- Event log: records of events observed at a system boundary
- Sampling log: periodical records of resource use
- Trace log: records for tracing down dependency between two independently-recorded logs

5.2. Flexible Control of Systems

We cannot anticipate all the possible behaviors of an open system when it is being developed. The behavior of an open system may change in the course of its operation. Therefore, it is impossible to specify all the responsive actions to failures when it is being developed. D-Script is our means to achieve flexible control of responses to failures using the script language.

D-Script consists of D-Script scenarios. A set of D-Script scenarios is derived from the service continuity scenario described in BPMN (Business Process Management Notation) [53] or an equivalent method. A D-Script scenario consists of primitive actions called D-Tasks and control nodes called D-Controls. A D-Control designates sequential execution, conditional branching, and parallel execution. D-Scripts are described in a newly developed language Konoha [44]. Fig. 12 shows an example of a D-Script scenario with D-Tasks and D-Controls.

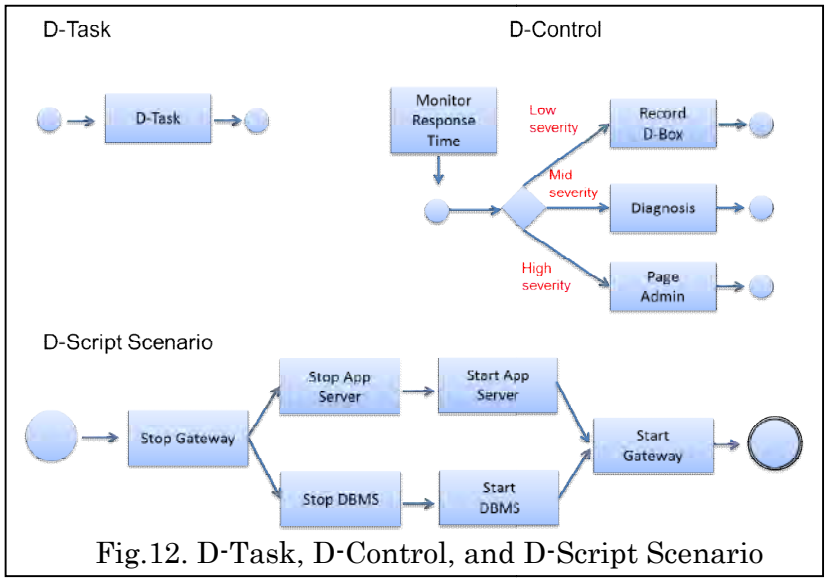

Fig.12. D-Task, D-Control, and D-Script Scenario

D-Script scenarios are modified when the system is changed. D-Script scenarios may have to be modified when unpredicted failures occur. This modification is performed by an authorized operator specified according to escalation rules described in D-Case.

D-Script runs on D-Script Engine, and designates D-System Monitor and D-Application Monitor to gather data on system's and application's states. When a value of such data deviates from the agreed-upon in-operation ranges, D-Script Engine invokes corresponding D-Script scenarios.

D-Script Engine has an integrated compiler that enables static type checking and security checking of given scripts, and executes compiled scripts in a reliable and secure manner, working with a TCB (Trusted Computing Base) or other hardware-supported security protection. In addition, the D-Script Engine provides logging as described in Section 5.1 including a persistent boundary event tracer, sample monitoring of resource usage, and D-Task monitoring. This helps track down the status of failure response actions without any additional logging code, and improve accountability achievement.

5.3. Application Lifecycle Management

Applications should be managed so that OSD is maintained through the DEOS process with the D-RE. All activities of applications are monitored by D-Application Monitor and managed by D-Application Manager. D-RE provides APIs for applications to enable those functions. The applications to utilize the APIs are called D-Aware Applications. In this Section, the states of application lifecycle and the role of D-Application Manager is described.

Create: This is the state of an application when it is registered to the D-Application Manager. D-Application Manager constructs the application's runtime environment by using information called the configuration information in the D-Case description.

Run: This is the state when an application starts its execution. In this state, the creation of a connection to D-Box to store data from D-Case monitor nodes, initialization of the snapshot function to store intermediate states of application execution, and initialization of a script engine to run a D-Script are carried out.

Update: This is the state occurring when the runtime environment is changed in accordance with a D-Case change. D-Script gives a description of how the execution environment is to be changed.

End: This is the state after D-Application Manager terminates the application. Any computing resource used by the application is released and the end of the application is recorded in D-Box.

D-Application Manager provides some basic services to monitor legacy applications (non D-Aware Applications) running in an application container.

5.4. Security Mechanism

To improve Open Systems Dependability, a security mechanism is mandatory to defend against various security threats. As security incidents are increasing, the security mechanisms of the operating systems, such as authentication, authorization, and access control, are becoming more sophisticated. Our goal is to provide a secure execution mechanism for operating systems; it guarantees the security mechanism of the operating system is working as it was designed.

Hijacking of operating systems is one of the most serious threats in computer security. This is because none of the security mechanisms can be trusted if the operating system, which plays the role of trusted computing base, is hijacked and functioning in an unexpected way. Our secure execution environment for operating systems defends against the security threats that attempt to hijack operating systems.

Since malicious attacks become more sophisticated over time, we need to evolve our security mechanisms to defend against them. In this particular project, we present a detection method to support the failure response cycle. A module employing our method can detect various attacks. Also, we provide some techniques to support the change accommodation cycle. They will help the security professionals to develop new countermeasures when unknown malicious attacks are discovered.

We make use of D-System Monitor and D-Visor to implement the above goals. D-System Monitor is clearly isolated from the target operating system due to the rigorous isolation provided by D-Visor. Thus, it is difficult for a hijacked operating system to compromise D-System Monitor. In D-System Monitor, we can safely monitor OS behavior and check whether the operating system is working as expected.

The key of our monitoring method is to check the response of the operating system to events. D-System Monitor can inspect various behavior of an operating system, coordinating with D-Visor. For example, the access to privileged registers, the execution of privileged instructions, and I/O operations can be inspected reliably because the operating system cannot feign this behavior. Furthermore, the underlying Virtual Machine Monitor (VMM), namely D-Visor, can inject hardware/software traps into the target operating system, and observe the behavior of the system in response to the injected traps. Based on this concept of trap injection and observation, it can be guaranteed that the target operating system will function as expected (Fig.13).

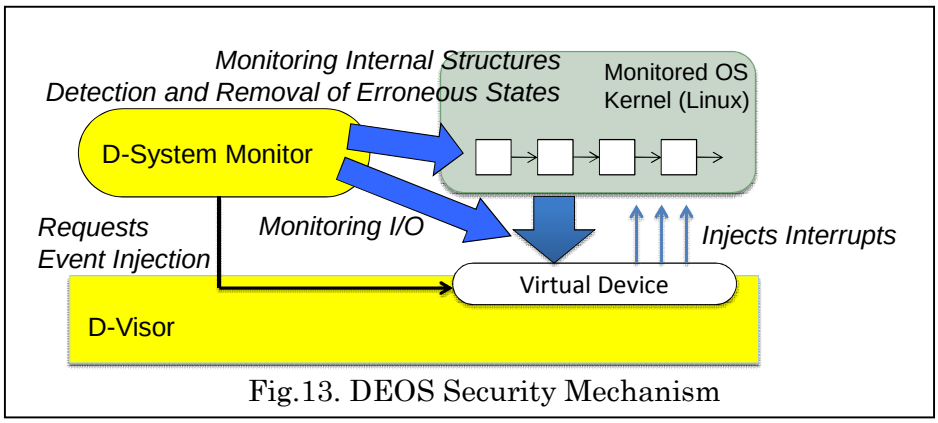

Fig.13. DEOS Security Mechanism

Our design has several advantages over traditional signature-based detection and prevention of malware infection. Traditionally, a signature must be developed for each malware sample. In contrast, we need to develop only a monitoring module for each class of malware. For example, to defend against the class of malware that tries to hide the existence of malicious files, we simply provide a monitoring module that matches the list of file names obtained from I/O operations with that obtained from the results of corresponding system calls; one module can detect all samples belonging to the same class of malware. To deal with a new class of malware, we have to develop a monitoring module for that class.

