DEOS-FY2010-WP-02E

JST-CREST

Dependable Operating Systems for Embedded Systems Aiming at Practical Applications

— DEOS Project —

DEOS: Dependable Embedded Operating System

White Paper

Version 2.0a

2010/12/01

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

Japan Science and Technology Agency

Preface to Version 2.0a

It has been one year since we published the DEOS Project White Paper Version 1.0 on September 1, 2009. The project is progressing in accordance with the plan described in the White Paper Version 1.0, and implementation of the technology as well as software development have started. We are pleased to publish this second version of the DEOS Project White Paper to present our recent progress.

Changes to chapters of Version 2.0a from Version 1.0:

Version 1.0		Version 2.0		Major difference
Chapter 1	Background	Chapter 1	Background	Part added
Chapter 2	Dependability	Chapter 2	Dependability	Little change
		Chapter 3	Phases of DEOS Processes	New
Chapter 3	Project Direction	Chapter 4	Project Direction	Renewed
Chapter 4	Items for Research & Development			Merged into Chapter 4
		Chapter 5	Major Research and Development Status	New
Chapter 5	Research and Development Organization	Chapter 6	Research and Development Organization	Part added
Chapter 6	Roadmap	Chapter 7	Roadmap	Renewed
Chapter 7	Issues and Concerns	Chapter 8	Further Issues for Practical Applications	Renewed
		Chapter 9	References	Renewed
Chapter 8	Appendix	Chapter 10	Appendix	Renewed

Contributors

Kenji Kono	Keio University
Jin Nakazawa	Keio University
Hideyuki Tokuda	Keio University
Hiroshi Yamada	Keio University
Hajime Fujita	University of Tokyo
Yutaka Ishikawa	University of Tokyo
Yutaka Matsuno	University of Tokyo
Toshiyuki Maeda	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
Kimio Kuramitsu	Yokohama National University
Midori Sugaya	Yokohama National University
Tatsuo Nakajima	Waseda 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
Toshinori Takai	National Institute of Advanced Industrial Science and Technology
Makoto Takeyama	National Institute of Advanced Industrial Science and Technology
Ichiro Yamaura	Fuji Xerox Co. Ltd.
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

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

Table of Contents

1 Background 6
2 Dependability 7
2.1 Brief Historical Review 7
2.2 Environment of Embedded Systems and Requirements 8
2.3 Open Systems Dependability 10
2.4 DEOS Process 12
3 Phases of the DEOS Process 13
3.1 System Change-Requests based on Stakeholders' Agreement 13
3.2 Design, Implementation, Verification, and Testing 14
3.3 Achievement of Accountability 14
3.4 Failure Prevention 15
3.5 Responsive Action 16
3.6 Cause Analysis 17
3.7 Normal Operations 17
4 Project Direction 18
4.1 Project Goal 18
4.2 Project Objectives 18
4.3 Realizing Open Systems Dependability 18
4.4 Project Deliverables 22
5 Major Research and Development Status 23
5.1 Framework 23
5.2 Dependability Metrics and Stakeholders' Agreement Processes 25
5.3 System Monitoring and Evidence Analysis 27
5.4 Security 30
5.5 Virtualization and its Application 32
5.6 Systems Software Verification 34
5.7 Dependability Measurement Tools and Test Tools 36
5.8 Process and International Standards 38
5.9 Research and Development Activity Results 40
6 Research and Development Organization 41
7 Roadmap 42
8 Further Issues for Practical Application 43
8.1 Handling of Intellectual Property Rights and Copyright 43
8.2 Open Systems Dependability Consortium 43
9 References 44
10 Appendix 45
10.1 Dependability Obstructions 45
10.2 Related Standards and Organizations 46

10.3 DEOS Project Terminology 48
10.4 DEOS Project Members 50

1 Background

Today's embedded systems such as mobile information equipment, office information equipment, home appliances, and car information systems are no longer used as independent stand-alone devices, but are connected to a network as part of a large system. Services are provided to these embedded systems through the network; making these devices useful to people all over the world, and bringing convenience and comfort into the lives of the people in societies where such systems are ubiquitous. Many of these embedded systems are increasing in complexity and scale in order to meet the diverse and sophisticated needs of their users. In the development of these systems, software, which has been used for some period of time and without detailed specifications, or which have been created by other developers, are used frequently being treated as a black box. Changes to external factors such as the modification of other systems that are connected via a network occur frequently. The maintenance and management of these systems involve specification changes while in use. Those factors of today's embedded systems make it impossible for developers and operators to know every detail of the system. As such, it is becoming more difficult to ensure the reliability and availability of these embedded systems. In addition, fatal system problems such as crashes caused by viruses, and information leaks due to unauthorized access continue to occur. It has become a major responsibility of product and service providers to take appropriate action against these threats, always ensuring user safety and security when using these products and services.

Recently many social infrastructures have experienced failures in their systems. These took considerable time to repair in some cases, and services were suspended until recovery, which caused considerable loss of benefits to users. This damage to service providers is not limited to the cost of analysis of the cause of the failure and work for recovery, but also includes the loss of business chances while the system is down and the loss of reputation with the public. The analysis of those failures shows that major causes of these failures include the system being developed without sufficient understanding of the behavior of all of the components, or the system being too complicated for every detail of the behavior of the system to be anticipated and controlled. Some system failures are attributable to insufficient planning by the designers and programmers. In some cases, the number of users, transactions, data volume, or coverage of the system exceeded the initial design limit, which led to failure of the systems. Changes or the addition of system functions after the launch of services to adapt to changes in the requirements of users caused the failure in some cases. These cases indicate that modern computer systems should not be assumed as a system whose function, architecture, and system boundary are fixed in both the way they are designed and implemented and the way they are used. They should be handled as systems which grow and change over time. It is expected that those characteristics will get stronger in the future.

Up until now, most development processes of embedded systems have adopted the common practice of creating a reliable development plan in advance, determining in detail the product or system specifications, and going through the long cycle of design, implementation and verification. It has become a standard practice to perform the "PDCA" cycle to enhance the special advantages (as well as function, performance, and quality) of individual products as well as the whole system. This process is quite effective for the development of products or systems which are not connected to a larger network and which have specifications and behavior that are quite defined and predictable at the beginning of development. However, as mentioned earlier, functions, structures, and boundaries of systems change over time, and the development and operation need to be performed under the condition that it will be nearly impossible to write complete specifications beforehand, envision a complete development plan, and correctly predict what all its network connections will be. For this kind of system, the management process for developing specifications based on a predicted range of conditions and for updating these specifications repeatedly by gathering feedback during the whole lifecycle is very crucial. Elemental technologies and system architecture to enable this process need to be developed [18]. The development process of a system suited to changing environments will not be the legacy "waterfall" model which requires completeness in each development phase, but process which adopts new software components and quickly implements the required functions in order to adjust to new requirements. The process of prototyping or trial production in actual environments, and improving the functions and quality of the systems with feedback from users, is demanded. In other words, we need to establish a new process in which both the feedback loops of the development process and the feedback loops of the operation and maintenance process move forward, while these dual loops interact with each other.

The above discussion indicates that dependability attainment technology for systems with fixed functions, structures, and boundaries is no longer sufficient to achieve and to improve a system's dependability. There is a need to establish a method to build and operate dependable systems that are based on new concepts and technologies [1, 4, 5, 6 and 12]. In this paper, the concept of "Open Systems Dependability" is proposed to meet those new requirements, and then the technologies, the architecture, and the processes to realize this concept are described.

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 and 19]. As a result of this discussion along with the increase in scale of hardware and software and with 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 and 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].

Changes in approaches taken to ensure reliability are reflected in international standards. International Safety Standards ISO 13849-1 (EN954-1) and Safety of Machinery - Electrical Equipment of Machines Standards IEC 60204-1 can handle simple systems, subsystems, and parts, but are not sufficient to deal with systems that include software. Functionality Safety Standards IEC 61508 were 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 software development, and 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 61058 as the base standard for software systems, machinery-related IEC 62061, process-related IEC 61511, nuclear-related IEC 61513,

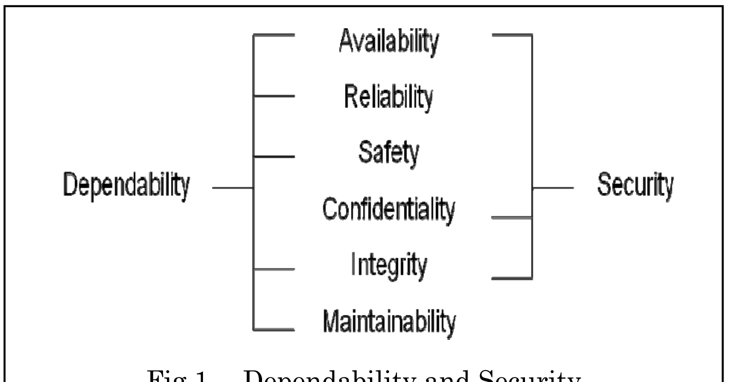


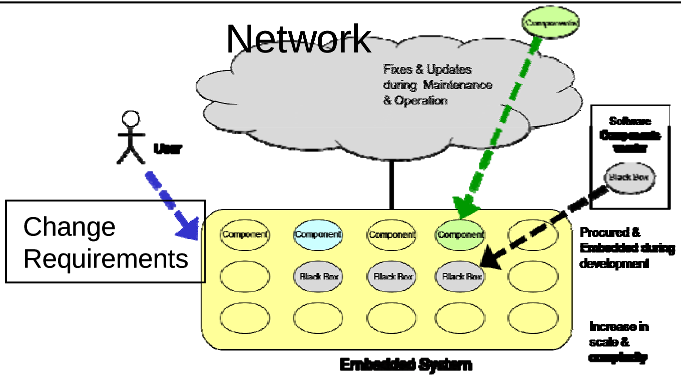
Fig.1. Dependability and Security

railway-related IEC 62278, etc. were established, and 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]. In this paper, dependability and security are defined using the terms given in Figure 1. However, in order to provide solutions to problems of complex modern systems with the functions, the structures, and the boundaries changing over time, assuming those factors are fixed and simply dividing and analyzing systems into these attributes and dealing with each attribute separately is insufficient.

2.2 Environment of Embedded Systems and Requirements

As it has already been described, embedded systems have become much more sophisticated and complex in order to meet the various needs of users; and they have grown larger in scale. The software architecture of embedded systems is determined, designed, and implemented based on requirement specifications. However, 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 for function improvement and change occur while the system is in operation. In this situation, amendments of the software are downloaded and new functions are added through the network. In this kind of environment, it is becoming exceedingly difficult for designers and developers to know each and every detail of the system throughout its lifecycle (Figure 2).


Network
Component
Fixes & Updates during Maintenance & Operation
User
Software Components vendor
Black Box
Change Requirements
Component
Component
Component
Component
Black Box
Black Box
Black Box
Procured & Embedded during development
Increase in scale & complexity
Embedded System
Fig.2. System Components and Services

Many of the modern embedded systems today are used with other systems to which they are connected via a network. In this case, users of the embedded systems utilize services that a single domain (consisting of networked systems) provide through the network. A single service domain also connects and interacts with other domains at different levels, and it is possible for services and the interface specifications of other service domains to undergo changes or to be discontinued. In this environment, the boundary of the system or service domain becomes unclear. Furthermore, there is the possibility that system designers and developers, operators and users may commit unintended errors. Likewise, with the spread of networks, unknown services are provided through the network expanding the possibility of unknown and unexpected interactions to occur; raising the concern that the system may be attacked on purpose

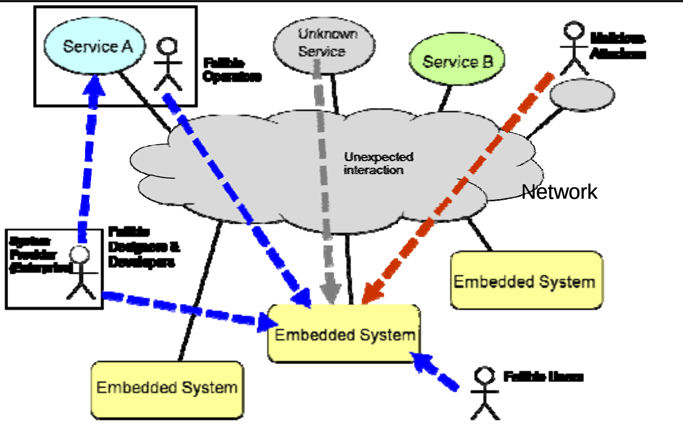
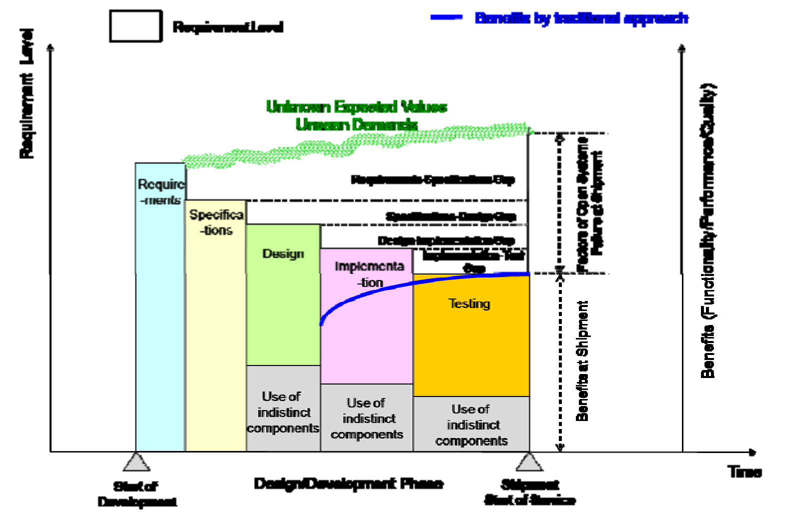


Fig.3. Services through Networks with Human Interactions

with malicious intent. For these reasons, the advent of networking has made predictability much more difficult to attain (Figure 3).

The analysis and classification of the cause of system failures discussed above, with consideration of the systems and services from both development and operation standpoints, will lead to the following factors.