To support the change accommodation cycle, we prepared a tool for analyzing malware behavior and an API to D-System Monitor serviced by D-Visor. To develop a new D-System Monitor, deep knowledge of malware is necessary. Our analysis tool employs symbolic execution to analyze modern malware that is quite difficult to analyze because they are tactically ciphered and obfuscated. We provide D-System Monitor Support API which allows us to develop, install, and uninstall D-System Monitors separately from D-Visor. This reduces the development and set-up cost of D-System Monitor, compared to the conventional method where we have to implement new monitoring functions in the VMM.

6. DEOS Development Support Tools

We plan to utilize existing software development support tools as much as possible. On the other hand, DEOS-specific tools for system software verification and dependability testing support are under development.

6.1 Software Verification Tools

The nature of open systems dictates that features in these systems will be added or modified even after the development of the systems is completed or they are deployed. When features are added or modified in existing systems, new bugs must not be introduced. One of the objectives of this project is to develop formal software verification tools for detecting defects in systems software [45, 46], thereby contributing to the improvement of Open Systems Dependability. In particular such tools will provide (part of) the evidence that is required by D-Case to be created (or modified) according to agreements among stakeholders in the "Design, Implementation, Verification, and Test" phase of the change accommodation cycle of the DEOS process. In order to achieve the objectives mentioned above, this project has been researching and developing two formal methods, model checking and type checking, for the formal verification of C programs as shown in Fig. 14 and Fig.15.

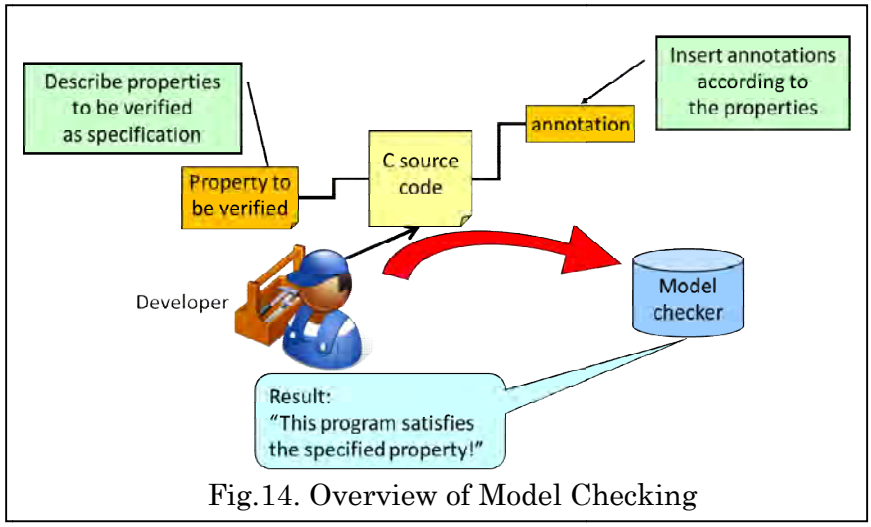

Fig.14. Overview of Model Checking

From the viewpoint of improving Open Systems Dependability, both of these methods have drawbacks and advantages. Model checking can verify relatively complex safety properties. However, this takes a long time. On the other hand, type checking can be done in a short time. However, it can only verify relatively simple safety properties. To address these problems, we combine the two methods in a complementary manner.

Model checking is a method of reading C programs, exploring all execution paths, and checking whether developer-specified properties (conditions) are satisfied or not. These properties are written in a specification language as conditions on the variables of the programs. Because the properties can be modified or added, measures to prevent a certain open systems failure which is caused by changes in the external environment or in user demands can be subjected to model checking by adding the conditions causing the failure to the properties to be checked.

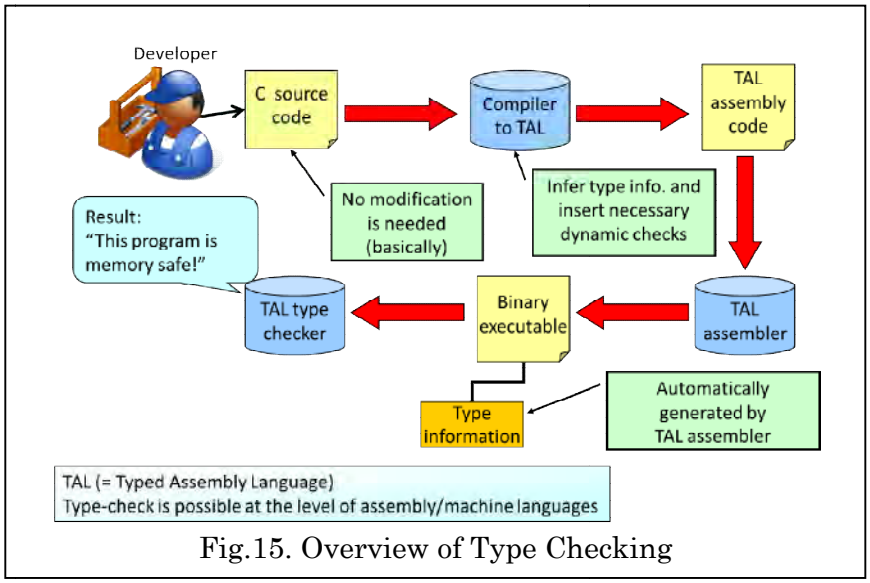

Fig.15. Overview of Type Checking

Type checking is a method for checking and ensuring that a program never performs illegal operations. For example, it ensures that the program never accesses outside of arrays or jumps to illegal addresses. First, programs written in C are compiled into a Typed Assembly Language (TAL), which is a type-safe assembly language handling the type information of programs. During this compilation, runtime checking codes are inserted where type-safety properties cannot be guaranteed statically. Then the generated TAL codes are assembled into binary executable forms, which also have the type information.

Thus the type-safety properties of the generated binary executable codes can also be checked. This makes it possible to ensure the simple safety properties of programs continuously even when a system needs to be modified because of changes in external environment or stakeholders' requirements.

6.2 Dependability Test Support Tools

When attempting to fulfill the dependability requirements of a system using D-Case, evidence is needed to show whether the system requirements are satisfied under several anticipated anomaly conditions. The limits of the system in protecting against these anomaly conditions must be calculated beforehand by the necessary quantitative measurements. In addition, after system updates, it must be verified that no problems arise in a system operation test. The cause of anomalies observed during actual operation must be analyzed in order to ensure accountability to stakeholders. In order to support systematic measurement of several dependability metrics in D-Case, DS-Bench and Test-Env are being developed [47, 48, 49, and 50]. DS-Bench measures dependability metrics and Test-Env performs rapid system tests. Anomalies include hardware faults, software bugs, overload, and human error. A common evaluation environment in which various anomaly loads can be evaluated must be implemented systematically. In order to observe anomaly conditions or evaluate system operation, a tremendous number of system tests must be carried out. This can only be done by automating the test process and managing computing resources appropriately. Complex testing using many test patterns must be accelerated.

The outline of DS-Bench/Test-Env is shown in Fig.16. DS-Bench/Test-Env is able to inject anomaly loads, which simulate anomaly situations such as component failure or overload, while conducting a benchmark program, which measures some dependability metrics (e.g. performance, downtime, etc.). DS-Bench/Test-Env partially shares the goals of the DBench project [26] which was carried out from the late 1990s to the early 2000s. However, we aim for the benchmarks to be reused, unlike DBench, and we also aim to make a database of benchmark programs and to make scripts specifying how to execute the benchmarking based on a fault scenario.

DS-Bench provides a web user interface for users to configure benchmark tests based on a fault scenario, typical benchmark programs, and anomaly loads set to emulate anomaly states. Users may append benchmarks and anomaly loads to the database if necessary. DS-Bench controls each machine to be evaluated, and organizes the program and fault injection based on the specific scenario. The fault scenario that describes when to generate faults at what workloads, and with which measurement tools, is described in XML. The results of the execution based on a fault scenario are also described in XML format, and are stored in a database. This database provides evidence of how the system responds to overload or faults.

DS-Bench/Test-Env involves two types of benchmark environments: physical machines and virtual machines. If the benchmark is for evaluating performance, a physical machine environment is directly used. Anomaly behavior caused by software problems, such as software bugs and overloads, and hardware malfunctions that can be caused by manipulating hardware devices other than computer components, such as network disconnection and power shutdown,

are also simulated using physical machines. To deal with faults in memory flip, I/O devices, etc., these faults can be simulated on virtual machines. If the system requires many computers and it is difficult to prepare the physical machines, virtual machines may also be used.