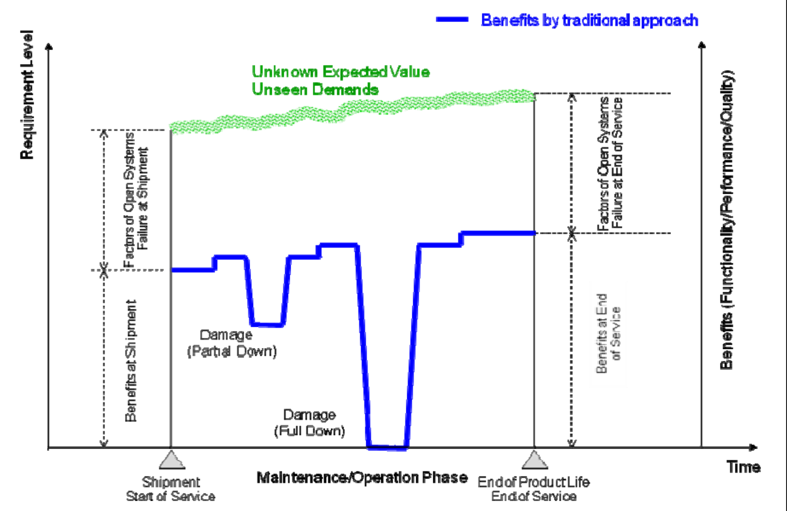
- **< Incomplete Specifications and Implementation, and Difficulty of Understanding the System>**
  It often happens that the initial requirement specifications become inadequate, the system's behavior becomes difficult to fully understand, and guarantees at the time of shipment cannot be made. (Fig.4). Specifically, there is likely to be:
  - An error or omission in the specification, design implementation, or testing, caused by discrepancies in characterizations of the system during the requirement development phase, specification phase, design phase, implementation phase, and testing phase, or by an error in the documentation



Requirement Level
Benefits by traditional approach
Requirement Level
Unknown Expected Values
Unseen Demands
Require-ments
Specifica-tions
Design
Implementa-tion
Testing
Requirements-Specifications Gap
Specifications-Design Gap
Design-Implementation Gap
Implementation-Test Gap
Use of indistinct components
Use of indistinct components
Use of indistinct components
Factors of Open Systems Failure at Shipment
Benefits at Shipment
Benefits (Functionality/Performance/Quality)
Start of Development
Design/Development Phase
Shipment Start of Service
Time

Fig.4. Incomplete Specifications/Implementation and the Difficulty of Understanding the System

  - 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
  - Poor system design, inadequate capacity of the planned system, demands upon programmers surpassing human capability
  - An error in administration, operation, or maintenance cased by the difficulty of understanding the whole system, particularly its software, due to its complexity and size
    - o An error in updating or amendment procedures
    - o Expiration of license
  - Use of "black box" components or legacy codes based on their external specifications, without knowledge of their internal design



Benefits by traditional approach
Requirement Level
Unknown Expected Value
Unseen Demands
Factors of Open Systems Failure at Shipment
Benefits at Shipment
Damage (Partial Down)
Damage (Full Down)
Factors of Open Systems Failure at End of Service
Benefits at End of Service
Benefits (Functionality/Performance/Quality)
Shipment Start of Service
Maintenance/Operation Phase
End of Product Life End of Service
Time

Fig.5. Uncertainty at Usage Environment, and Difficulty in Predicting System Behavior

- **< Uncertainty about Usage Environment, and Difficulty in Predicting System Behavior>**
  Changes in the usage environment and configuration throughout the lifecycle of the system make it difficult to completely predict the behavior of the system while still in the design phase. (Fig.5) Specifically, there are:
  - Changes in user's expectations

or capability during the maintenance and operation phases – changes in requirements and level of requirements, operation capability, or skill/experience/negligence of operators.

- Unexpected usage changes, such as those brought about by the significant increase in users, or number of units, as well as by changes in economy.
- Update or alteration of a component's function and system configuration on a system in operation through manual operation on through network.
- Specifications of services and components that are being used are corrected and updated via the network; unexpected network connections; and, intentional malicious attacks and intrusions through external entities.

Today's embedded systems with ever changing functions, structures, and boundaries must obtain dependability and be able to deal with the inherent ***incompleteness*** and ***uncertainty***. We cannot create a flawless system that can already handle all possible scenarios that could take place in the future. Failure, therefore, cannot be completely avoided.

Contemplation of this situation has led people to make various definitions of *"dependability"*. The following are examples of such definitions:

◆ *"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]

◆ *"The capacity for the services offered by the system to be maintained at a level acceptable for the user even if various accidents occur."* [17]

2.3 Open Systems Dependability

As we have discussed so far, we need to deal with the dependability of systems of which functions, structures, and boundaries keep changing over time. Systems with those characteristics are called Open Systems, in contrast to Closed Systems which assume fixed functions, structures, and boundaries that stay the same through the life of the systems.

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 (a whole system is dividable into subsystems or components, and the behavior of the whole system can be understood by understanding all of the subsystems or the components).

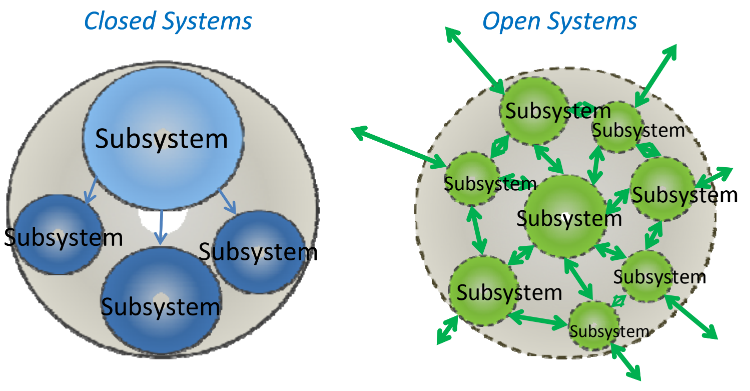


Fig. 6 . Closed Systems and Open Systems

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 hold the characteristics of opens systems. That is, those systems are connected to a network and the boundaries of the systems are unclear and change over time. Requirements for the functions and performance of these systems from stakeholders such as users of the services, providers of the services, and providers of the systems, etc., change, and the functions and performance of the systems change over time. Replacement or changing the components of the system occurs consistently for performance improvement, bug fixes, expiration of maintenance, and expiration of contracts, and the relationship of the subsystems or components of the system keep changing.

It may be possible to assume a system as a closed system at a specific time, which means there is no change for a certain period of time, and then consider the lifecycle of the system by accumulating 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 the specifications, repeating this process for each period of time in the lifecycle. However, it is extremely difficult to separate the phases that 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 on-going. To approach this situation, we should focus on the "ever-changing systems" and establish the concept of dependability focusing on the continuity of services and business by managing the continuously changing systems. Our approach is to consider the concerned systems as open systems and to focus on how we should improve dependability throughout the lifecycle of the systems. Based on our discussion thus far on the characteristics of modern embedded systems, we define **Open Systems Dependability** with the following description:

Functions, structures, and boundaries of modern embedded systems change over time. Hence incompleteness and uncertainty may result in failures in the future, and they are inherent to embedded systems (factors in open systems failure). Open Systems Dependability is the ability to continuously prevent the said 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 (this can largely be done during the manufacturer's development phase *before* shipment) and minimizing the damage due to open systems failures (this is largely done during the operation phase *after* shipment), concentrating on open systems failures resulting from *incompleteness* and *uncertainty*. Indeed, "Open Systems Dependability" complements and further enhances "closed systems dependability".

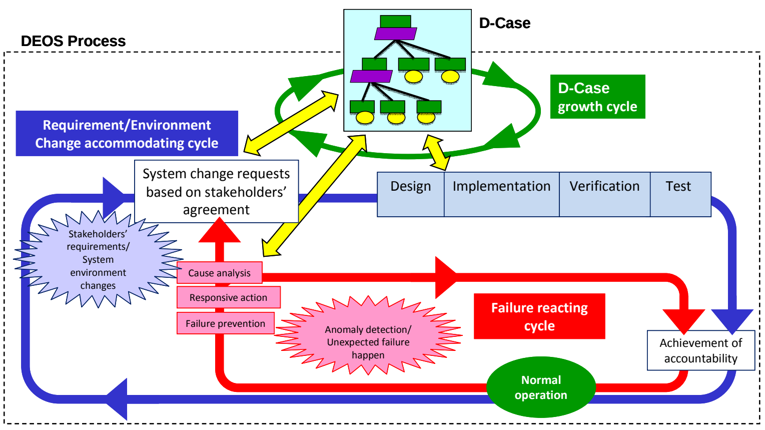
In summarizing the discussion in the previous paragraphs, Open Systems Dependability is defined as "providing services continuously by managing unpredictable failures on ever-changing systems". "Managing" means to assure ***sustainability of services*** with the best effort, which is most important to both users and providers of the services. The continuity of business which is assured as a result of continuity of service is also important for service providers.

2.4 DEOS Process

To achieve open systems dependability, that is to achieve continuity of services on the open systems in which functions, structures, and boundaries of the systems keep changing over time, we concluded that an approach from the process perspective is required. This is the process of the continuous improvement of dependability. We call this process the DEOS Process. We identify two cycles in this process; one is the "requirements/environment change accommodation cycle" which is a cycle to adapt the system according to requirements or environmental change, and the other is the "failure reacting cycle" which is a cycle to take immediate action and fix failures that occur in the system in service and operation. In summary a DEOS Process is a process to continuously improve the dependability of computer systems, and consists of 2 cycles (Fig.7), namely

1. Requirements/Environment Change Accommodation Cycle, and
2. Failure Reacting Cycle.

The Requirements/Environment Change Accommodating Cycle begins is triggered by the change of requirements of stakeholders or by a change to the system's environment. Examples of requirements changes are requests to change the content or the quality of services or to improve the function or the performance of the system, and requests to change the services or systems to meet changes in regulations or standards. On the other hand, examples of environmental changes are changes to network function or performance, changes raised by the change of services provided by other systems in the network, changes caused by hardware change, changes caused by the expiration of software licenses or maintenance contracts, and the changes required to cope with an increase in users. To accommodate those changes, the cycle, to get agreement of these change by the stakeholders, to go through design, implementation, verification, and testing, to describe the change and improvements to the users, and then to resume normal operation, begins.


DEOS Process
D-Case
D-Case growth cycle
Requirement/Environment Change accommodating cycle
System change requests based on stakeholders' agreement
Design
Implementation
Verification
Test
Stakeholders' requirements/ System environment changes
Cause analysis
Responsive action
Failure prevention
Anomaly detection/ Unexpected failure happen
Failure reacting cycle
Achievement of accountability
Normal operation
Fig.7. DEOS Process

The Failure Reacting Cycle requires prompt action. The cycle is triggered by failure prediction or by the occurrence of a fault or error, and in some cases is triggered by a failure. In the phase of failure prevention, responsive action, and cause analysis. Preventive action is taken if possible. In the case that a failure has occurred, an immediate fix must be taken. It is crucial to take accountability, that is an explanation to service and system users about the status of the failure, the immediate action taken, the plan for long term action and a permanent fix, and so on, and then the service and system go back to normal operation. The long term action and the permanent fix, with the result of root cause analysis, are agreed to be implemented by stakeholders. That is, this initiates the Requirements/Environment Change Accommodating Cycle. Normal operation includes preventive maintenance, daily improvement action (Kaizen), and preventive

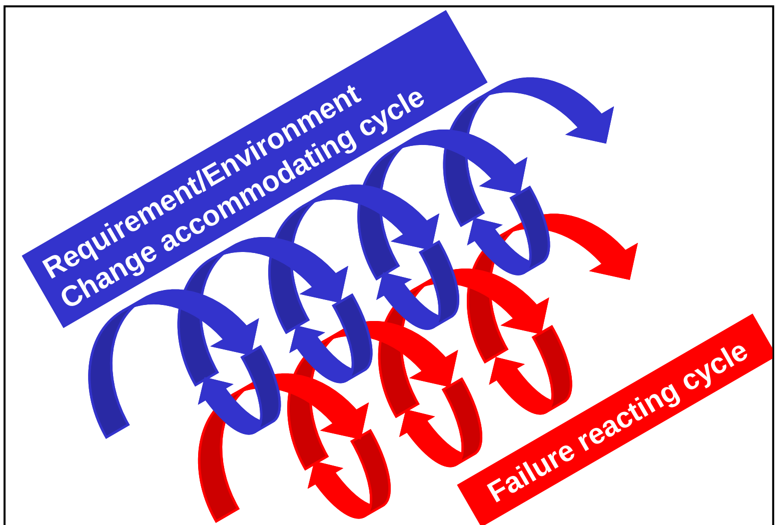


Fig.8. Image of the DEOS Process double helix in space-time

dry runs to be done before or after daily service.

The characteristics of the DEOS process are; 1)it consists of the two cycle, 2) it contains two phases, "system change-request based on stakeholders' agreement" and "achievement of accountability" . "D-Case" is used in the DEOS Process to support stakeholders to understand change-requests brought up among themselves, to examine and discuss each interest related to the change-request, and to reach agreement on the change to be made. As it has already been described, the DEOS Process implies that the system grows up over time, and the process forms into a double helix in space-time. Figure 8 shows the image of the double helix.

3 Phases of the DEOS Process

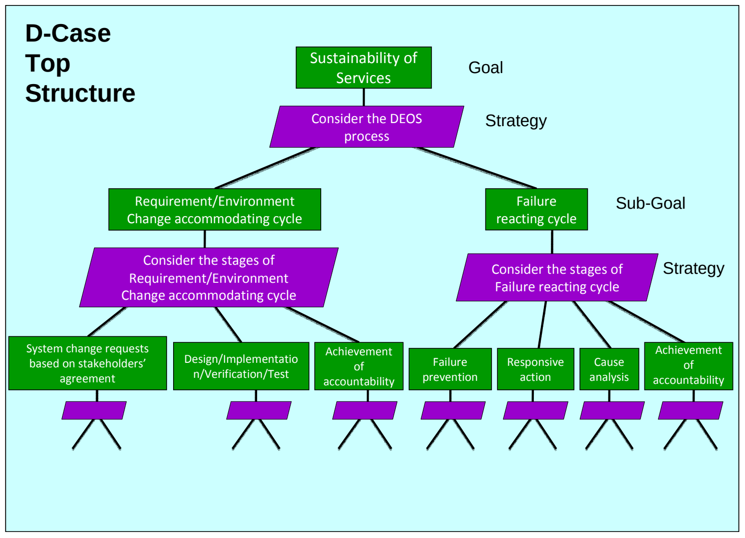
It has already been described that the DEOS Process is a process to continuously improve the dependability of open systems, and consists of the 2 cycles requirements/environment change accommodating cycle, and failure reacting cycle. The concept and required actions in each phase are discussed in this chapter. The phases in the "requirements/environment change accommodating cycle" are "system change-requests based on stakeholders' agreement" and "design, implementation, verification, and testing", and "achievement of accountability phase". The phases in the "failure reacting cycle" are "failure prevention", "responsive action", "cause analysis", and again "achievement of accountability phase". It is assumed that continuous dependability improvement action such as preventive maintenance, daily improvement action (Kaizen), preventive dry runs before or after the daily service, and education to engineers and operators are implemented in normal operation.

3.1 System Change-Requests based on Stakeholders' Agreement

When requirements from the stakeholders change (including new requirements), the environment of the system changes, or a change in the system is required as a result of cause analysis of a failure, changes are thoroughly described as written specifications without exception so that all the stakeholders understand the change without any misunderstanding. Requirement engineering addresses the method of describing the requirements and the tools to support the method, which should be fully utilized in this context.

Here the stakeholders, we assume in this white paper, are as listed.

- 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,
    - Operators and maintainers,
    - Providers of hardware, and
- Certifiers (Authorizers) of services or products.


D-Case Top Structure
Sustainability of Services
Goal
Consider the DEOS process
Strategy
Requirement/Environment Change accommodating cycle
Failure reacting cycle
Sub-Goal
Consider the stages of Requirement/Environment Change accommodating cycle
Consider the stages of Failure reacting cycle
Strategy
System change requests based on stakeholders' agreement
Design/Implementation/Verification/Test
Achievement of accountability
Failure prevention
Responsive action
Cause analysis
Achievement of accountability
Fig.9. D-Case Top Structure

In order to realize accommodation to change it is essential for all of the stakeholders to correctly understand the requirements,, to find solutions to possibly conflicting requirements, and to make decisions on how to implement and with what schedule. For this "description language or notation for stakeholders to share understanding" or "executable description language or notation for the agreement of requirements and specifications" is required to realize the process described. A set of the concept, method, and tool called D-Case is under study and development in the DEOS project. D-Case uses GSN (Goal Structured Notation), which is a description notation similar to Assurance Case. Assurance case is described in a tree structure. The tree structure itself is not substantial, but the strategy to decide how to branch from a node, the construction, or the condition of the branch, which are described as a DEOS process, is significant. The top structure of D-Case which describes Open Systems Dependability will be as shown in figure 9. The goal will be discussed and broken into sub-goals in the process of D-Case writing, and finally each node is connected to corresponding evidence. The bottom structure of D-Case will be as shown in figure 10.

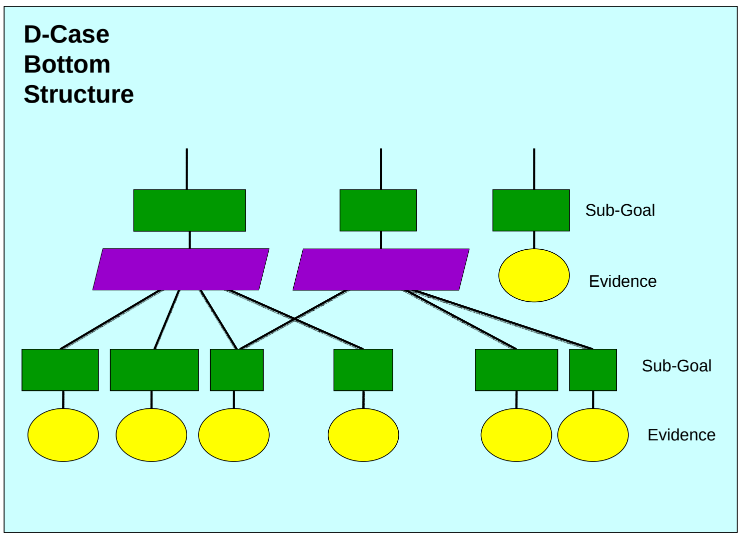


Fig.10. D-Case Bottom Structure

3.2 Design, Implementation, Verification, and Testing

This phase corresponds to the so-called design and development phase. There has been a lot of research and development in this area, and various methods and tools have been proposed. Useful methods and tools should be utilized in this phase for open systems as well. There are some specific requirements for this phase to improve open systems dependability.

- Design and implementation technology
  - Describe the requirements correctly and document as specifications
  - Design a system based on the specifications without mistakes
  - Implement the design without mistakes
- Verification and testing technology
  - Test the implemented system correctly according to the specifications
  - Detect software errors using formal method
  - Verify the system on a simulator
  - Measure system margin by injecting faults or abnormal conditions
  - Compare the system with expected normal status or known standard levels
  - Measure the dependability indices
  - Verify the validity of the dependability indices

3.3 Achievement of Accountability

Service providers, providers of social infrastructure, and commodity producers in particular have the responsibility to disclose and explain to service consumers, users of products, and society in general about occurrences and causes of failures, expected recovery time, and to amend their development and operation processes and procedures in routine operations so as to prevent such failures from occurring in the future. Carrying out these activities satisfies not only the social responsibility of the providers, but also creates the consensus required to run social infrastructure, and protect the brand of the service or system providers. It also helps to gain trust from service consumers and society,, and to protect their profits.

(1) Accountability in requirements/environment change accommodating cycle

- Describing changes of requirements
  - Investigation into the stakeholders' requirements
  - Processes and status of stakeholders' agreement
  - Content of services, system functions, schedule of service offerings, and terms and conditions of the offering, after the accommodation of the change
  - Status of compliance to laws and standards
- Describing change of environment
  - Investigation into the environment changes
  - Processes and status of stakeholders' agreement
  - Content of services, system functions, schedule of service offerings, and terms and conditions of the offering, after the accommodation of the change
  - Status of compliance to laws and standards

(2) Accountability in failure reacting cycle

- Describing status at failure prevention or at failure occurrence
  - Records of system monitoring and evidence of system behavior
  - Status of failure prevention
  - Status of responsive action, temporary fix, and service recovery schedule
  - Status of cause analysis
  - Schedule for root cause analysis and permanent fix implementation
  - Processes and status of stakeholders' agreement
  - Records of design and development, operation and maintenance, and education of engineers
  - Records of education of personnel related to the services
- Describing processes and status of daily business and system operation
  - Processes of reaching stakeholders' agreement and of maintenance of the agreement (D-Case to be described later)
  - Compliance with the DEOS Process
  - Education and processes related to personnel for system design, development, operation, and maintenance

Systems used to provide services need to have the capability to support the achievement of accountability as described above.

3.4 Failure Prevention

(1) Prediction of failures and detection of anomalies

We will be able to establish a procedure to prevent failures if we can predict them while systems are in operation by detecting some anomalies in advance which are potential causes of failure. The action taken will depend upon how soon we can predict the failure. The earlier we can predict it, the more valuable the prediction is for the system or service providers. For instance, we may be able to prevent the failure if we can predict it a few minutes before. Even if we cannot prevent it from occurring, we may be able to minimize any damage caused by the failure. If we predict a failure a few seconds before its occurrence, we may be able to prevent the entire system from shutting down, even though it may be impossible to prevent partial failure. Alternatively, if we can predict a failure a few milliseconds in advance, we may be able to keep the records of operations and system status changes which will be useful for the analysis of the root cause of the failure later, even if we cannot prevent its occurrence or system shutdown.

The operations and status of the system before a failure could be recorded on a device like a flight recorder, which keeps record of flight data before an aircraft accident by continuously writing the latest flight data. However, if the system predicts a failure in advance, the system will have more time to select the relevant data and to record only that data for a longer period. Therefore, we consider that methods to predict failures and to detect anomalies are critical for system dependability. Such methods would involve:

- Detecting anomalies which caused similar failures
  - Infer which anomalies are relevant from patterns of failures in the past

- Detecting signs of anomalies
  - Predict anomalies
  - Detect signs which may imply failures in some cases
- Rehearsals (to be described in section 3.7)

(2) Restricting/Limiting system resources to prevent failures

We may manage to avoid system crashes or to take some action by postponing an occurrence of a system crash by restricting system resources when an anomaly which suggests a system crash is detected.

- Restrict behavior of a component which shows an anomaly so that the component may not cause a failure

3.5 Responsive Action

Failures are inevitable in open systems environments. Although predictions and rehearsals are beneficial, they cannot be carried out in all situations. For situations where they cannot be applied, it is crucial to take some other action to minimize damage after a failure. The action required after the occurrence of a failure are described below. There are two goals for action after a failure: One is to maintain or regain the trust of service consumers by minimizing their dissatisfaction and inconvenience, and the other is to maintain the business profits of the service providers.

It is expected the responsive action is taken automatically and autonomically while systems are in operation. However usually this is not the case, and in most cases the operation of whole systems or a part of the systems are suspended temporarily, stakeholders make some decisions, and maintainers or operators take some action with human intervention. It is preferable that the period of suspended time is minimized. To manage this situation, usually twofold action is planned; temporary fix or solution, and long term or permanent fix or solution with root cause analysis.

The action thus must:

- Minimize the damage and prevent a service shutdown by isolating failures
  - Temporarily halt the failed service, or return the service to a safe operational mode
  - Maintain services other than the failed service
  - Minimize inconvenience to consumers due to the failed service

- Quickly uncover the root cause of the failure, repair the failure, and effect recovery of the system
  - Discover the root cause of the failure, and determine the repair which will prevent the same failure from occurring in the future
  - Minimize the insecurity of service consumers
  - Maintain and even improve mutual trust

3.6 Cause Analysis

It is always expected to pinpoint the cause of failure, and it is difficult in many cases. Even if the cause of failure is not pinpointed, narrowing down the area of the cause of failure will reduce the cost and time for the cause analysis significantly, and as a result, it will help to shorten the period of system down time and to find the root cause and long term solution. The technologies required to support this need to be investigated and developed. Accurate recording of system behavior or appropriate recording of evidence for cause analysis play a key role.

There has been significant research and development in this area in the past and this is on-going. The technology in this area is also covered in the DEOS project with a new concept and methodology. The focus areas in this project are;

- To observe system behavior and keep records (logs)
- To investigate and develop the technology to manage large amount of records and to keep only required log without bankruptcy.
- To narrow down the area of the cause of failure in case pinpointing of the cause of failure is not achievable
- To record the behavior of the system just before the system goes into system crash (System Recorder)

3.7 Normal Operations

Keeping records of the systems' operation and inspecting the records periodically will help to maintain the dependability of the system. The operators or maintainers may find the symptoms of system failure from this activity. Memory leak may cause significant system failure at some point of operation. Keeping the system memory at clean status is an effective preventive action.

- Keep records of system operation
- Prevent system aging

Proactive rehearsal to simulate future systems helps to predict future failure. A failure may occur when a system reaches a certain state and then runs past a certain date and time. In this case we can see whether a failure may occur by advancing the system date and time. This is known as a rehearsal. The rehearsal will be effective for failure prevention described in section 3.4. Appropriate rehearsal operations of an actual computer system while the system is in operation are determined in accordance with the computer system itself and the operation environment. (During the development phase, similar activities are frequently performed, and they are called "testing.") If the system operation is suspended every night, some rehearsals can be done while the system is suspended.

However, if the computer system is available for use 24 hours a day and 365 days a year, some rehearsals can be performed only upon delivery, before it starts service to its customers. If we subsequently want to perform any rehearsals, we will need a mirror system.

- Rehearsals may be performed before the service becomes available to customers (before the initial release of the system, or before the system starts service each day)
- Rehearsals may be performed on a mirror system.

4 Project Direction

4.1 Project Goal

In this project, "Dependable Operating Systems for Embedded Systems Aiming at Practical Applications", the requirements for today's embedded systems such as reliability, security, and usability, are derived from the concept of "Open Systems Dependability". A dependable OS for embedded systems, as well as the concepts, architecture, specifications/implementation guidelines, management process, framework, development environment and tools and other related platform technologies necessary for various types of practical implementation are developed. Evaluation criteria are set, clarified, and standardized. In this project, "OS" is not defined as "operating system" in the strict sense of the word. "OS" here has a broader meaning, including all system software layers supporting the system applications. Moreover, the embedded systems we have in mind are defined so as to include systems used as social infrastructure and connected to a network, such as traffic information control systems and railway ticket systems.

Note that, at present, since systems for monitoring, production control, communication control, office information equipment, vehicle information equipment, robots, information electronic devices, mobile phones, mobile information terminals, and others can be regarded as applications of the OS we seek, further narrowing of our range of research according to real users' needs should be considered.

4.2 Project Objectives

The following are the objectives of this project:

1 Establish a clear concept of dependability appropriate for the 21st century.
   - 1.a Evaluate and refine the concept of Open Systems Dependability that was discussed in Chapter 2.
   - 1.b Develop elemental technologies, management processes, and system architecture that will enhance Open Systems Dependability throughout a service and system's lifecycle.

2 Promote practical applications.
   Create specification and implementation guidelines for a dependable embedded system that is suited for actual practical uses; develop frameworks, a development environment, and tools; establish a management process; and construct a demonstration system for evaluation.

3 In accordance with 1 and 2 above, set an evaluation standard for dependability, and promote its standardization and clarification (to be suitable as an international standard.)

4 Start a consortium or user organization for the utilization, maintenance, and enhancement of 1 to 3 above.

The elemental technologies, processes, management practices, and system architecture required to build a dependable system will be evaluated and enhanced throughout the development of practical systems in a continuous improvement cycle. The deliverables of the project are expected to be used by industry in the provision of products and services, and subsequently improved through the feedback received regarding their practical usage. Furthermore, it will be essential in the future to establish a society-wide open structure to support the sharing of system failure information, and to carry out social responsibilities such as indemnity and accountability.

4.3 Realizing Open Systems Dependability

There are 3 domains in the implementation structure that need to be integrated in a well-organized way to properly manage Open Systems Dependability: *elemental technologies, management processes, and system architecture* [13].

This project conceptualizes, does research for, and develops a fundamental, well-organized structure. We intend to make our results widely utilizable by embedded system and service providers (Fig. 11). The research teams from 9 research organizations are working as one to achieve this objective.

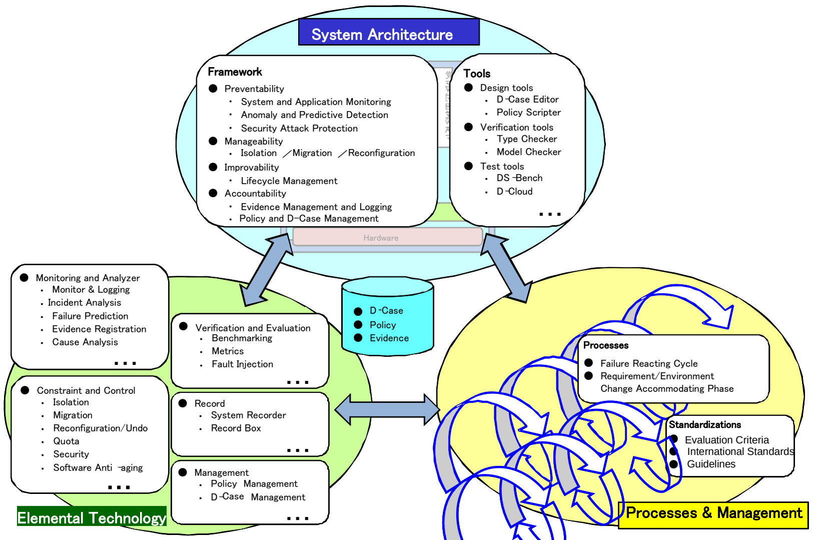


Fig.11. Realizing Open Systems Dependability

(1) Systems Architecture

The system architecture should have the following features in order to integrate the elemental technologies and the management processes effectively:

- Schemes for implementing elemental technologies
- Schemes for supporting the management processes (cycle of planning, execution, monitoring, and analysis)
- Schemes supporting system improvement, and enabling changes while the system is in operation.

To achieve and improve Open Systems Dependability, we need to construct a set of system software including middleware and various tools used in development phases and operational phases. In this project, we realize the set of software by the framework and tools described below.

Framework

We call our implementation of the framework "D-fops", which stands for Dependability Framework for Open Systems. D-fops is intended to incorporate our research accomplishments into an integrated software package that companies can use to evaluate the usefulness of the concept of Open Systems Dependability and of dependability-related technologies in their products or systems. Research integrated into D-fops includes "monitoring and analyzing", "virtualization technologies and their applications" and "security," which are described later in this document.

If some companies are interested in using the code in an experimental environment, the Dependable Embedded Operating System Research and Development Center (DEOS R&D Center) will provide a system for evaluation of the software together with them. D-fops will be a reference implementation example, and a user may customize it for use in their commercial products or services.