The environment for executing the dependability benchmark is provided by Test-Env. Test-Env manages the facilities for the benchmark including commodity computers, network switches, and PDUs (power strips), and assigns these resources based on the requirements of DS-Bench before the startup of the benchmark. Test-Env also offers groups of virtual machines as a private cloud-computing environment, and the cloud management software sets up a virtual machine as necessary. Anomaly loads to simulate hardware malfunctions are injected by Test-Env. Test-Env generates the faults at appropriate times under the scenario given by DS-Bench.

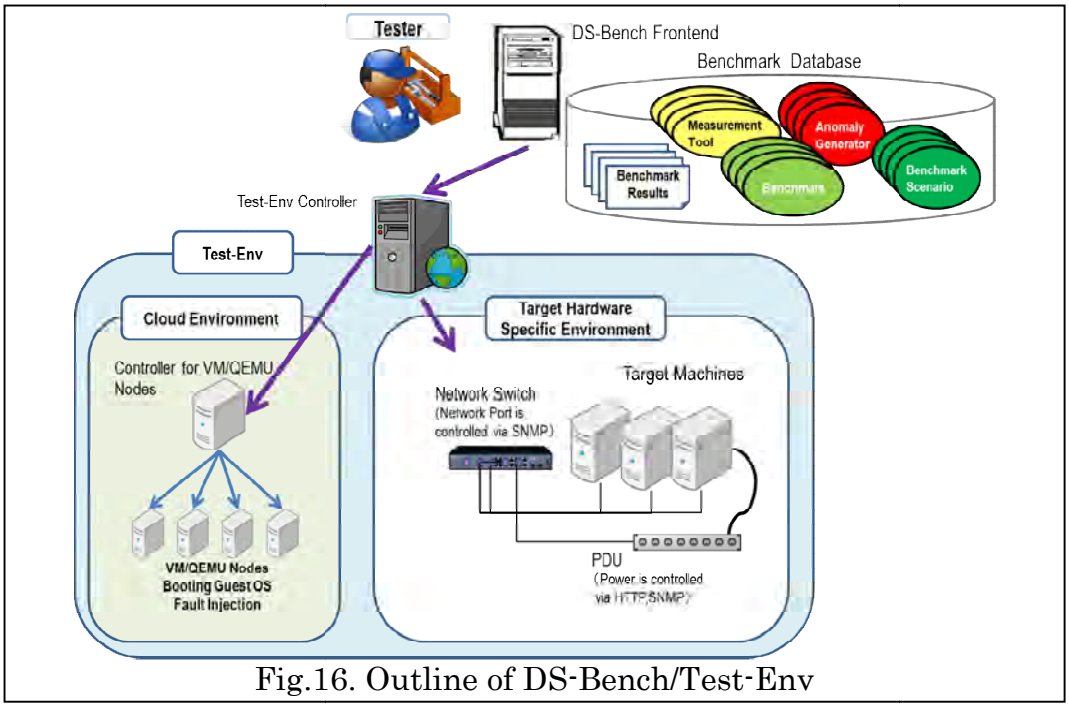

Fig.16. Outline of DS-Bench/Test-Env

7. Effectiveness of Applying DEOS: A Case Study

In this chapter, the effectiveness of DEOS process and architecture in dealing with system failures when it is applied to open systems will be described. An actual incident in the past will be used as a case. Possible failures in future large scale systems will also be discussed.

1) A case where software defect causes system failure

An IC card for automatic train turnstiles is commonly used now. One morning when the system was starting for the operation of the day, the system did not operate properly. There was no clue as to the cause of the failure, and it did not come into operation for several hours. The affected railway companies had no choice but to open up all of the automatic train turnstiles for free. It took time to find out real cause of the failure.

If DEOS had been applied to the systems, there would have been agreed-upon responsive action described in the failure response cycle of D-Case. For example, there may have been instruction to set the system back to its status just before it crashed. The system had operated normally until the night before. Most likely a change made during the night, when the system was not in operation, had triggered the system failure. Investigation, revealed that there was no update of the program. However, it was found that data specifying expired IC cards to be revoked was downloaded. If it had been decided to re-start operation based on the older data including the expired IC cards, the system could have been brought back to its status before the failure and could have continued operation with one-day-old data. This would have allowed the expired IC cards to continue to be used, but also would have allowed the whole system to keep operating.

Another possible action could have been to disconnect each automatic train turnstile from the servers. An automatic train turnstile is designed to store the transaction data of the past few days locally. Its operation could continue with its local data for a while without server connection. With D-Case describing the agreement of stakeholders and with D-Script generated from D-Case, urgent responsive action could have been possible.

The next step would have been to narrow down the potential causes to specific software modules, using the system log. After the cause of the failure was identified, the required changes that would prevent a similar failure from occurring would be proposed to stakeholders. They would evaluate these proposed changes in requirements, and decide whether to agree with these. When agreed upon, the requirements would be implemented through the change accommodation cycle. To continue the services, it is needed to achieve accountability by explaining the causes of the system crash, the actions to be taken to resolve the problem, and the improvements to be made in the system.

2) A case of performance unbalance within the system causing system failure

The structure of a system providing services on the Internet may differ depending on its scale. Most systems with a considerable number of accesses or transactions have structures with 3 layers: a web server to interact with end users, application servers to execute applications, and a DB server to handle data. The environment of these IT systems changes while the system is in operation in many cases. To accommodate an increase in the number of accesses and to maximize the business, the system expands servers and improves application software. In many cases, such changes have to be done while the system is in operation. The expansion or improvement of a part of the system without sufficient consideration of its relationship with other parts of the system can cause unexpected system failure. For example, if only the web server is expanded to accommodate increased access, it may cause overload of the application server. If a new service is added on the application server, it may cause system crash due to overload. Changes in the system structure must be carefully performed considering the balance of the capacity and performance of the whole system.

By applying DEOS to a system of this kind, we can utilize DS-Bench/Test Env before we make such changes. Of course, all the change requirements go through the DEOS process so that D-Case description is made and D-Script is given for necessary responsive actions. Even if we faithfully carry out the DEOS process and consider all the possible cases of use, a failure could occur. In such a case, a proper responsive action will be taken and then the change accommodation cycle will be carried out in a way similar to Case 1) described above.

3) A case of system integration causing system failure

Today's computer systems use off-the-shelf software or legacy codes developed for the previous generation of systems as components. An enormous volume of integration testing is required before a cut-over to a service carried out with a new computer system or before the launch of a product with an embedded system. Even then, though, such testing may not cover all of the possible cases of use. Especially, in the case of a system with software modules that are loaded or unloaded dynamically, it is almost impossible to cover all possible cases during the test. Defining recovery procedures for unexpected behavior of the system will help, and could be the best backup plan to deal with failures.

In order to apply DEOS to a system of this kind, there must be a recovery procedure described in D-Script. D-RE provides the capability for system checkpoint-restart and application checkpoint-restart. These checkpoint-restart mechanisms allow rollback of the system to a point before the failure. D-RE provides APIs to execute the checkpoint function in several layers, so that the function can be carried out in accordance with the level of failure. An application program that uses the D-RE APIs in cooperation with the D-RE, such as checkpoint execution, safe start of application program, or safe termination of application program, is called a D-Aware Application in DEOS. Using API for D-Aware Applications enables the application to cooperate with the system and to perform safe operation. For applications not designed as D-Aware Applications, D-RE provides the function of quick restart, and the system can thus perform restart. This function will help in temporary recovery from a failure that is difficult to reproduce. Accountability requires determination of the nature of the system crash and the immediate action taken thereafter, and report end users on these.

4) A case where aging of the system causes system failure

One example of bugs is a memory-leak, which often leads to system malfunction. It is very difficult to detect particular software to cause a memory-leak. GC (Garbage Collection), which may help in avoiding a memory-leak, may not be supported in all programming languages. Leaving a memory-leak as is will reduce memory space, which then leads to decrease in the system's performance or stoppage of the system's operation.

If DEOS is applied to the system, D-RE has the capability to monitor the decrease of memory space and rejuvenate the system automatically, which will help keep the system status from changing unpredictably.

5) A case where software-license-expiration causes system failure

Today's systems usually utilize off-the-shelf software. A license agreement thus is required for these systems. Usually the license agreement is valid only for a specified period of time, and the license is updated before it expires. In most cases, such off-the-shelf software breaks off their operation automatically when the license expires. For a system with tens or hundreds of licensed software modules, all of the licenses may not be properly administrated. There are cases where an oversight here causes system failure.

If the DEOS is applied to the system, the system goes through daily inspection while the system is in operation, and such failures are avoided. The DEOS Runtime Environment has the capability to set the system clock separately for each system container. Using this capability, a part of the system may operate based on the system's future schedule, identify pieces of software whose licenses will expire at a specific time, and have the operator take action for license update. This may not be feasible for a system which requires actual data transaction for the system to

operate, or which is in operation for 24 hours 7 days and does not allow dry runs assuming a future time, but may be effective for other systems that can do dry runs and daily inspections every day after the day's operation with the date and time set for the next day.

8. Steps toward Practical Use

8.1. International Standards

8.1.1. Importance of International Standards

To exploit the results of the DEOS project, we recognize the importance of following international standards for the following reasons: to share the concept of Open Systems Dependability, to provide guidelines for social infrastructures, and to achieve common use of tools.

(1) Sharing the concepts of Open Systems Dependability

As mentioned in Section 2.1, there already exist some international standards trying to deal with problems arising from "openness", even though this term is not explicitly mentioned. For example, today it cannot be expected that complex systems can be kept totally safe. In these situations, the concept "functional safety" in such as IEC 61508 and ISO 26262 can be regarded as a cost-effective approach to keep and assess certain safety levels. Moreover, ISO/IEC 20000 for IT service management and ISO/IEC 27000 for security management are management standards for sustaining certain services or security levels in ever-changing systems, i.e. the main characteristics of open systems. However these standards do not share any common concept that connects them horizontally. In other words, these standards have been established as temporary solutions in order to tackle recently arising problem situations case-by-case. Reflecting upon these circumstances, we can conclude that it is necessary to establish an upper-level standard to cover various different areas and to share common concepts relating to Open Systems Dependability.

(2) Provide guidelines for social infrastructures

In this project, we consider that two key activities, consensus building among the stakeholders and achievement of accountability, are indispensable to ensure the dependability of open systems. DEOS process and DEOS architecture are an effective way to realize these key activities.

To take steps toward the practical execution of these activities, it is necessary to construct frameworks that include widely accepted social rules for consensus building, a standard format, a database of assurance cases showing that dependability is ensured, and a framework for consumers to join the consensus building processes.

Certification can be considered as an instance of achieving the agreement of society to certain processes. This implies that certification frameworks have a significant role in achieving Open Systems Dependability.

(3) Achieving common use of tools

In general, in order for a technology to permeate society, the dissemination of tools for the technology can be a decisive process. For Open Systems Dependability, practical tools for reaching agreements and determining accountability are essential. Standards for certifying or handling tools can aid in their dissemination.

8.1.2. Plan for Standards

Below we describe outlines of standards we intend to establish and our activities for them.

(1) Standard for the concept of Open Systems Dependability

The IEC 60300 series international standards define the concept of dependability management. IEC 60300-1 is being revised and we are now participating in this revision activity. In this activity, some concepts developed by this DEOS project are expected to be employed, with the provision that existing IEC 60300-1 is not based on the concept of Open Systems Dependability. We are planning to make a new proposal to IEC TC56 or another suitable international standardization body that work should be done to apply the OSD concept.

(2) Standards for the DEOS process and architecture

ISO/IEC 15026 is the international standard for defining integrity levels for reducing or managing risks. "Integrity level" is a general concept encompassing safety integrity levels (SIL). ISO/IEC 15026 is now being revised for system and software assurance. We have joined this revision activity as editors. The term "assurance" originally means "confidence" or "promise". However, this term as used in the phrase "system assurance" means "high-reliability". ISO/IEC 15026-2, a standard for the concept of assurance cases, has been already published.

We participate in some working groups in OMG (Object Management Group), an international organization for standardization of UML, and have joined a new standardization project for "consumer product". We are promoting dependability as a major concept in this project. Our continuing efforts there will reflect our ideas, such as D-Case and the DEOS process. In a revised version of ISO/IEC 15026, processes for realizing assurance through the life cycle will be included. Our goal is to develop processes that are consistent with the DEOS process and that have no implementation problems. We are striving for the concept of the DEOS process and to be reflected in the revised ISO/IEC 15026. We also are working with the Open Group, a global consortium set up to define TOGAF (The Open Group Architecture Framework) and are developing a process to utilize of D-Case in TOGAF.

(3) Standards for implementations and tools

In OMG, we are working to establish a standard for assurance cases reflecting the idea of D-Case patterns and D-Case related tools (D-Case Editor, D-Case Viewer, and D-Case Verifier).

8.2. DEOS Consortium

The DEOS project has many accomplishments, including the concept of Open Systems Dependability (OSD), the DEOS process, and architecture, run-time environments, and various tools. The ultimate goal of the DEOS project will be achieved, however, only when these deliverables are actually used, maintained, and improved by many users. The establishment of a consortium is one way to achieve this goal. As one of our DEOS project activities, we will strive to establish such a consortium.

The tentative objectives of this consortium are as follows:

1. Understanding, promoting, and further developing the concept of Open Systems Dependability,
2. Establishment of industry-wide and society-approved standards,
3. Promoting the DEOS process and architecture in various application areas, aiding in the growth of various businesses related to Open Systems Dependability, and

4. Development and maintenance of the deliverables and new products of DEOS.

8.3. Handling of Intellectual Property Rights and Copyright

The intellectual property rights, including copyright, of each deliverable primarily belong to the research organization that produced it. The intellectual properties should be licensed so as to best spread the use of the project's future deliverables, and a licensing policy will be decided upon accordingly. Intellectual property will be transferred to a specific organization if there is support from each research organization concerned. One of the proposals currently being considered is to share the IP with consortium members in the form of RAND (Reasonable and Non Discriminatory Licensing) so that DEOS-related deliverables will be widely used, and so that companies and organizations will be motivated to participate in the consortium and to contribute to the deliverables as much as possible.