Tools

A set of tools called DEOS tools will be developed which incorporates the research accomplishments regarding dependability metrics and agreements, policy management, system software verification, and dependability metrics measurements and evaluation. Although some tools may be used alone, we intend that most of the tools will be effectively utilized together with existing tools for design, verification, testing, operation, and management. We plan to evaluate the usefulness of the concept of the Open Systems Dependability and of our framework and tools in actual development and operation.

(2) Elemental Technologies

The following table shows elemental technologies which we think at present are mandatory to achieve Open Systems Dependability, and how they contribute to each phase of DEOS processes described in Section 2.4.

		System change requests based on stakeholders' agreement	Design/Implementation/ Verification/Testing	Achievement of accountability	Preventability	Responsive action	Cause analysis	Normal operation	Related sections
Monitoring & Analyzing	System monitoring and logging (monitoring & logging)	✔		✔	✔	✔	✔	✔	5.3. 5.5.
	Event analysis and verification (incident analysis)	✔		✔	✔	✔	✔	✔	5.3.
	Predictive detection (failure prediction)				✔				5.3.
	Cause analysis	✔		✔	✔	✔	✔	✔	5.3. 5.5.
Resource Allocation & Control	System resource control and behavior control (quota)				✔	✔			5.1.
	Software aging prevention				✔			✔	5.5.
	Security attack protection throughout network (security)				✔	✔		✔	5.4. 5.5.
	Building system containers (isolation)			✔	✔	✔			5.5.
	Verification of virtualization layer		✔						5.5.
	Testing with time-shift (time-shift rehearsal)				✔			✔	5.1.
	Isolation, reconfiguration and restoration of failed part (isolation/reconfiguration/removal)					✔			5.1.
	Subsystem removal and repair (migration)					✔			5.1.

Recording	Recording at the last minute before system goes down (system recorder)	✔		✔		✔			5.1.
	Consistent timing system and protection against fake data (record box)	✔		✔	✔	✔		✔	5.1.
Verification & evaluation	Model checking for systems software		✔		✔				5.6.
	Type checking for systems software		✔		✔				5.6.
	Dependability evaluation metrics (metrics)	✔		✔	✔			✔	5.2.
	Measurement of system load, anomalies, and behavior			✔	✔			✔	5.7.
	Speeding up a large scale system test by resource management				✔	✔			5.7.
Management	Dependability consensus description, update and management (dependability case management)	✔		✔	✔	✔		✔	5.2.
	Fail safe mechanism for unknown failure			✔		✔			5.1.
	Policy script and management	✔		✔	✔	✔		✔	5.2. 5.3.

Table 1. Elemental technologies and how they contribute to the four characteristics

Note: The five categories in the leftmost column are for reference purposes. The usage of each technology is not limited to these categories.

(3) Processes & Management

As we discussed in the previous chapter, making open system services continuously available requires preventing system failure in advance, responding to unexpected failures quickly, and improving the system appropriately to accommodate changes in the stakeholders' requirements and the systems environment. In addition to these, the actions to system failures or the activities for the system improvement need to be described to stakeholders. As it has been discussed in Chapters 2 and 3, we propose a DEOS Process consisting of a "requirements/environment change accommodation cycle" and a "failure reacting cycle" to achieve this.

The current V-model and agile software development processes focus on the system development phase. The DEOS Process is different from these conventional models in that it focuses on the operation and maintenance cycle. We consider that the system development phase covered in conventional software development processes is a very early, albeit distinctive, part of the operation and maintenance cycle in that it also is carried out incrementally according to changes in stakeholders' requirements and the system environment. Also, reacting to unexpected failures appropriately is important for open systems. We assumed two cycles based on the above consideration; both cycles related to each other. It is required to improve the system to provide long term solutions to react to failures in the "failure reacting cycle", which leads to the "requirements/environment change accommodation cycle" . Besides, improvements to the system through the "requirements/environment change accommodation cycle" become a potential cause of future failure, which leads to the "failure reacting cycle". These two cycles move forward interacting with each other and in parallel throughout the lifecycle of one system. Figure 12 shows how the elemental technologies developed in the DEOS project are used in the DEOS process.

- Requirements/Environment change accommodating cycle
  - Partial improvements to the system are performed to accommodate change. Verification of the modified part is done using Type/Model Checker which verifies the software by mathematical methods, and the result is used as evidence of the correctness of the software.
  - The system with partial improvements is simulated by the large scale computing and simulation capability of DS-Bench/D-Cloud, and the result is used as evidence that the partial improvement does not cause faults.
  - The improved part of the system is released for use.

- Failure reacting cycle
  - Detection and analysis of failures is performed using D-Logger and D-Analyzer.
  - Reactive action to the failure is performed using D-Visor and D-Effector.
  - Reconfiguration of the system to react to the failure is performed.
  - Log data of the failure is recorded in D-Box as an action of future failure prevention.

Throughout this process, D-Case is used to get to agreement from stakeholders by describing the action to the failure in open systems and the improvement of the system, as well as accurate records of system behavior and appropriate evidence. D-Case documents are updated whenever a system is changed with the reconfiguration function or code enhancements, and may be used to explain to stakeholders the latest dependability status of the system at any time.

The elementary technologies mentioned here will be described in detail in Chapter 5.

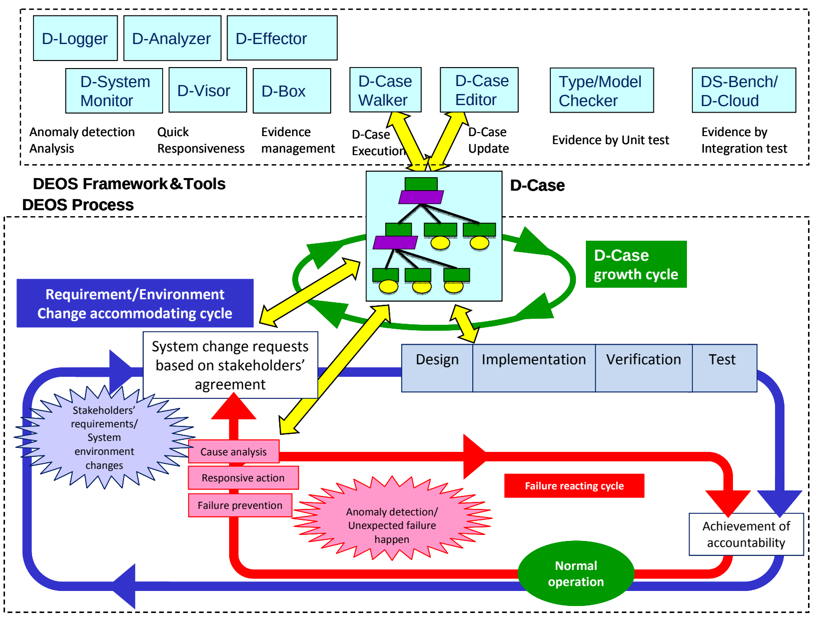


Fig. 12. Systems Development & Operation using DEOS Technologies

4.4 Project Deliverables

The following are the expected deliverables of this project:

- Open systems dependability concept
- Elemental technologies（specification for each technology, API reference document, code, implementation guidelines, etc)
- Process（process guidelines, etc)
- Framework, development environment and tools (system architecture specifications, API reference document, code, implementation guidelines, etc)
- Metrics, standard guidelines and record format
- Open systems dependability consortium

5 Major Research and Development Status

5.1 Framework

Objectives

D-fops, the Dependability Framework for Open Systems, is an integration of the elemental technologies of the DEOS project (such as 5.2: Dependability Case and Stakeholder Agreement Processes, 5.3: System monitoring and Evidence Analysis, 5.4: Security and 5.5: Virtualization and its Application). D-fops implements the following functions:

	System change requests based on stakeholders' agreement	Design/Implementation/ Verification/Testing	Achievement of accountability	Preventability	Responsive action	Cause analysis	Normal operation
Recording at the last minute before system goes down (system recorder)	✔		✔		✔		
Consistent timing system and protection against fake data (record box)	✔	✔	✔	✔	✔		✔
System resource control and behavior control (quota)				✔	✔		
Fail safe mechanism for unknown failure			✔		✔		
Software aging prevention				✔			✔
Testing with time-shift (time-shift rehearsal)				✔			✔
Isolation, reconfiguration and restoration of failed part (isolation/reconfiguration/removal)					✔		
Subsystem removal and repair (migration)					✔		

Strategies

D-fops is under development to achieve the objectives mentioned above. It is designed to support the DEOS process (shown in 2.4). It consists of a set of system features to sustain the dependability of services desired by stakeholders. It is primarily for embedded systems, but is also applicable to other systems including computer systems for social infrastructure. The functions of each service component of D-fops are described below.

D-Visor

This provides a mechanism for securely isolating two or more subsystems, each of which is run in an isolated partition called a System Container. System Containers are independent from each other, preventing anomaly/failure in one container from affecting the other containers.

D-Application Manager

This provides a mechanism for securely isolating two or more applications, each of which is run in an isolated named space called an Application Container. In addition, it controls the lifecycle of each application program (invoking, revising, and terminating the program) and provides a mechanism for a system designer to develop a system which can monitor/analyze the behavior of application programs and take appropriate action. It also provides a control API that allows improvements in Open Systems Dependability. An application program which is implemented carefully to improve dependability using the API is called a D-Aware Application. The D-Application Manager provides basic monitoring services to other existing application programs, so-called legacy applications, and enables appropriate action upon these applications to be taken.

D-Logger/D-Analyzer/D-Effector

D-Logger monitors the behavior of applications. D-Analyzer analyzes the monitoring results. D-Effector takes necessary action. D-Analyzer also has functions to select the data which is necessary as evidence and to compress this data. D-Box and D-Analyzer cooperate with each other to record evidence.

D-Case Walker

This takes action dictated by system action policies and the agreed-upon definition of dependability described in D-Case, which will be discussed later in section 5.2, in order to improve Open Systems Dependability.

D-System Monitor

This independently monitors the system behavior from outside the OS. It also ensures system integrity by monitoring system failures and attacks from the outside.

D-Box

This records useful information such as D-Case, policy, and evidence useful to improve Open Systems Dependability. It guards information integrity using access control, encryption, and manipulation detection.

Deliverables

- System architecture specification
- API definition
- D-Analyzer, D-Case Walker, D-Logger and D-Box program codes, etc
- Implementation Guideline for Embedded systems, etc

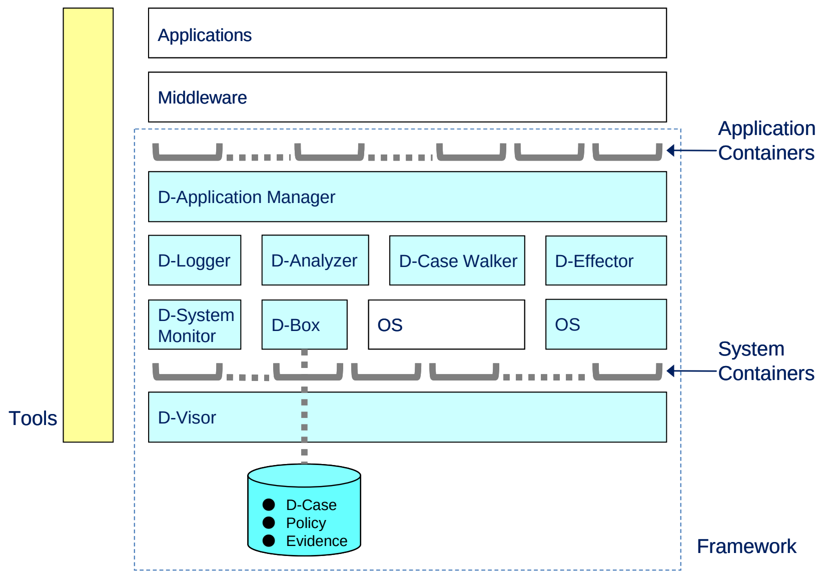


Fig.13. Framework System Configuration

5.2 Dependability Metrics and Stakeholders' Agreement Processes

	System change requests based on stakeholders' agreement	Design/Implementation/ Verification/Testing	Achievement of accountability	Preventability	Responsive action	Cause analysis	Normal operation
Dependability evaluation metrics（metrics）	✔		✔	✔			✔
Dependability consensus description, update and management （dependability case management）	✔		✔	✔	✔		✔
Policy script and policy management	✔		✔	✔	✔		✔

Objectives

The diversity and changeability of open systems make it inherently challenging for stakeholders to obtain agreements and assurances on the dependability of systems.　Furthermore, in an open environment, stakeholders must maintain the dependability of both the whole system and all subsystems whose boundaries are ambiguous and dynamic. Meeting these challenges requires support for the agreement-making process and evidence-based accountability for the correct implementation of agreements. Also, a mechanism for sharing dependability information through system interfaces is required, for various system relations, e.g., the inclusion relation of an embedded system and embedded OS components and the correspondence relation of servers and clients in network systems.

To satisfy these requirements, first, we need a dependability modeling language for describing the dependability requirements in a form understandable to diverse stakeholders.　Second, we need a dependability metrics as a base for mutual agreement among stakeholders. Third, a process should be in place that ensures traceability between the dependability descriptions and actual system behaviors.　The process not only tracks the development phases of a system but also monitors the system during operation, constantly checking that dependability requirements are being satisfied. Furthermore, in open systems, information of dependability requirements must be shared among not only stakeholders, but also among stakeholders and systems. Therefore, we need to develop a translation mechanism from dependability requirements written in the modeling language into codes readable by systems; interfaces for sharing dependability requirement information among stakeholders and systems.

Thus, we propose the "D-Case method" with the following goals.　As explained later, a D-Case is an assurance case for dependability.

1. D-Case Language, a dependability modeling language to facilitate dependability agreements and evidence-based accountability and a translation mechanism from D-Case Language into codes readable by systems
2. Dependability Metrics for open systems to evaluate and ascertain described and achieved dependability.
3. D-Case Tools for D-Case document creation, consistency checking, verification, presentation for agreements and accountability, maintenance, and interoperability with existing tools.
4. D-Case Process, system life-cycle processes that define and coordinate D-Case activities / tool use, at each life-cycle phase, in relation to existing ones.

Strategies

For these goals, we set the following directions.

1. D-Case language

We designed the D-Case Language, a dependability modeling language to describe agreements among stakeholders on dependability and to explain how it is assured based on evidence. We adopted as a starting point the Goal Structuring Notation (GSN) developed by Tim Kelly and his colleagues at the University of York, one of two prominent graphical notations for arguments in assurance cases. Arguments in GSN are structured as trees with a few kinds of nodes, including: ``goal'' nodes for claims that arguments are meant to substantiate, ``strategy'' nodes for reasoning steps that decompose a goal into subgoals, and ``evidence'' nodes for references to direct evidence that respective goals hold. D-Case Language extends GSN in several ways. For example, we add ``monitoring'' nodes to denote operation-phase evidence supplied by monitoring mechanisms as well as evaluation functions based on dependability metrics. We call a document written in D-Case Language a D-Case document.