9. References

1. H. Yasuura, “On Dependability”, T. Nanya, “Concept and Issues on Dependability”, K. Iwano, “Dependability in Social Services”, in Dependability Workshop Report (in Japanese), CRDS-FY2006-WR-07, CRDS, JST, March 2007
2. International Federation for Information Processing WG 10.4 on Dependable Computing and Fault Tolerance: http://www.dependability.org/wg10.4/
3. A. Avizienis, J.-C. Laprie, B. Randell, C. E. Landwehr, “ Basic Concepts and Taxonomy of Dependable and Secure Computing”, IEEE Trans. On Dependable and Secure Computing, Vol.1, No. 1, Jan.-March 2004
4. T. Kikuno, “Requirements for Dependability in the 21th Century”, Speech at the Kickoff Symposium of JST/CREST Dependable Embedded OS Project, December 2006
5. M. Tokoro, “On Designing Dependable Operating Systems for Social Infrastructures”, Keynote Speech at MPSoC, Awaji Island, Japan, June 25, 2007.
6. H. Yasuura, “Dependable Computing for Social Systems”, Journal of IEICE、Vol.90, No.5, pages 399-405, May 2007
7. T. Kano & Y. Kikuchi, “Dependable IT/Network”, NEC Technology, Vol.59, No.3, 2006, pages 6-10
8. M. Y. Hsiao, W. C. Carter, J. W. Thomas, and W. R. Stringfellow, “Reliability, Availability, and Serviceability of IBM Computer Systems: A Quarter Century of Progress”, IBM J. Res. Develop., Vol. 25, No. 5, 1981, pages 453-465
9. A. G. Ganek and T. A. Corbi, “The Dawning of the Autonomic Computing Era”, IBM Systems Journal, Vol. 42, No. 1, 2003, pages 5-18
10. “An Architectural Blueprint for Autonomic Computing, 4th edition”, IBM Autonomic Computing White Paper, June 2006
11. Autonomic Computing: http://www-03.ibm.com/autonomic
12. Mario Tokoro, ”Research and Development at Technology Maturity Stage”, Journal of IEICE, Vol.90, No.9, 2007, pages 742-744
13. Mario Tokoro (ed.), “Open System Science”, IOS Press, 2010
14. H. B. Diab, A. Y. Zomaya, “Dependable Computing Systems”, Wiley-Interscience
15. G. M. Koob, C. G. Lau, “Foundations of Dependable Computing”, Kluwer Academic Publishers
16. M. C. Huebscher, J. A. McCann, “A Survey of Autonomic Computing”, ACM Computing Surveys, Vol. 40, No.3, Article 7, August 2008, pages 7:1-7:28
17. K. Matsuda, Foreword, IPA SEC Journal No.16, Volume 5, No. 1(Volume 16), 2009, page 1
18. T. Forbath, interview “Japanese Company Should Break-Away from the 20th century Development Process ”, NIKKEI ELECTRONICS 2009.2.23, page 29
19. A. Avizienis, “ Design of fault-tolerant computers”, In Proc. 1967 Fall Joint Computer Conf., AFIPS Conf. Proc.Vol.31, pages 733-743, 1967
20. Nassim Nicholas Taleb,The Black Swan: The Impact of the Highly Improbable, Random House.
21. Leveson, Nancy G., Safeware: System Safety and Computers, Pearson Education.
22. Failure Chains: Warning of the Prius Recall, Nikkei BP press(in Japanese)
23. Owada Naotaka and others, Why Systems Go Down. Nikkei BP press (in Japanese)
24. H. B. Diab, A. Y. Zomaya, “Dependable Computing Systems”, Wiley-Interscience
25. G. M. Koob, C. G. Lau, “Foundations of Dependable Computing”, Kluwer Academic Publishers
26. K.Kanoun, L.Spainhower, “Dependability Benchmarking for Computer Systems”, IEEE Computer Society
27. B.Kirwan,"A Guide to Practical Human Reliability Assessment”, CRC Press
28. Science Council of Japan, Board of Informatics, Sectional committee of Security and Dependability, “Toward the Popularization of IT Infrastructure Realizing Safety and Security”, (Chair: Hideki Imai), 2008/6/26. http://www.scj.go.jp/ja/info/kohyo/pdf/kohyo-20-t58-4.pdf
29. O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Structured Programming, Academic Press, London, 1972 ISBN 0-12-200550-3
30. Birtwistle, G.M. (Graham M.) 1973. SIMULA Begin. Philadelphia, Auerbach.
31. Humphrey, Watts (March 1988). "Characterizing the Software Process: a Maturity Framework". IEEE Software 5 (2): 73–79. doi:10.1109/52.2014. http://www.sei.cmu.edu/reports/87tr011.pdf.
32. Humphrey, Watts (1989). Managing the Software Process. Addison Wesley. ISBN 0201180952.
33. Capability Maturity Model Integration: http://www.sei.cmu.edu/cmmi/
34. Ultra-Large-Scale Systems: The Software Challenge of the Future, http://www.sei.cmu.edu/library/assets/ULS_Book20062.pdf
35. Dependable Operating Systems for Embedded Systems Aiming at Practical Applications Research Area (DEOS Project) White Paper Version1.0, DEOS-FY2009-WP-01, JST, Sep. 1, 2009

36. Dependable Operating Systems for Embedded Systems Aiming at Practical Applications Research Area (DEOS Project) White Paper Version2.0, DEOS-FY2010-WP-02, JST, Dec. 1, 2010
37. T P Kelly, R A Weaver, "The Goal Structuring Notation - A Safety Argument Notation", In proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases, July 2004
38. Workshop on Assurance Cases: Best Practices, Possible, Obstacles, and Future Opportunities, DSN 2004
39. European Organisation for the Safety of Air Navigation. Safety case development manual. European Air Traffic Management, 2006
40. Railtrack. Yellow Book 3. Engineering Safety Management Issue3, Vol. 1, Vol. 2, 2000.
41. Guidance for Industry and FDA Staff - Total Product Life Cycle: Infusion Pump - Premarket Notification [510(k)] Submissions http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm206153.htm#6
42. City University London Centre for Software Reliability http://www.city.ac.uk/informatics/school-organisation/centre-for-software-reliability/research
43. D-Case Editor http://www.il.is.s.u-tokyo.ac.jp/deos/dcase/
44. Kimio Kuramitsu. Konoha: implementing a static scripting language with dynamic behaviors. In Proceeding S3 '10 Workshop on Self-Sustaining Systems, ACM Press, 2010.
45. Takahiro Kosakai, Toshiyuki Maeda and Akinori Yonezawa, "Compiling C Programs into a Strongly Typed Assembly Language", In Proceedings of the 12th Asian Computing Science Conference (ASIAN 2007). pp. 17-32. Doha, Qatar, Dec. 2007.
46. Motohiko Matsuda, Toshiyuki Maeda, and Akinori Yonezawa, "Towards Design and Implementation of Model Checker for System Software", In Proceeding of the 1st International Workshop on Software Technologies for Future Dependable Distributted Systems (STFSSD 2009), Tokyo, Japan, January 2009.
47. Shimpei Kato, Hajime Fujita, Jin Nakazawa, Mohohiko Matsuda, Toshiyuki Maeda, Yuki Kinebuchi, Toshihiro Hanawa, Shin'ichi Miura, Yoichi Ishiwata, Yutaka Matsuno, Hiroki Takamura, Hiroshi Yamada, Tetsuya Yoshida, Kimio Kuramitsu, Midori Sugaya, and Yutaka Ishikawa, "A Benchmark Framework for Dependable Systems", In Proceedings of DSW2007, pp.171-178.
48. Toshihiro Hanawa, Hitoshi Koizumi, Takayuki Banzai, Mitsuhisa Sato, and Shin'ichi Miura, "Customizing Virtual Machine with Fault Injector by Integrating with SpecC Device Model for a software testing environment D-Cloud", the 16th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '10), pp. 47-54, Dec. 2010. doi: 10.1109/PRDC.2010.37
49. Takayuki Banzai, Hitoshi Koizumi, Ryo Kanbayashi, Takayuki Imada, Toshihiro Hanawa, and Mitsuhisa Sato, "D-Cloud: Design of a Software Testing Environment for Reliable Distributed Systems Using Cloud Computing Technology", the 2nd International Symposium on Cloud Computing (Cloud 2010) in conjunction with the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid 2010), pp. 631-636, May 2010. doi: 10.1109/CCGRID.2010.72
50. Toshihiro Hanawa, Takayuki Banzai, Hitoshi Koizumi, Ryo Kanbayashi, Takayuki Imada, and Mitsuhisa Sato, "Large-Scale Software Testing Environment Using Cloud Computing Technology for Dependable Parallel and Distributed Systems", the 2nd International Workshop on Software Testing in the Cloud (STITC 2010), co-located with the 3rd IEEE International Conference on Software Testing, Verification, and Validation (ICST 2010), pp. 428-433, Apr. 2010. doi: 10.1109/ICSTW.2010.59
51. Daniel P. Siewiorek, Robert S. Swarz, "Reliable Computer Systems: Design and Evaluation", Third Edition. A K Peters/CRC Press. 1998.
52. A.M. Davis, Just Enough Requirements Management: Where Software Development Meets Marketing, Dorset House. 2005.
53. Object Management Group/Business Process Management Initiative: http://www.bpmn.org/
54. Smalltalk: http://www.smalltalk.org/main/
55. Nancy G. Leveson: "A new accident model for engineering safer systems", Safety Science, Volume 42, Issue 4, Pages 237-270, April 2004.

Appendix

A.1. DEOS Research Area Organization

The research area of "Dependable Operating Systems for Embedded Systems Aiming at Practical Applications" is led by a research supervisor and a deputy research supervisor, with support of several area advisors. They advise the research teams as to the direction of the research and development activities, and also evaluate the progress of the activities. The area management advisors give suggestions on directions in the development of research results into highly practical applications. Several engineers or managers from technology companies have been appointed as research promotion board members. They support the research activities, and have regular meetings with researchers to advise them from the view point of users of the project deliverables.

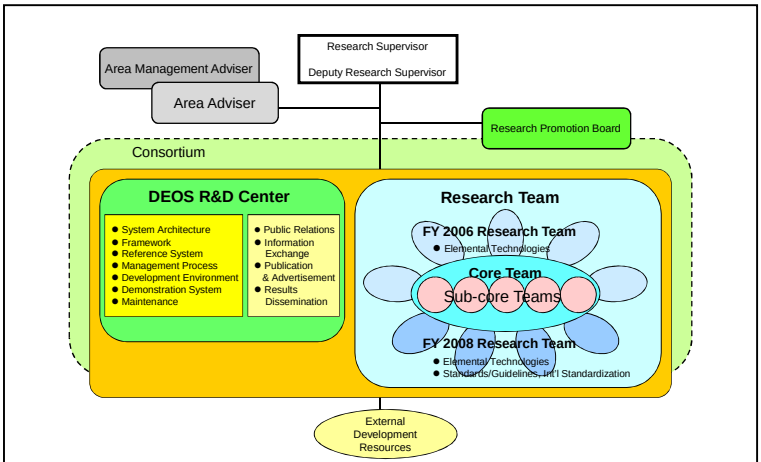

Fig.17. Research Area Formation

This project formed five research teams in 2006 and added four new teams in 2008. The teams appointed in 2006 started their activity mainly with elemental technologies such as virtual machines, virtualization of server groups, systems software verification, benchmarking, fault simulation, and real-time and low power consumption systems, continually investigating and discussing the characteristics of the targeted systems and the concept of dependability. The teams appointed in 2008 supported the activity of the 2006 teams, and together established the DEOS process and the DEOS architecture. The 2008 teams started research and development on requirements analysis, consensus building processes, descriptions, security, etc., and also are making new international standards to reflect the outcomes of the project. The 2006 teams and 2008 teams will continue research and development activities until March 2012 and March 2014, respectively

A core team was formed in 2008 with researchers appointed from the research teams. The core team oversees the activities of the research teams, and has been working to define the direction of the project and decide upon the research and development themes of the DEOS project, in parallel with the research activities of each research team. In 2010, sub-core teams were formed under the core team to study further the themes decided upon, and each team is pursuing their studies jointly with members of other research teams. They focus on research and development of major components of the DEOS process and architecture. Currently there are sub-core teams for D-Case & Metrics, D-Script & Monitor, VM & Multi-OS, System Software Verification, and DS-Bench & Test-Env, all of them making progress in their activities. The sub-core teams have been and will be regrouped as required to maximize the outcome of the project.

The Dependable Embedded OS Research and Development Center (DEOS R&D Center), established in 2007, works to make the research results into practical use by such activities as co-evaluation of research outcomes with enterprises and actual use of research results in those enterprises' products and services, by performing integration of the research results, refactoring of software considering intellectual property rights and maintainability, testing, evaluation of readiness for practical use, packaging, and so forth (Fig. 17).

A.2. DEOS Project Roadmap

The major milestones of the project and research activities are as follows (Fig. 18).

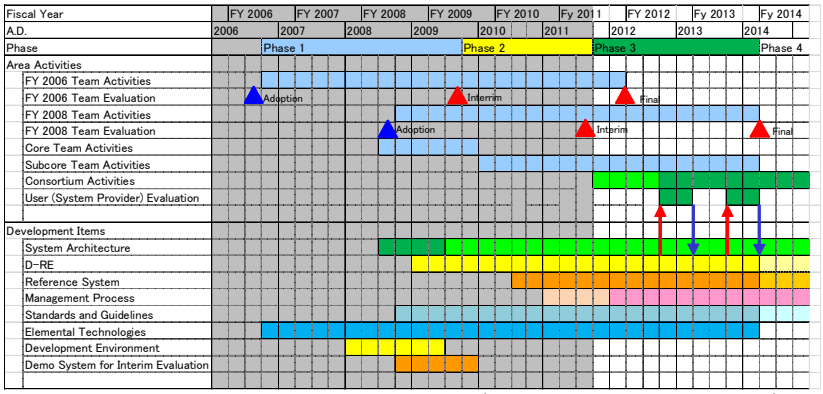

Fig.18. DEOS Project Roadmap (As of November 2011)

- Phase 1 (2006/10-2009/9): Establishment of the dependability concept; presentation of systems architecture containing major evaluation indexes and development/operation processes supporting said concept; and demonstration of the 2006 research team's demo system in which a number of elemental technologies were integrated. (Presentation completed 2009/9 by the 2006 research team.)
- Phase 2 (2009/10-2011/9): Implementation of a system architecture, D-RE, and tools which adopt the elemental technologies of the research teams; preparation to establish a consortium (or user organization) composed of potential user companies and research organizations; activities to bring required items up to international standards; demonstration of D-RE and tools which incorporate some of the elemental technologies from research teams. (The above will be presented in November, 2011. It will be the final public presentation of 2006 research teams and the interim public presentation of the 2008 research teams.)
- Phase 3 (2011/10-2014/3): Formation of a consortium startup committee; trial usage of D-RE and tools by potential consortium members; continuing evaluation and feedback; transition to actual development and commercialization; international standardization of required items.
- Phase 4 (2014/4-): Continuing utilization, maintenance and development of the project's products by the consortium; support for establishment of industry specific standards.

A.3. DEOS Research Area Members

Research Supervisor
Mario Tokoro, Sony Computer Science Laboratories, Inc.

Deputy Research Supervisor
Yoichi Muraoka, Waseda University

Area Advisors
Kazuo Iwano, IBM Japan, Ltd.
Koichiro Ochimizu, Japan Advanced Institute of Science and Technology
Tohru Kikuno, Osaka University
Kohichi Matsuda, Information-Technology Promotion Agency, Japan
Yoshiki Seo, NEC Corporation
Hidehiko Tanaka, Institute of Information Security
Hiroto Yasuura, Kyushu University

Research Directors
Yutaka Ishikawa, University of Tokyo

Satoshi Kagami, National Institute of Advanced Industrial Science and Technology
Yoshiki Kinoshita, National Institute of Advanced Industrial Science and Technology
Kenji Kono, Keio University
Kimio Kuramitsu, Yokohama National University
Toshiyuki Maeda, University of Tokyo
Tatsuo Nakajima, Waseda University
Mitsuhisa Sato, University of Tsukuba
Hideyuki Tokuda, Keio University

Research Promotion Board Members
Nobuhiro Asai, IBM Japan, Ltd.
Tadashi Morita, Sony Corporation
Masamichi Nakagawa, Panasonic Corporation
Takeshi Ohno, Yokogawa Electric Corporation
Ichiro Yamaura, Fuji Xerox Co., Ltd.
Kazutoshi Yokoyama, NTT Data Corporation

Area Management Advisors
Kazuo Kajimoto, Panasonic Corporation
Yuzuru Tanaka, Hokkaido University
Seishiro Tsuruho, HAL Tokyo
Daiji Nagaoka, Fuji Xerox Co., Ltd.

Dependable Embedded OS Research and Development Center
Makoto Yashiro, Japan Science and Technology Agency

A.4. Cases of Recent Failures

	Date occurred	Problem description	Cause
1	April 21, 2011	Service outage in Amazon ES2, etc. This failure also stopped services of Engine Yard, Heroku, and other websites.	Network configuration was mistaken in Amazon EBS, a storage service for a virtual machine.
2	August 10-12, 2010	Users were unable to access the Mixi service (the largest SNS site in Japan).	Memcached bug. Memcached daemon suddenly terminated when it had many connections/disconnections.
3	May 22, 2009	JavaScript embedded in cell phones from DoCoMo allowed unauthorized access to any websites. DoCoMo stopped its sales.	JavaScript implementation had a flaw, which allowed unauthorized access to any websites. Possibly there was an implementation problem of the SOP (Same Origin Policy) security policy used by web browsers. DoCoMo suspected that this policy was not well written in their specifications for their cell phones.
4	February 24, 2009	Google Apps Gmail users were unable to access their accounts.	An unexpected service disruption occurred during a routine maintenance event in a data center. In this case, users were directed towards an alternate data center in preparation for maintenance tasks, but the new software that optimizes the location of user data had the unexpected side effect of triggering a latent bug in the Gmail code. The bug caused the destination data center to become overloaded when users were directed to it, which in turn caused multiple downstream overload conditions as user traffic was automatically shifted in response to the failures.
5	September 14, 2008	Check-in terminals in several airports became non-operational, which caused the cancellation of several flights.	A certificate authorizing access from a terminal to the server system expired in the early morning of September 14th.
6	July 22, 2008	Some information from a derivatives	The working memory for one description was defined