A novelty of D-Case is that in a D-Case document, we argue dependability of systems by precisely following DEOS process. To achieve Open Systems Dependability, stakeholders must make agreements on future systems changes, using their knowledge and wisdom to their best effort. To that end, not only the information of the structure of the target system and the subsystems (i.e., a model of a relation of the whole system and the subsystems), which is often used in safety cases, but we also describe how the target system adheres to each phases of the DEOS process. In this way, we need to establish a methodology for describing both the structure of systems and processes in a uniform way. D-Case modeling language, as the language for dependability description must be elaborated to satisfy this requirement.

2. Dependability Metrics

To express differing degrees of dependability requirements for systems with different purposes, quantitative measures of dependability are necessary. It is also essential for meaningful negotiations among stakeholders towards mutual agreements on dependability. So we develop dependability metrics for the quantitative evaluation of systems dependability.

In the development phase, customers and developers may use the metrics to argue how much of a weighted combination of dependability requirements can be realized at what cost, etc. In the operation phase, operators coping with failures may be helped by real-time measurements of dependability in devising recovery strategies etc.

D-Case documents are crucial in considering dependability metrics since it is they that clarify what to measure and what measured values mean to stakeholders. Weights can be assigned to goals and measurements on them can be integrated into a quantitative evaluation depending on how each goal is supported by sub goals and evidence. The live link between D-Cases and systems, explained below, enables timely evaluation on live systems based on operation-phase evidence.

3. D-Case Tools

The following are being developed: D-Case Editor to edit/verify D-Case documents, D-Case Viewer to display/monitor dynamically changing live D-Cases for systems at the operation phase, and a Dependability Metrics Visualization Tool to provide visual, quantitative evaluation functions for D-Case Editor and Viewer.

4. D-Case Process

We provide D-Case process for dependability agreements and assurance based on evidence. In the DEOS process, according to system failure and requirements and environment changes, D-Case documents are updated, and gradually corrected and refined. We call the process for maintaining D-Case documents as the D-Case process. Also, to facilitate writing D-Case documents, we design D-Case patterns for each system domain.

A case of using these tools is as follows. At each phase of a system life cycle, D-Case Editor is used to record as D-Case documents, explain, and verify various negotiations and agreements among stakeholders on dependability. Agreements are made according to the D-Case Process for that phase, and, in the process, evaluation scores are given by Dependability Metrics Visualization Tool to provide an objective basis for negotiation. At the operation phase, the D-Case is relayed to the D-Case Viewer. Dynamic display/monitoring of D-Case together with runtime evidence and D-fops action from systems/environments helps operators/maintainers/users to recover from and improve upon unexpected failures. D-Cases need to change as all systems change during their lifetime due to changing stakeholder requirements, reconfiguration, etc. D-Case methodology aims to contribute to Open System Dependability by managing/maintaining stakeholder agreements, evidence-based assurances, and accountability in a synchronized manner with changes in live systems/environments (Fig. 14).

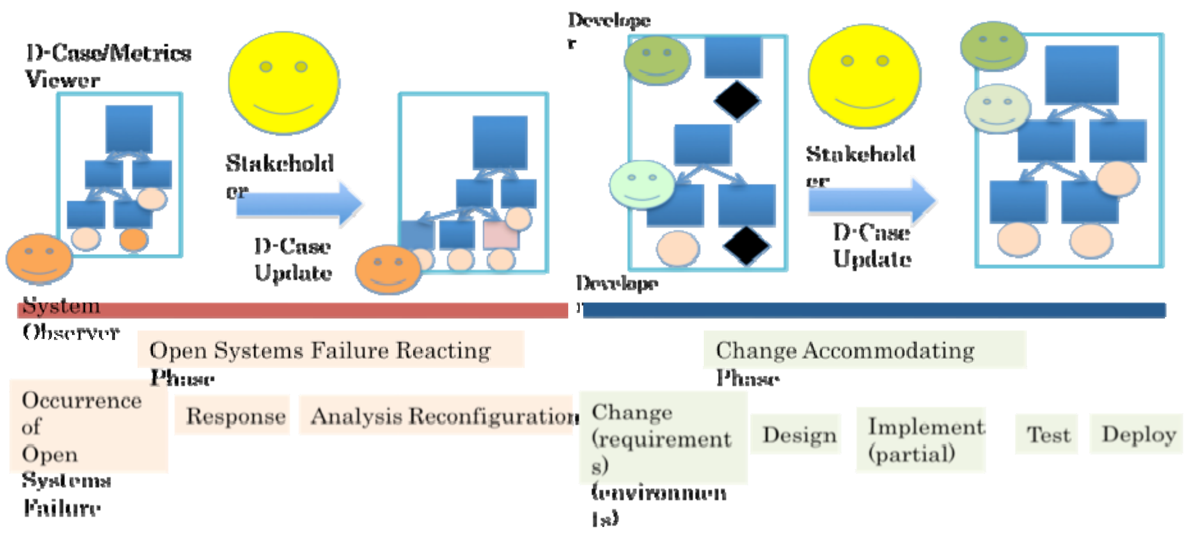


Fig. 14. D-Case Process

Deliverables

- D-Case modeling language specification
- Dependability metrics
- D-Case Tools : D-Case Editor、D-Case Verifier、D-Case Viewer、and Dependability Metrics Visualization Tool.
- D-Case process
    - D-Case Pattern

5.3 System Monitoring and Evidence Analysis

		System change requests based on stakeholders' agreement	Design/Implementation/ Verification/Testing	Achievement of accountability	Preventability	Responsive action	Cause analysis	Normal operation
Monitoring & Analyzing	System monitoring and logging (monitoring & logging)	✔		✔	✔	✔	✔	✔
	Event analysis and verification (incident analysis)	✔		✔	✔	✔	✔	✔

	Predictive detection (failure prediction)				✔			
	Cause analysis	✔		✔	✔	✔	✔	✔

Objectives
Computer systems have always faced a variety of changes. Some of the changes will be the cause of unexpected failure. We focus primarily on system monitoring and log analysis for tracking changes occurring in the operation phase. The system monitored changes are recorded and logged to support a variety of types of dependability management, such as analyzing the logged events for evaluation, proactive failure prediction, and root cause analysis, which constitutes a main part the foundation of open systems dependability.

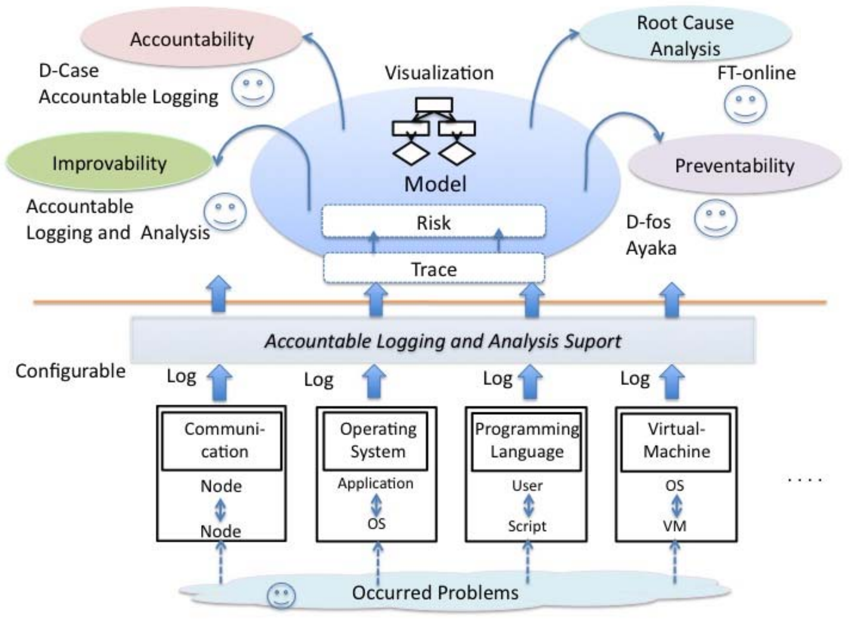


Fig. 15. Conceptual Overview of System Monitoring and Log Analysis

Figure 15 depicts an overview of our concept model. A log is recorded at various levels of systems, such as communication system, operating system, language runtime, and VMM (Virtual Machine Monitor). These logs are analyzed and stored as evidence for indicating the actual behavior of systems at failure, and for root cause analysis. We have investigated and developed D-Logger and D-Analyzer working on D-fops, and will provide them as online log analysis mechanisms. The evidence evaluation based on log analysis will be used for the prediction of failures and for root cause analysis. We will provide a tool for giving a view for D-Analyzers that will be the part of the root cause analysis for engineers.

Our ultimate goal is to deliver accountability by conducting risk assessments that are based on the evidence for making profiling of a log of the current state of a system, and will support providing procedures including failure avoidance and executing accountability and daily checking and improvement and root cause analysis for failures for achieving open system dependability.
We will take a twofold approach for achieving these purposes.

(1) Evidence Management
A variety of logging records come from operating systems, communication nodes and individual application programs. Each of these records can be evidence for a certain aspect of the whole. The integration of these logs is needed and would be useful for root cause analysis, failure avoidance,

executing accountability and quick action for failures. This would help in failure recovery and continuous improvement. We address this problem with a newly defined model of evidence management, and developed for open systems dependability.

(2) Accountable Logging and Analysis
Logging and analysis has been widely used in computer systems. However, usually in those systems, a very small fraction of internal status information is recorded, and this is not necessarily sufficient for assuring the verification of the correctness of system behavior. Thus, we have developed a new logging method that is to collecting evidence for accountability.

Strategies
We have addressed the creation of accountability logging, analysis and evidence management with the following research direction strategies.

(1) Accountable Real-time Operating System.
As the complexity of real-time systems increase, the model-theoretic and predictable design of a theoretical model of real-time tasks for real-time prediction purposes becomes impractical and difficult. We only model to determine in real-time the correctness of the execution of real-time tasks that can be inspected based on log analysis of available logged data. This research direction includes the need to deal with the overhead problem of real-time logging and the implementation issues of real-time on the Linux model.

(2) Accountable Programming Language.
We have focused on constructing a function for the automated generation of application logging that allows us to inspect and evaluate the correctness of programs. The correctness of programs is judged based on the specifications of goal-oriented software requirements, and this judgment is entered as an annotation in the source code. Runtime errors are inspected through stack analysis and the analysis of other internal information. To implement these ideas, we have extended our implemented scripting language, named Konoha, in terms of both the language grammar and its scripting engine.

(3) Traceability of Log Analysis Chain D-Analyzer
Anomaly detection is a crucial part of the evidence evaluation process. That is, anomalies refer to unspecified incorrect behavior, whereas some levels of correctness are specified in the accountable logging commitment. Rather than the anomaly specification, researchers apply modeling techniques that enable them to detect anomalies in accordance with changing situations in open systems.

Evidence consists of multiple independently observed events, and the time and place of these events must be traceable in terms of the time and place of their occurrence to verify, and the validness of the log analysis. We address the traceability problem by organizing a variety of loggers and an analyzer, using domain-specific language-based management.

(4) Stream Evidence Engine (SEE)
D-Analyzers are connected with a traceable log analysis engine named the Stream Evidence Engine (SEE) that includes a domain specific language for evidence analysis, and a streaming evidence analysis engine. D-Analyzer can use the integrated interfaces of streams through SEE. SEE will also provide the load-balancing mechanism to distribute its load of analyzing, and it sends the results for the evidence server through their interfaces.

(5) Evidence Viewer Model
The collected logs for use of evidence range from accountability establishment (i.e., root cause analysis) to policy change (i.e., risk analysis). A unified dependency viewer is needed to improve the usability of observed events as evidence. This research must include the visualization of this unified model for a human operator and the system integration of automated management into D-fops. In particular, we will present an online fault analysis viewer that views various aspects of system

behavior in terms of dependable risk analysis (a FTA extension), and event traceability (root cause analysis).

Case Study in Robotics
As a case study of a highly advanced embedded system, we are planning to apply our developed techniques to robots that have been developed at AIST (National Institute of Advanced Industrial Science and Technology).

Deliverables
Our final deliverables are planned to be as follows:

- Multi-OS architecture; supports a real-time logging mechanism for inspecting that verifies the correctness of real-time task execution in real time.
- Accountable Konoha language, a newly designed dependable scripting language that supports a static type checker and automated logging for inspection verification of the correctness of scripted programs.
- D-Logger; collects various types of logs from the various layers in the system.
- D-Analyzer; provide the various type of analysis for the log, such as a model-based anomaly detector and proactive performance anomaly detector that support the detection of anomalies from their logs with novel approaches.
- Evidence Viewer; evidence management viewer that views various aspects of system behavior in terms of agreed-upon dependability agreements (D-Case), risk analysis (a FTA extension), and event traceability (cause analysis).
- Knowledge Data for Robotics failures.

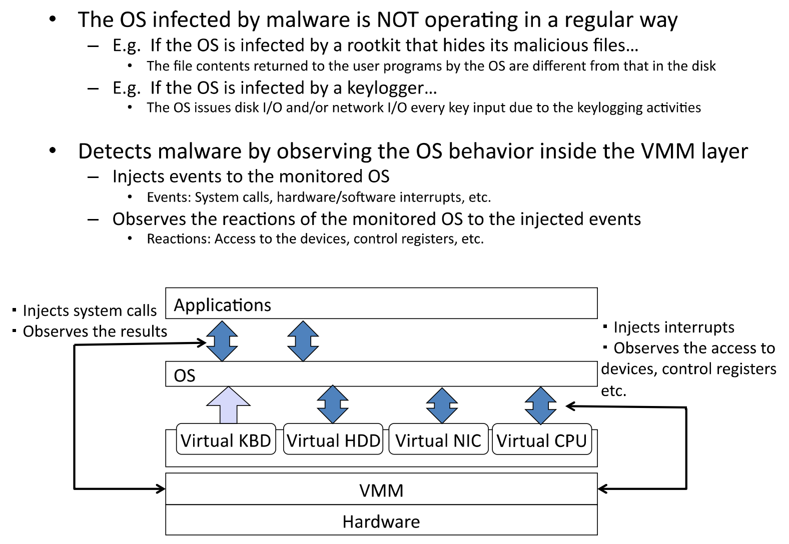
5.4 Security

	System Change Requests based on Stakeholders' Agreement	Design, Implementation, Verification, Testing	Achievement of Accountability	Failure Prevention	Responsive Action	Cause Analysis	Normal Operation
Technology to defend attacks from networked nodes (Security)				✔	✔		✔

Objectives
To improve open system dependability, a security mechanism is mandatory to defend against various security threats. We provide a secure execution mechanism for operating systems; it guarantees that the security mechanism of the operating system is working correctly. It monitors the runtime behavior of the operating systems and guarantees it is working as it is expected.

The hijacking of operating systems is one of the most serious threats to 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 functions in an unexpected way. We provide a secure execution environment for operating systems by defending against attempts to hijack operating systems.

In open systems, security mechanisms such as authentication, authorization, and access control should evolve as the demands of applications change. Actually, these fundamental security mechanisms are still hot topics of security research. In this project, we allow the use of new operating systems equipped with advanced security mechanisms, and guarantee that the operating systems are working in the expected way by running them in a secure environment.



- The OS infected by malware is NOT operating in a regular way
  - E.g. If the OS is infected by a rootkit that hides its malicious files…
    - The file contents returned to the user programs by the OS are different from that in the disk
  - E.g. If the OS is infected by a keylogger…
    - The OS issues disk I/O and/or network I/O every key input due to the keylogging activities
- Detects malware by observing the OS behavior inside the VMM layer
  - Injects events to the monitored OS
    - Events: System calls, hardware/software interrupts, etc.
  - Observes the reactions of the monitored OS to the injected events
    - Reactions: Access to the devices, control registers, etc.

Fig. 16. Monitoring OS behavior using VMM

Strategies
To implement the above goals, we use virtual machine technologies. More concretely, we prepare at least two virtual machines, one for executing a monitored operating system and the other for executing a monitoring operating system. The monitoring operating system monitors the behavior of the monitored operating system, checks the validity of the behavior, and raises an alert when the operating system behavior is deviant. (See Figure 16).

The monitoring operating system can inspect various types of behavior of the monitored operating system. For example, access to privileged registers, execution of privileged instructions, and I/O operations can be reliably inspected because the monitored operating system cannot fake these behaviors. Furthermore, the underlying VMM can inject hardware/software traps into the system. Through this trap injection and observation, it can be verified whether the monitored operating system is functioning as expected (See Figure 16). We also provide the rigorous isolation of virtual machines that can avoid denial-of-service attacks.

Our design has several advantages over the traditional signature-based detection and prevention of malware infection. Traditionally, a signature must be developed for each malware *sample*. In contrast, for our monitoring system we have only to develop 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 system call results; 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, but the monitoring operating system is carefully designed to facilitate the development of such modules.

To develop monitoring modules, intimate knowledge about malware is necessary. Modern malware is quite difficult to analyze because they are tactically ciphered and obfuscated. Even if we use a debugger to analyze malware, the malware stops its execution when it detects the existence of the debugger. In this project, we have also developed a suite of malware analysis tools; currently, we are developing a behavior analysis tool for malware that makes use of symbolic execution.

Deliverables
- D-System Monitor module to monitor OS behavior
- Virtual Machine Security Architecture
- Support tool for analyzing malware behavior

5.5 Virtualization and its Application

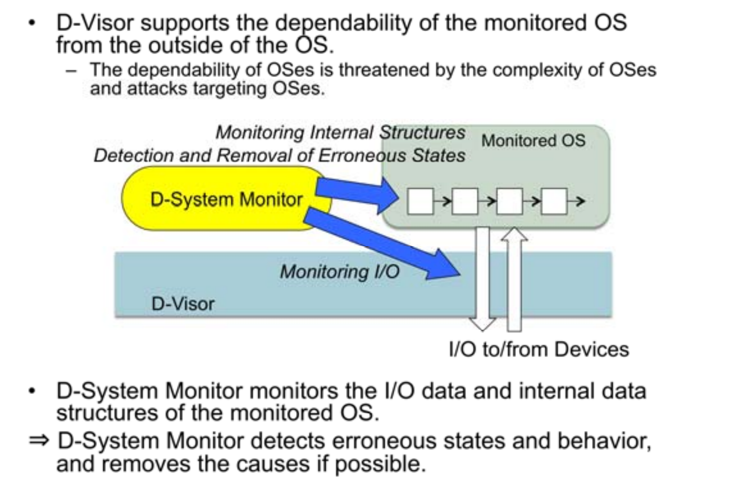
	System Change Requests based on Stakeholders' Agreement	Design, Implementation, Verification, Testing	Achievement of Accountability	Failure Prevention	Responsive Action	Cause Analysis	Normal Operation
Technology to monitor and log various events that occur in systems (Monitor & Logging)			✔				✔
Technology to identify and extract causes (Cause Analysis)				✔	✔	✔	
Technology to maintain memory resources in clean condition (Software Anti-aging)				✔			✔
Technology to defend attacks from networked nodes (Security)				✔	✔		✔
Technology to realize System Containers (Isolation)			✔	✔	✔		
Technology to make virtualization layers reliable (Verification of Virtualization Layer)		✔					

Objectives
The D-System Monitor aims to contribute to Open Systems Dependability by employing virtualization technologies, and D-Visor provides the virtualization technologies that D-System Monitor requires. D-System Monitor and D-Visor are parts of the D-fops (Framework), and they work together with the other components that constitute D-fops. D-System Monitor realizes technologies such as Monitor & Logging, Cause Analysis, and Security, as its elemental technology. D-Visor also realizes Software Anti-aging and Isolation technologies. D-System Monitor and D-Visor constitute the basis of the security architecture described in Section 5.4.

Strategies
D-fops (Framework) is based on virtualization technologies, and its functionalities rely on services constructed on a virtualization layer outside of an OS. D-System Monitor and D-Visor are a monitoring service and a virtualization layer of D-fops, respectively. Their details are described below.

The objective of D-System Monitor is to realize Monitor & Logging, Cause Analysis, and Security. In order to achieve its objective, D-System Monitor continuously monitors the behavior and internal data structures of an OS from the outside. And then, D-System Monitor analyzes the monitored data in order to guarantee the OS is operating in the expected way.

D-Visor realizes the Isolation mechanism and implements System Containers in order to enable D-System Monitor to safely monitor an OS from the outside. By executing an OS in a System Container realized by the Isolation mechanism of D-Visor, the execution environment of D-System Monitor can be clearly separated from the environment of the monitored OS (see Fig. 17).



- D-Visor supports the dependability of the monitored OS from the outside of the OS.
  - The dependability of OSes is threatened by the complexity of OSes and attacks targeting OSes.

Monitoring Internal Structures
Detection and Removal of Erroneous States
Monitored OS
D-System Monitor
Monitoring I/O
D-Visor
I/O to/from Devices

- D-System Monitor monitors the I/O data and internal data structures of the monitored OS.

⇒ D-System Monitor detects erroneous states and behavior, and removes the causes if possible.

Fig 17. Dependability Support by D-System Monitor on an isolated environment realized by D-Visor

D-System Monitor guarantees the monitored OS is operating in the expected way by analyzing its runtime information provided by D-Visor. D-Visor provides the runtime information, such as access to privileged registers, execution of privileged instructions, and I/O processing and data, for the D-System Monitor. Moreover, D-Visor can monitor the reactions of the monitored OS to more specific events by injecting external interrupts and software interrupts that invoke system calls. From the monitored behavior, D-System Monitor can determine whether the monitored OS behaves as expected. D-Visor also provides Software Anti-aging mechanism that quickly and simply recovers data integrity in a monitored OS.

D-Visor should not cause failures since it is introduced into the system in order to achieve Sustainability, which is the essential objective of Open Systems Dependability. Thus, D-Visor cannot be compromised either since it is the base of the system that enables the monitoring of an OS from the outside of it. Therefore, the implementation of D-Visor is verified by the model checker, which is being researched and developed by this project. The verification effort of D-Visor aims to provide a specification description to avoid fatal errors that lead to the service outage of System Containers.

It is very difficult that a single implementation of D-Visor fulfills all the requirements for every kind of embedded systems. Since embedded systems are used for a wide variety of purposes in a number of ways, their utilization environments differ substantially. Low power consumption can be a very high priority requirement for some systems while real-timeliness can be the highest priority requirement for some other systems. There are also many kinds and configurations of embedded processors used in target systems. Therefore, it is better to provide different implementations of D-Visor that match the needs of target embedded systems. While D-System Monitor utilizes System Containers, it requires different functions for System Containers depending upon the functionalities it realizes. Thus, a metric is being developed to help users construct systems that best fit their needs.

Deliverables

- ➢ D-System Monitor modules
  - ● Software module to monitor OS internal data structures
  - ● Software module to monitor OS behavior
- ➢ D-System Monitor runtime API
- ➢ D-Visor for embedded systems
- ➢ D-Visor for hard real-time systems
- ➢ Verification specification description for D-Visor

5.6 Systems Software Verification

	System Change Requests based on Stakeholders' Agreement	Design, Implementation, Verification, Testing	Achievement of Accountability	Failure Prevention	Responsive Action	Cause Analysis	Normal Operation
Technology for Model Checking of Systems Software		✔		✔			
Technology for Type Checking of Systems Software		✔		✔			

Objectives
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 program development approaches and tools for detecting defects in systems software including operating system kernels, thereby contributing to the achievement of Open Systems Dependability, especially for the "Design, Implementation, Verification, and Testing" phase of the DEOS process. In addition, the approaches and tools support the "Fault Prevention" phase in the sense that the tools can detect bugs in programs before executing them.

Strategies
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 which are frequently used for systems software (Figure 18 and Figure 19) in the "Design, Implementation, Verification, and Testing" phase. In the DEOS process, verification tools are used not only in the development phase of a software system, but also in the maintenance phase where a system continuously evolves in order to improve Open Systems Dependability.

From the viewpoint of achieving Open Systems Dependability, both of the 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 amount of time. However it can verify relatively simple safety properties. To address this problem, 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 conditions of failure to the properties to be checked.

While properties checked by model checking can be written by programmers, it is unclear what properties should be written and how. To make it clear, we provide a specification description to extend the Linux kernel safely by defining the constraints on functions and variables.

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 with 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 forms 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 user demands.

Deliverables

- Model checker for systems software written in C
- Specification description language for systems software
- Specification descriptions for Linux kernel extensions
- Typed Assembly Language (TAL) for systems software and type checker
- Type-safe compiler from C to TAL

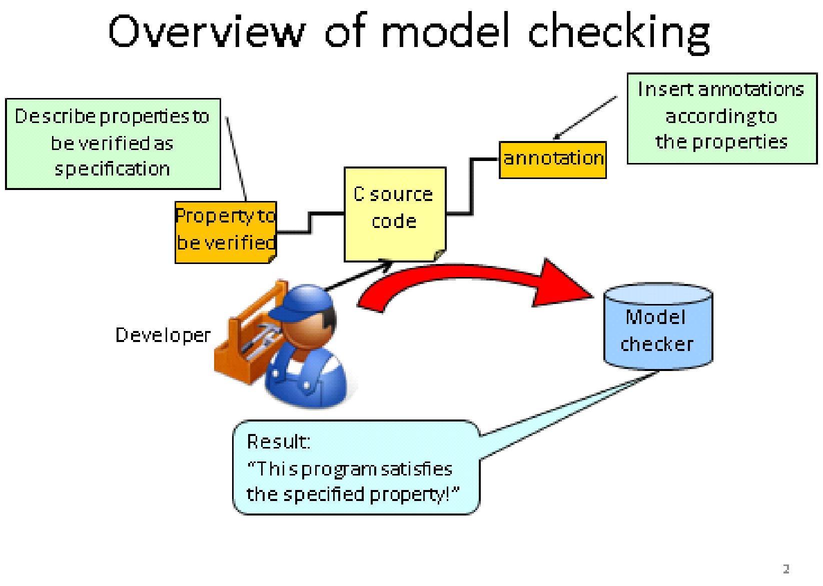

Overview of model checking
Describe properties to be verified as specification
Insert annotations according to the properties
annotation
C source code
Property to be verified
Developer
Model checker
Result: "This program satisfies the specified property!"
2

Fig. 18.

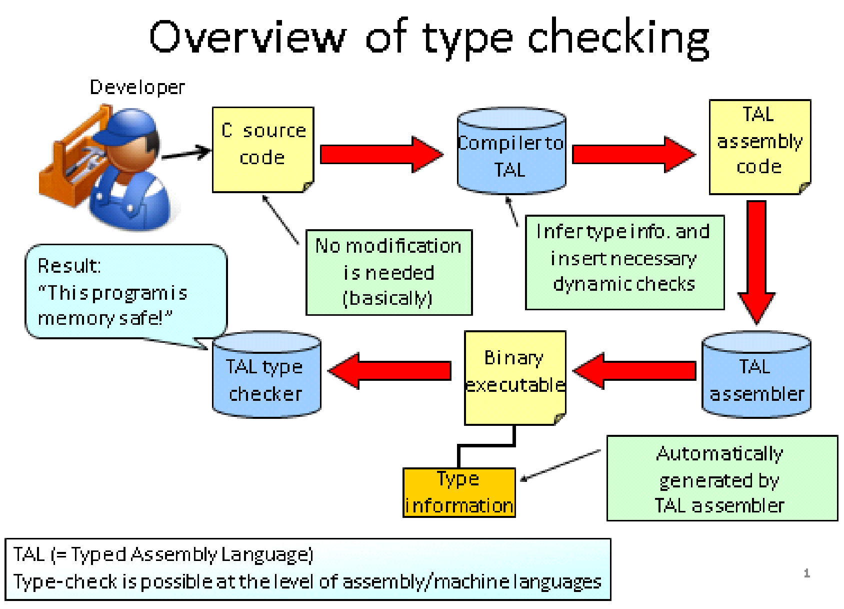

Overview of type checking
Developer
C source code
Compiler to TAL
TAL assembly code
No modification is needed (basically)
Infer type info. and insert necessary dynamic checks
Result: "This program is memory safe!"
TAL type checker
Binary executable
TAL assembler
Type information
Automatically generated by TAL assembler
TAL (= Typed Assembly Language)
Type-check is possible at the level of assembly/machine languages
1

Fig. 19.

5.7 Dependability Measurement Tools and Test Tools

	System Change Requests based on Stakeholders'	Design, Implementation, Verification, Testing	Achievement of Accountability	Failure Prevention	Responsive Action	Cause Analysis	Normal Operation
Observation of Load, Anomaly Condition, and Behavior		✔	✔	✔			✔
Acceleration of the System Test by Managing Massive Computing Resources		✔		✔	✔	✔	

Objectives
In the D-Case process, anticipated anomaly conditions are enumerated, and evidence is needed to show whether the system requirements are satisfied under each anomaly condition. The limit of the system against these anomaly conditions must be calculated beforehand by necessary quantitative measurements. In addition, in the case of system updates, it must be verified that no problems arise with a system operation test. The cause of anomalies observed in actual operation must be analyzed in order to ensure accountability to stakeholders. One of the objectives of this project is to develop tools for measuring dependability and for rapid system testing, which will contribute to achieving Open Systems Dependability.

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.

Strategies

The DBench project was an approach to evaluate the dependability of the system carried out from the late 1990s to the early 2000s. The dependability benchmark framework, which is a conceptual framework, was developed in the DBench project. In this framework, the benchmark environment consisted of a Benchmark Target, Workload, Faultload, and Measurement. In the DBench project, a separate benchmark environment for each specific application domain was implemented based on the defined framework.

On the contrary, in our project, DS-Bench/D-Cloud have been designed and implemented to provide a dependability evaluation environment with adaptability and extensibility. DS-Bench/D-Cloud can be commonly used for various application areas. Our project partially shares the goals of the DBench project. 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.
Anomaly behavior caused by software problems, such as software bugs and overload, and hardware malfunctions that can be caused by controlling hardware devices, such as network disconnection and power shutdown, are simulated using real machines. If the specifications of a device are written in the SpecC system description language, hardware faults can be simulated by the combination of the SpecC simulator and Virtual Machine Monitor on D-Cloud.
Based on the framework of DS-Bench, the fault scenario that describes when to generate faults at what workloads, and with which measurement tools is described in XML. This fault scenario is reflected in the scenario for setup and execution of the D-Cloud testing environment, which is also described in XML, and it is executed in virtual machines and real machines.
The results of the execution based on a fault scenario are also generated in XML format, and are stored in a database. This database provides evidence of how the system responds to overload or faults.