		trading system could not be delivered to users.	to be much smaller than the actual required size, which caused the loss of several descriptions.
7	October 12, 2007	Ticket gates that have IC card readers in several locations in the Tokyo area were not operational.	There was a primitive error in the logic that breaks big data into small chunks during the transmission of some essential information from the server to ticket gates. This caused an infinite retry loop in receiving data on the ticket gate side.
8	September 19, 2006, October 23, 2006, May 16, 2007 and May 23, 2007	● It was difficult to make a phone call with IP connections. ● Connections between NTT East and NTT West were broken. ● The FLETS broadband service was discontinued for 7 hours.	● Bugs in the signaling server caused an overflow of signaling processing. ● An under estimation of capacity during planning caused an overflow of signal processing. ● Bad data was restored after maintenance, which caused the signaling server to stop. ● One router failure caused the propagation of bad routing information to other routers, but it was not sure that restarting that router was the solution.
9	May 27, 2007	ANA Ticketing and Check In systems stopped its service, and 130 flights were cancelled and 306 flights were delayed.	A network equipment problem caused by a hardware fault led to congestion between the host computers and terminals. However, the relationship between the network equipment problem event and the congestion event is unknown.
10	March 1, 2003	The FDP (Flight plan Data Processing) system stopped service, so that 215 flights were cancelled and 1500 flights were delayed.	One preexisting bug caused this failure. That bug was activated when a specific memory address was accessed. Tests of the system did not cover peak time in the morning.

A.5. Requirements Management Information

category	Requirements management information	objects
Stakeholder consensus-building	Agreements upon elicited requirements Agreements upon change requirements List of environmental conditions Design specifications on ordinary operations Design specifications on deviations List of expected events Policies dealing with un-expected events	Requirements Change requirements Environment Ordinary operation Responsive action expected events un-expected events
Accountability	Requirements list for accountability achievement List of responsibility conditions for environment Accountability achievement manual for ordinary operations Accountability achievement manual for deviations List of responsibility conditions for expected events List of discharge conditions for un-expected events	Requirements Environment Ordinary operation Responsive action expected events un-expected events
Service continuity	Requirements specification for service continuity List of conditions on service continuity environment Inspection report on service continuity requirements Ordinary operation manual Service continuity scenario specification Service confirmation report on expected events Service confirmation report on un-expected events	Requirements Environment System Ordinary operation Responsive action expected events un-expected events

A.6. Factors in Open Systems Failures

Classification	Underlying Factors	Specific Factors	System Failure Examples
＜Incompleteness＞ It often happens that the initial requirement specifications become inadequate, mismatch exists between specifications and implementation, and the system's behavior becomes difficult to fully understand at the time of shipment or service-startup.	★Incomplete understanding of system due to: ・Expansion of system ・System complexity ・System becoming a node of a network ・Frequent use of open source software ・Frequent use of black box software ・Frequent use of legacy code ★Changes of system components ★Configuration changes, integration issues	◆An error or omission in specification, design implementation, or testing, caused by the difficulty of understanding the whole system, particularly its software, due to its complexity and size ◆An error or omission in the specifications, design implementation, or testing, caused by discrepancies in characterizations of the system during the requirements management phase, or the development phase, or by an error in the documentation ◆An error in updating or amending procedures for administration, operation, or maintenance, or an error caused by expiration of license ◆Inconsistency between external specifications and actual operation of software components such as "black box" components or legacy codes ◆Lack of certainty in execution priority, sequence, timing of modules, etc. ◆Installation of components that were not planned for in development phase, or were not included in test configuration, such as components downloaded via a network while in operation	●Unexpected behavior prohibited by requirements ●Unexpected behavior prohibited by functional specifications ●Unexpected behavior prohibited by test specifications ●Unexpected behavior after integration of components, even though each component passed functional tests ●Unexpected behavior after updating of a component in operation ●System stops suddenly
＜Uncertainty＞ Changes of users' requirements or the usage environments throughout the lifecycle of the system make it difficult to completely predict the behavior of the system in the design or operation phases.	★Expectations change, human ability changes ★Operation environment changes ★Configuration changes, integration issues ★System ability balance issues ・Performance design ・Capacity planning ★System becomes a node of a network ★Changes of system components	◆Changes in system requirements caused by the changes in business goals ◆Changes in user's requirements or expectations of the system, or changes of operators' skill or capability during the maintenance and operation phases ◆Unexpected usage changes, such as those brought about by the significant increase in users, or number of units, as well as by changes in the economy ◆Update or alteration of a component's function and of the configuration of a system in operation by manual operation or through network ◆Changes in system resources (system aging, memory restriction, CPU clock limitation) ◆Changes in interacting parties over a network (response change, function change) ◆Unexpected connections through a network or increase of interaction through a network and intentional malicious attacks and intrusions from external entities	●Process execution slows down. ●End users wait in a long queue. ●System displays "please wait for a while" to requests that are accepted before. ●Electronic money not accepted by system after starting new service. ●Contributions or donations not accepted by a banking system. ●ATM did not work late at night through it was announced that 24 hour service had started. ●Credit card numbers were stolen and used unlawfully. ●System stops suddenly.

A.7. DEOS Process Evaluation Items List (Example)

Evaluation Items				Evaluation Check Mark (Points) per stage		
Top level	Second level	Third level	Fourth level	Plan	Implementation	Operation
Ordinary Operation State						
	Daily Inspection					
		Start-up Inspection executed?				
		Operation Diary Inspection executed?				
		Memory leak inspected?				
		Operation diary written?	Check writing rules of Operation Diary			
		Rehearsal executed?	Check Result			
	Failure Prediction, Failure Detection					
		How to predict?	What is predicted?			
		How to detect failures?				
		Reporting path, reporting step defined?	Operator trained?			
	Objectives Change, Environment Change					
		How to find stakeholders' objectives?				
		How to detect operating environment change?				
		Data gathering Path, Sensing Reporting defined ?				
	Auto Recovery					
		In-Operation Range defined?				
		Auto recovery conditions defined?				
Failure Response Cycle						
	Prediction and Avoidance					
		How to predict?	What is predicted?			
		How to avoid?				
	Quick Management					
		Quick management path defined?				
	Cause Analysis					
		Required information gathered?				
	Accountability Achievement					
		Responsible spokesman for each failure level defined?				
		Explanation template for each failure level defined?				
		Target audience for report on each failure level defined?				
		Explanation timing for each failure level defined?				
Change Accommodation Cycle						
	Requirements elicitation, Risk analysis					
		Requirements fully elicited?				
		Risks analysed?	Feasibility, safety, profitability, cost analysed?			
	Cause Analysis					
		Root cause of failure found?				
		Time and cost for cause analysis recorded?				
	Stakeholders' Agreement					
		Requirements, issues and concerns recorded?				
		Related documents updated?	Updated procedures defined?			
		Risks analysed well?	Feasibility, safety, profitability, cost analysed?			
		Agreement objectives defined?	Domain, levels defined?			
		Adequate number of stakeholders gathered?				
		Escalation path defined?				
	Design, Implementation, Verification, Test					
		Software verified?				
		Dependability test passed?				
	Accountability Achievement					
		Responsible spokesman for each failure level defined?				
		Explanation template for each failure level defined?				
		Target audience for report oneach failure level defined?				
		Explanation timing for each failure level defined?				

A.8. DEOS Project Glossary

Accountability achievement: Dealing with system failure by explaining the current status, the cause, and the recovery and future plan clearly to the stakeholders.

Agreed requirements: Requirements that are negotiated among stakeholders through consensus building.

Availability: The ability of a system to maintain a high operating ratio.

Black box: A system or software component whose internal design is unknown and which is integrated to a system based on external specifications only.

Cause analysis: Identification of the cause of failure.

Change accommodation cycle: A cycle of processes to accommodate changes in stakeholders' objectives or changes in the environment.

Closed system: A system whose boundary, functions, and structure are fixed.

Consortium: A group of organizations or persons who share objectives and intend to cooperate to achieve those objectives.

D-Aware application: Application program that is implemented to improve dependability using the APIs provided by D-RE.

D-Case: Method and tool for stakeholders' agreement.

DEOS architecture: An architecture that supports the DEOS process for a specific area or application. Such an architecture consists of various kinds of tools, a DEOS agreement description database, and a DEOS runtime environment.