Deliverables

Figure 17 schematically illustrates the DS-Bench and D-Cloud we aim to create. Their overall features are as follows.

- DS-Bench/D-Cloud Front end :
  This provides a Web user interface for users to configure the fault scenario.
  Through this interface, many types of software dealing with Anomaly Loads, Performance Benchmarks, Monitoring Tools, and Anomaly Scripts are entered in the database. Users can use this interface to configure the dependability benchmark environment. Our project will provide the deliverables shown below. In addition, users can flexibly append a group of software as necessary.
  - Anomaly Loads : Anomaly Loads are classified into two types: fault emulators for software, and fault emulators for hardware. The fault emulators for software consist of programs that artificially generate overloads of the CPU, disk I/O, network traffic, and heavy memory usage. The fault emulators for hardware control the commodity PDU (power strip) to emulate power shutdown faults and the network switch to emulate network disconnection faults. To deal with faults in memory flip and I/O devices, etc., an environment with these can be simulated on Virtual Machines. I/O devices that are not used in general-purpose computers must be described in the SpecC system description language.

- Performance Benchmarks : These are programs that evaluate whether the required performance is sustained under anomaly loads. Benchmark programs that are used to evaluate commonly used products are available.
- Monitoring Tools : These are tools to monitor the states of the computer.
- Anomaly Scripts : These are fault scenario databases defined by users, and some examples may be provided.

➢ D-Cloud and Fault-VM :
The environment for executing the dependability benchmark is provided by D-Cloud and Fault-VM. Benchmarks that are executable with Virtual Machine and benchmark environment without special hardware are performed in the cloud computing environment offered by cloud management software (currently, Eucalyptus), including I/O device models described in the SpecC language. If the target system consists of real machines or requires a special environment, the real environment is directly used. D-Cloud assigns the programs based on Anomaly Scripts to specific Virtual Machines or real machines.

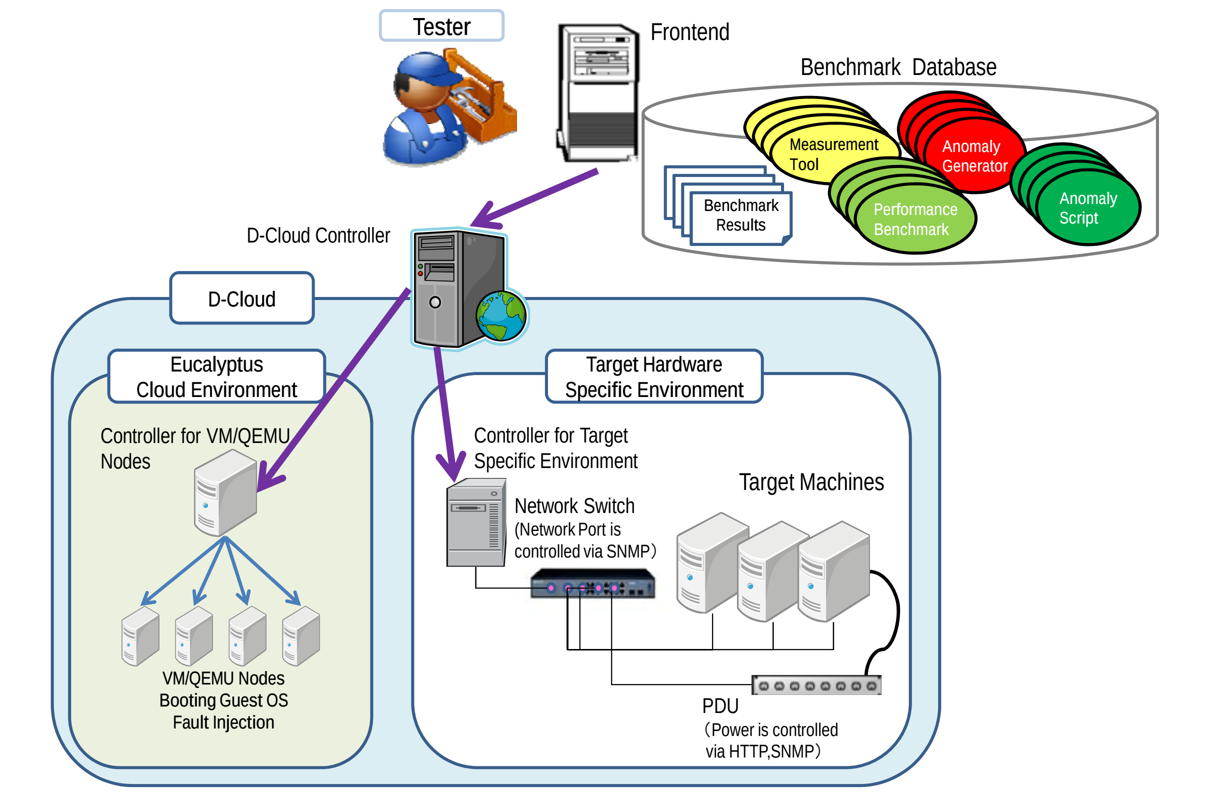


Fig.20. Image of DS-Bench/ D-Cloud as a goal

5.8 Process and International Standards

Objectives

Our goal is to develop a dependability standard for information systems that provides objective criteria with which developers can argue for, and users can be convinced of, the dependability of a system. We also establish a certification scheme based on these criteria, assessing a system's conformity to the standard. The certification of DEOS will make its value clear, and will itself be a tangible added value of the whole system. We intend the standard to be internationally recognized. We plan to set up study groups within the international standards bodies in which we participate, to promote international collaboration in this area, and to submit a standard proposal together with international partners.

Strategies
The research and development being conducted has the following focuses:

1. Formation of the “User Oriented Dependability” concept
2. Drafting dependability standards
3. Establishing an evaluation methodology
4. Establishing lifecycle technology

Item 1 clarifies the concepts of indeterminacy and diversity that have become issues in Open System Dependability, and explores solutions to the challenge of developing "measures against unexpected failures," seemingly a contradiction in terms. Item 2 formulates the qualities required of dependable information systems, based on the open system dependability concepts clarified in Item 1. Item 3 provides methods to evaluate a system's conformity to the standard. Item 4 gives guidelines on how to design a system lifecycle for continuing conformity to the standard.

Details of activities and progress with regard to each research focus are as follows.

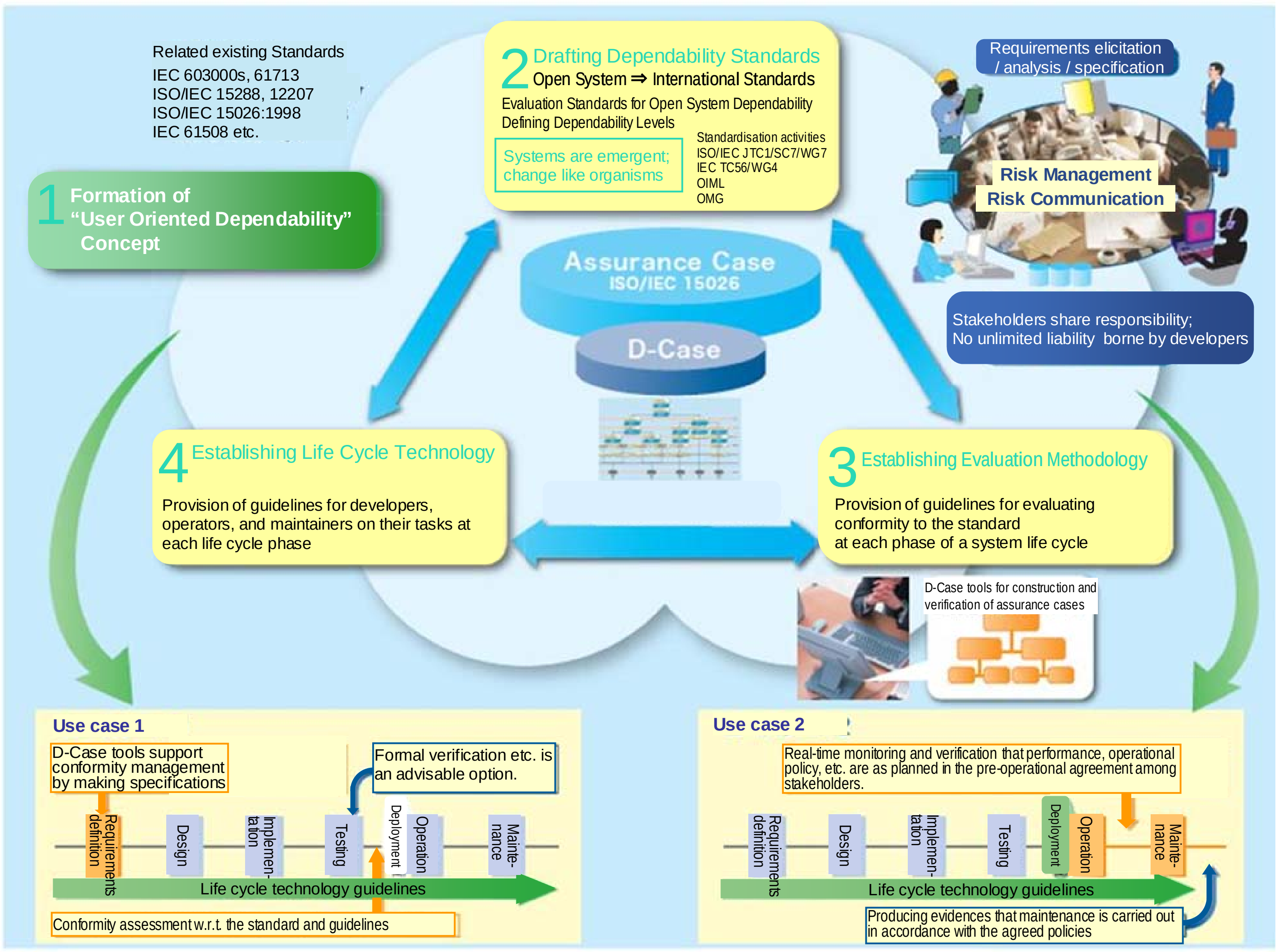

Related existing Standards
IEC 603000s, 61713
ISO/IEC 15288, 12207
ISO/IEC 15026:1998
IEC 61508 etc.
2 Drafting Dependability Standards
Open System ⇒ International Standards
Evaluation Standards for Open System Dependability
Defining Dependability Levels
Systems are emergent; change like organisms
Standardisation activities
ISO/IEC JTC1/SC7/WG7
IEC TC56/WG4
OIML
OMG
Requirements elicitation / analysis / specification
Risk Management
Risk Communication
Stakeholders share responsibility; No unlimited liability borne by developers
1 Formation of “User Oriented Dependability” Concept
Assurance Case
ISO/IEC 15026
D-Case
4 Establishing Life Cycle Technology
Provision of guidelines for developers, operators, and maintainers on their tasks at each life cycle phase
3 Establishing Evaluation Methodology
Provision of guidelines for evaluating conformity to the standard at each phase of a system life cycle
D-Case tools for construction and verification of assurance cases
Use case 1
D-Case tools support conformity management by making specifications
Formal verification etc. is an advisable option.
Requirements definition
Design
Implementation
Testing
Deployment
Operation
Maintenance
Life cycle technology guidelines
Conformity assessment w.r.t. the standard and guidelines
Use case 2
Real-time monitoring and verification that performance, operational policy, etc. are as planned in the pre-operational agreement among stakeholders.
Requirements definition
Design
Implementation
Testing
Deployment
Operation
Maintenance
Life cycle technology guidelines
Producing evidences that maintenance is carried out in accordance with the agreed policies

Fig. 21. International Standard and Process Development Approach

Formation of “User Oriented Dependability” concept
Through reviews of existing dependability concepts, discussions among Open System Dependability groups, and external feedback to the results of the above, we came to identify crisis

management for information systems as a key concept in addition to risk management. Since conventional risk management starts by enumerating risks, it may only counter risks perceived in advance. The achievement of Open System Dependability, on the other hand, starts by recognising the necessity to cope with unexpected failures, which is more akin to crisis management in society at large. The means of crisis management include emergency procedures, accountability, and recurrence prevention.

Drafting dependability standards

We are active in the working groups of international standards bodies such as ISO/IEC JTC1/SC7/WG7 (Software and system engineering - Lifecycle management) and IEC TC56/WG4 (Dependability - System aspects of dependability) as members, editors, or experts, cultivating collaborations to form the world community for the formation of new dependability standard proposals. In particular, since 2009, we have been acting as co-editors of the draft international standards ISO/IEC 15026 for assurance cases.

Establishing evaluation methodology

Agreement among diverse stakeholders in a system will become an ever more important issue for crisis management and risk management. It is necessary to have systematic support for reaching complex agreements, for checking correctness of agreement procedures, etc. A tool to construct D-case documents and verify their consistency is being developed.

Establishing lifecycle technology

For Open System Dependability, it is important for agreement among stakeholders to be appropriately maintained throughout system lifecycles. The D-Case management process to attain this is being developed.

Deliverables

- Publication of concept definitions: establishing Open System Dependability concepts.
- ISO/IEC international standards: draft standards for Open System Dependability, starting the process towards formal approval by ISO/IEC.
- OMG industry standards: proposals for D-Case standards with industry-wide interoperability made at OMG meetings.

5.9 Research and Development Activity Results

Each of our teams has made contributions, including software for the framework and documents which describe processes, standards, guidelines, etc. Fig.22 is a map of these contributions.
DEOS tools produced by this project must work with existing tools. Software developed by this project is just for reference, and it must be remade to be of practical use to the products and services of each user. The DEOS development center will work with users to fit the system to user environments.

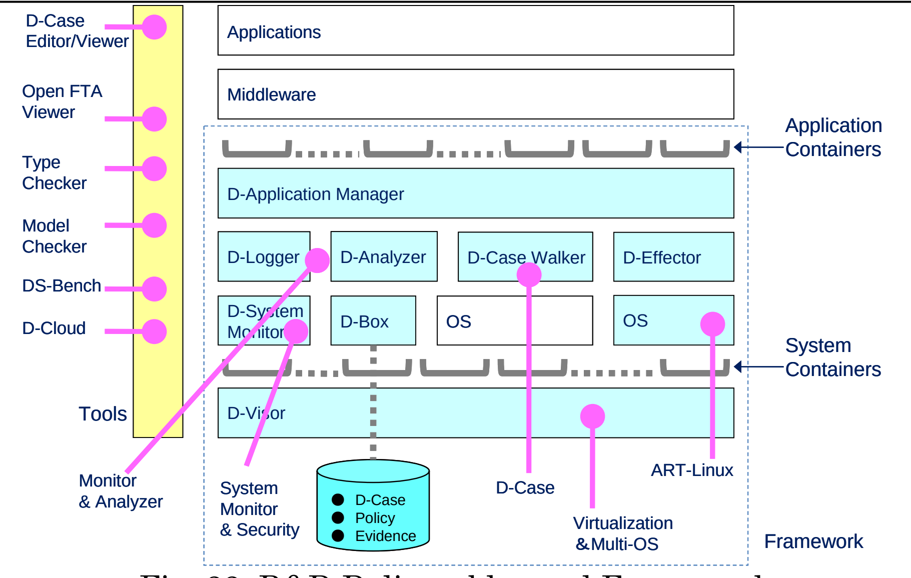


Fig. 22. R&D Deliverables and Framework

6 Research and Development Organization

This project formed five research teams in 2006 and added four new teams in 2008. They will continue research and development until March 2012 and March 2014, respectively. At the Dependable Embedded OS Research and Development Center (DEOS R&D Center), the deliverables of the research teams will be integrated for practical use, reconfigured in consideration of intellectual property and maintenance issues, tested, assessed, and packaged in collaboration with the businesses that will use them in actual products (Fig. 23).