DEOS process: The iterative process consisting of the "change accommodation cycle" and the "failure response cycle" to achieve Open Systems Dependability.

Dependability: Ability to deliver services that can justifiably be trusted. This is traditionally defined as the combination of availability, reliability, safety, integrity, and maintainability.

Deviated requirements: Requirements that have deviations from in-operation range.

D-Script: Dynamic responsive script language.

Elemental technology: Technology to realize individual required functions.

Elicited requirements: Identified requirements from each stakeholder's objectives and needs.

Evidence: Information which supports a claim through an argument.

Failure: Deviation from the range of operations (service level) specified in the stakeholders' agreement.

Failure prevention: Prevention of system failure involving prediction of system failures and anomalies.

Failure response cycle: A cycle of processes to respond to system failures.

Formal verification: Proof of correctness or incorrectness of programs by formal or mathematical methods.

Incident: An unfavorable event.

Incompleteness: The property that a system cannot be completely represented.

In-Operation: The state that provides the service continuously meeting stakeholders' agreement (in the range of service agreement level).

In-Operation range: The allowable operation range of service level that is agreed upon among stakeholders.

Integrity: The ability of a system to prevent improper system and data alteration.

Legacy software: Software whose designers and maintainers are not available for maintenance but which is still built-in and working in a system.

Manage: To solve a problem with efficient use of effort, and develop a state of the system in a favorable direction.

Metrics: Qualitative or quantitative indices to evaluate objects.

Model checking: Software verification technique that exhaustively checks whether a program works correctly by using a system model.

Open system: A system whose boundary, function, and structure change over time.

Open Systems Dependability: The ability to continuously prevent occurrence of open systems failures, to provide appropriate and quick action when such failures occur, to minimize the damage, to safely and continuously provide the services expected by users as much as possible, and to maintain accountability for the system operations and processes.

Open systems failure: Failure that is caused by incomplete and uncertain factors of the system.

Ordinary operation: Daily operation with appropriate periodical inspection and preventive maintenance, including efforts to avoid a repeat of previous failures by performing continuous improvement activities (Kaizen).

Ordinarily operated requirements: Requirements that are implemented and operated ordinarily in the real system.

Process: Steps (phases) of development and operation for systems or services.

Realized requirements: Requirements that are implemented and embodied in a real system.

Reliability: The ability of a system to perform a specified function for a specified period of time.

Responsive action: Quick and proper response to system failure, to achieve the desired system state.

Requirements elicitation: Identification of requirements from each stakeholder's objectives and needs through consensus building.

Requirements management: Method of managing requirements based on stakeholders' agreement.

Requirements state management model: Requirements are managed by the four states that are elicited, agreed, ordinarily operated, and deviated states. The elicited and agreed requirements states are managed at off line, whereas ordinarily operated, and deviated requirements states are managed at online.

Serviceability（Maintainability）: The ability to efficiently maintain a system through e.g., modification, debugging, and repair.

Security: The protection of a system from external attack that causes degradation of availability, reliability, serviceability, or integrity.

Specification description language: A language which describes the properties that a program needs to satisfy.

Stakeholder: Individual or organization who has a right, share, or interest in the service and/or system in question. For example, users of services or products (the whole society in the case of systems for social infrastructure), providers of services or products, providers of systems (designers, developers, maintainers, operators, and providers of hardware), and certifiers (authorizers) of services or products are stakeholders.

Stakeholders' Agreement: Consensus among the stakeholders on requirements.

System architecture: A system's concept, fundamental functions, and structure.

TAL: The abbreviation of Typed Assembly Language. Assembly language in which values are annotated with their types, so that type checking is possible.

Type checking: Software verification technique that identifies errors in a program on the basis of the presence of explicitly or implicitly stated variable types.

Uncertainty: The possibility that a property of a system may change and cannot be predicted in advance.

Virtualization technology: Technology for managing computational resources by abstraction of the system, and for enabling the logical partition of computational resources.

A.9. DEOS Project Abbreviation

ADD: Agreement Description Database
DEOS: Dependable Embedded Operating Systems or Dependability Engineering for Open Systems
D-DST: DEOS Development Support Tools
D-RE: DEOS Runtime Environment
OSD: Open Systems Dependability

A10. Related Standards and Organizations

Standards

- IEC 61508: Functional Safety
  http://www.iec.ch/zone/fsafety/fsafety_entry.htm
- IEC 60300-1: Dependability Management
  http://www.iec.ch/cgi-bin/procgi.pl/www/iecwww.p?wwwlang=E&wwwprog=sea22.p&search=text&searchfor=IEC+60300-1&submit=OK
- IEC 60300-2: Dependability Program Elements and Tasks
  http://www.iec.ch/cgi-bin/procgi.pl/www/iecwww.p?wwwlang=E&wwwprog=sea22.p&search=text&searchfor=IEC+60300-2&submit=OK
- ISO/IEC 12207: Software Life Cycle Processes
  http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=21208
- ISO/IEC 15288: System Life Cycle Processes
  http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=43564

- ISO 26262: Road vehicles -- Functional safety http://www.iso.org/iso/catalogue_detail.htm?csnumber=43464
- ISO/IEC 20000: Information technology -- Service management http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=51986
- ISO/IEC 27000: Information technology -- Security techniques -- Information security management systems http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=56891

Process Guides

- CMMI: Capability Maturity Model® Integration http://www.sei.cmu.edu/cmmi/
- DO-178B: Software Considerations in Airborne Systems and Equipment Certification http://www.rtca.org/
- MISRA-C: http://www.misra-c.com/
- IEC 61713: Software dependability through the software life-cycle processes- Application guide http://www.iec.ch/cgi-bin/procgi.pl/www/iecwww.p?wwwlang=E&wwwprog=sea22.p&search=text&searchfor=IEC+61713&submit=OK
- IEC 62347: Guidance on system dependability specifications http://www.iec.ch/cgi-bin/procgi.pl/www/iecwww.p?wwwlang=E&wwwprog=sea22.p&search=text&searchfor=IEC+62347&submit=OK

Software

- SELinux: Security-Enhanced Linux http://www.nsa.gov/research/selinux/index.shtml
- AppArmor®: a Linux application security framework http://www.novell.com/linux/security/apparmor//
- Xen® hypervisor: the powerful open source industry standard for virtualization http://www.xen.org/

Related Organizations/Projects

- ISO: International Organization for Standardization http://www.iso.org/iso/home.htm
- IEC: International Electrotechnical Commission http://www.iec.ch/
- ISO/IEC JTC1: Joint ISO/IEC Technical Committee 1 http://www.iso.org/iso/standards_development/technical_committees/list_of_iso_technical_committees/iso_technical_committee.htm?commid=45020
- IEC/TC56: Technical Committee 56: IEC Technical Committee for International Standards in the field of Dependability http://tc56.iec.ch/index-tc56.html
- OpenTC Consortium: Open Trusted Computing Consortium http://www.opentc.net/
- Linux-HA Project: High Availability Linux Project http://linux-ha.org/
- Carrier Grade Linux Workgroup: http://www.linuxfoundation.org/en/Carrier_Grade_Linux
- TCG: Trusted Computing Group http://www.trustedcomputinggroup.org/home
- ERTOS Group: Embedded Real-Time Operating-Systems Group http://ertos.nicta.com.au/
- ARTEMIS: Advanced Research & Technology for Embedded Intelligence and Systems http://www.artemis.eu/
- CPS Program: Cyber-Physical Systems Program http://www.nsf.gov/pubs/2008/nsf08611/nsf08611.htm
- MISRA: Motor Industry Software Reliability Association http://www.misra.org.uk/
- AUTOSAR: Automotive Open System Architecture http://www.autosar.org/
- JasPar: Japan Automotive Software Platform and Architecture http://www.jaspar.jp/
- FlexRay Consortium: Consortium for the communications system for advanced automotive control applications http://www.flexray.com/
- NoTA: Network on Terminal Architecture http://www.notaworld.org/
- LIMO Foundation: Industry Consortium dedicated to Linux-based operating system for mobile devices http://www.limofoundation.org/
- CoBIT: Control Objectives for Information and related Technology http://www.isaca.org/Knowledge-Center/COBIT/Pages/Overview.aspx
- ITIL: Information Technology Infrastructure Library http://www.itil.org/en/vomkennen/itil/index.php

DEOS Project