Although there are currently nine teams participating in this project, their research in their assigned topics alone may not be enough to achieve our mission. For instance, file system dependability is deemed important, but it is not currently being studied.

From here on, through the proposals of the system architecture, the design (basic and detailed) of the reference system, and the process of identifying the required elemental technologies, it will become clearer whether open source software can be utilized as is or not; or, whether it is necessary to conduct further research and development activities in this project.

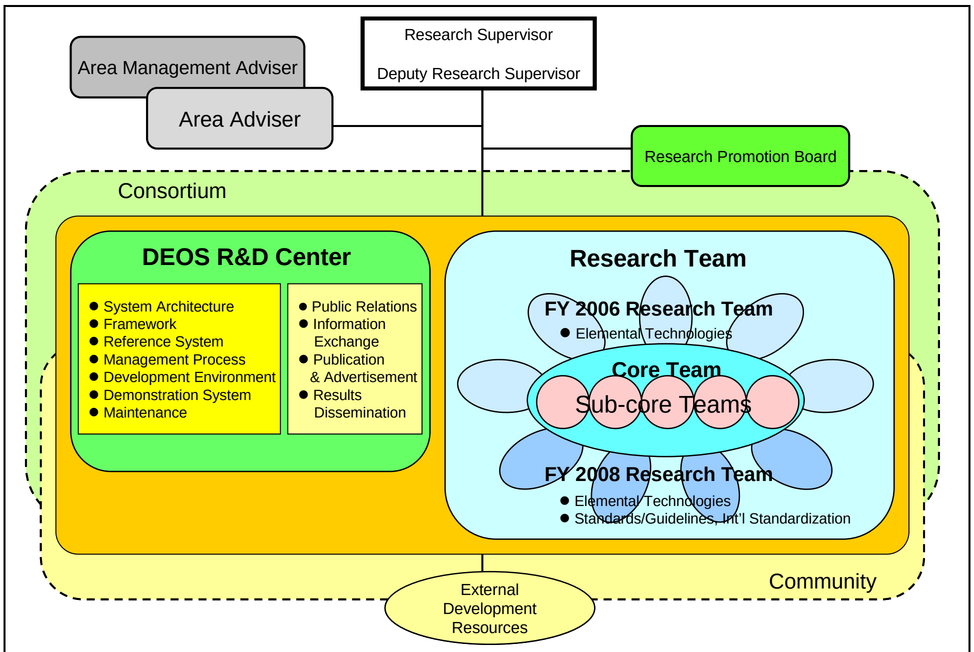


Fig.23. Research and Development Organization (As of October 2010)

To move this project forward, a Research Promotion Board was established which includes members from private industry to represent the viewpoint of system or service providers. The board members and the system and service providers will constantly verify the OS requirements, determine the requirements for its practical implementation and identify problems related to its practical application. The progress of the project will be disclosed to the public, opinions from people outside the team will be invited, and feedback that can be applied to the whole project will be gathered. This will be done to make the concept of dependability and the methods of developing and maintaining dependable systems into common public property. The present age of borderless computer systems and businesses brought on by the Internet and economic globalization calls for the concepts and techniques of this project to be shared with the international community.

On April 1st 2010, we regrouped the core team, whose members are from each research group, into 5 sub-core teams as given below. Each sub-core team researches an area more deeply, to enable practical use. Most of the results of the sub-core teams and research teams are to be integrated within the framework (D-fops) or built into tools. We provide the deliverables needed for system or service providers to realize dependable systems and services.

- D-Case & Metrics team:
  Research and development, consensus-building for dependability metrics and their required tools.
- EBI team:
  Research and development of system monitoring, analysis and profiling and required tools.
- VM & Multi-OS team:
  Research and development of virtualization, its application, and required tools.
- Systems software verification team:
  Research and development of system software verification and its required tools.
- DS-Bench & D-Cloud team:
  Research and development of dependability measurement and evaluation, and required tools.

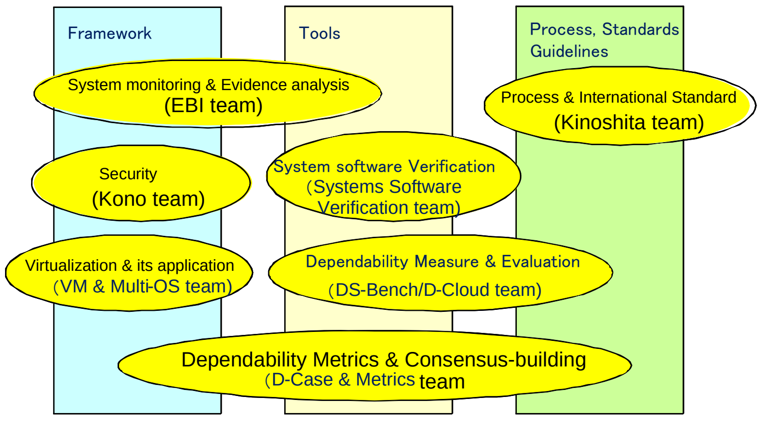

Framework
Tools
Process, Standards Guidelines
System monitoring & Evidence analysis (EBI team)
Process & International Standard (Kinoshita team)
Security (Kono team)
System software Verification (Systems Software Verification team)
Virtualization & its application (VM & Multi-OS team)
Dependability Measure & Evaluation (DS-Bench/D-Cloud team)
Dependability Metrics & Consensus-building (D-Case & Metrics team

Fig.24. Role of each Sub-core team

Also, the following teams collaborate with sub-core teams to realize Open Systems Dependability. (Fig. 24)

- Kono team: Research and development of security and required tools.
- Kinoshita team: Research and development of processes and international standards for dependability

7 Roadmap

The completion of the following phases devoted to the items described earlier are the principal milestones of the entire project (Fig.25).

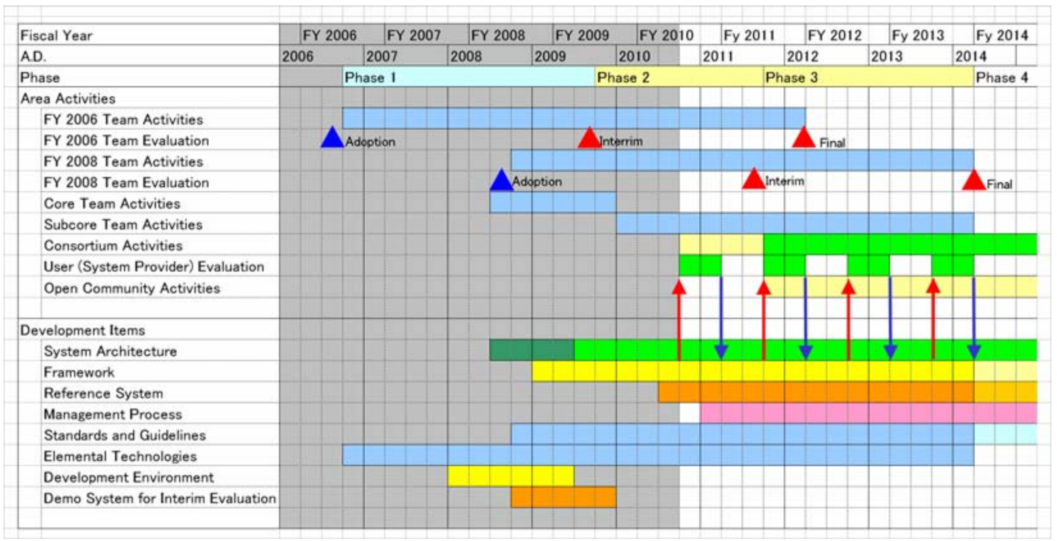


Fig.25. Roadmap (As of October 2010)

- 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 have been integrated. (Presentation completed 2009/9 by the 2006 research team.)
- Phase 2 (2009/10-2011/9): Implementation of a system architecture, framework, and reference system which adopts the elemental technologies of the research team; establishment of a consortium (or user organization) composed of actual potential users including those from private industry; formation of an open community for the research and development of elemental technologies and system architecture; standardization of required items; trial usage of the framework and reference system by consortium members and the gathering of feedback thereof; demonstration of the research teams' frameworks and reference systems in which some of the elemental technologies have been integrated. (The above will be presented in September 2011 by the 2008 research team as their interim public presentation.)
- Phase 3 (2011/10-2014/3): Trial usage of the framework and reference system by consortium members; continuing evaluation and feedback; transition to actual development and commercialization; standardization of the required items.
- Phase 4 (20014/4-): Functional enhancements; and continuing utilization, maintenance and development by the consortium.

8 Further Issues for Practical Application

8.1 Handling of Intellectual Property Rights and Copyright

The research and development deliverables of this project will be provided to as many system and service providers and users as possible for their practical use. The deliverables will be provided in the form of OSS (Open Source Software) as much as possible, in order to contribute to the development of a social infrastructure using dependable embedded systems. There are various kinds of OSS licensing, such as GNU GPL (GNU General Public License), GNU LGPL (Lesser General Public License), New BSD License, and MPL (Mozilla Public License). The selection of license type will depend on what is considered to be the best way to spread the use of the deliverables of this project.

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 policy will be decided accordingly. Intellectual property will be transferred to a specific organization if required and if there is support from each research organization concerned. The final objective is to provide an IP "one-stop shop" for the user through one organization such as the DEOS Consortium. Collaboration with external research and development groups as well as standards organizations will be encouraged, in order to provide user-friendly deliverables. (The specifics of such collaboration will be considered as a future issue.)

8.2 Open Systems Dependability Consortium

To put the deliverables of this project to practical use, it is required that their prototypes be tried out by system and service providers for evaluation, that they go through a large scale evaluation process, that the evaluation results are used to improve them for practical use, and that the number of their supporters for practical use be increased. The establishment of a consortium of system and service providers who are potential users, businesses that are potential providers of the deliverables, and related research organizations will help the deliverables of this project to continue being used after their evaluation. This consortium may well take the role of international leadership in dependability technology, attracting supporters and users of the deliverables and technologies, and contributing to standardization, the development of new technologies, maintenance, education, certification, etc. The DEOS Center will strive to establish this consortium before project completion, working with potential users and research organizations. Future issues whose clarification is needed to achieve these objectives are the method of cooperation with potential users and research organizations, as well as the role, formation, and funding of the consortium. For potential users and research organizations to agree to join the consortium, they must understand how critical Open Systems Dependability is, and each system or service provider must realize that Open Systems Dependability must be tackled throughout each concerned industry and as a cross-industry matter. Though the required conditions or the roles of the consortium are under study with market research, current issues for the objectives or business of the consortium are summarized below.

1. Understanding the importance of Open Systems Dependability and influencing public opinion

   When system failure occurs, the system or service provider necessarily takes action, and the users and the service provider will be satisfied to some extent if that action is what was expected by the users or community. The uncertainty of users and society at large regarding the cause of the failure is relieved if they see evidence of what has happened presented according to a standard format and procedure.

   The report "Towards the Realization of Safe and Reliable IT Infrastructure in Information Society" issued by the Information Science Committee of the Science Council of Japan in 2008 [28], proposed the "establishment of a fact-finding commission for accidents involving vulnerability of information systems", that would make proposals about law and education systems. Collaboration with this kind of organization should be considered.
2. Establishment of industry-wide or society-approved standard procedures or international standardization

   The procedures for handling system failure and describing it to users or to the public, the content to be described and required evidence of causes are to be defined and standardized. This will clarify accountability and provide a standard throughout the community.
3. Certification of standard compliance

   There may be a need to certify that the products or processes comply with the established standards. There is a potential business here, in which a third party agency does the certification with support from the consortium.
4. Development and maintenance of the deliverables by the development community

   The development of the framework and elemental technologies required to support Open Systems Dependability cannot by fully carried out by the members of the DEOS project. Maintenance and continuous development will be required after the DEOS project is completed. Some of those activities may be carried out by the consortium, but it is important to utilize overseas resources, to establish an international community for DOES, and to get overseas world-class researchers and engineers to join in these activities. We hope for a world-wide discussion of the Open Systems Dependability concept, the improvement of architecture, application of the process to the real world, and research and development to realize and improve Open Systems Dependability.

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. 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. 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 and others, "Open System Science", NTT Publishing Co., Ltd
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
 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)
28. Dependable Operating Systems for Embedded Systems Aiming at Practical Applications Research Area (DEOS Project) White Paper Version1.0, DEOS-FY2010-WP-01, JST, Sep. 1, 2009

10 Appendix

10.1 Dependability Obstructions

If we try to focus on a software's working environment as a dependability obstruction, its faults can be divided into four categories: *faults from the environment during both operation and development*, *faults that are hardware-related*, *faults brought about by human error*, and *faults caused by a random attack*. Each fault category can be fully understood by considering the dependability obstructions that type of fault causes in each lifecycle from the time of inception until termination. A summary is shown in the table below (Table 2).

	Environment (Operating Environment, Development Environment, etc.)	Hardware	Human Error	Security Issue / Risks
Specification	changes in the environment [natural environment, system environment (hardware, software), user environment (organization, operating environment), reuse of existing applications, frequent changes in requirement]	errors in hardware resource estimation, lack of consideration for software productivity	errors in specification estimation, performance simulation (inaccurate load), noncompliance to standards	theft of plans/specifications, sending of erroneous/malicious information from outside
Design	wrong tolerance design (for natural environment, system environment and operating environment), lack of consideration for target operator (end-user, administrator, etc.), bugs in design tool, errors in analysis of system interdependency, trouble arising from reusing existing programs, errors in design tool selection, frequent changes in specification	insufficient implementation of test functions, lack of consideration for software performance (inadequate software and hardware partitioning), specification mismatch between parts	errors in architecture selection/design, errors in interface design between modules/subsystems, omissions in design review, misinterpretation of specifications, errors in hardware performance estimation, errors in user interface design, wrong handling of exceptions, mismatch in software versions, insufficient design in fault recovery	theft of design information
Implementation and Unit testing	bugs in development tool, insufficiencies in development environment, bugs in development environment (version mismatch, etc), inadequate training program, insufficient verification (failure and performance) of the software to be used, frequent changes in design	schedule delay/miss in the development of target embedded hardware, poor yield/quality, specification mismatch bugs (that can be fixed within the term/cannot be fixed within the term)	errors in coding, omissions in code review, errors in algorithm, errors in library selection, integration version mismatch, errors in timing assumption, escapes in unit tests, piracy, patent infringement	illegal production fraud, theft of source code, patent litigation, copyright lawsuit, embedding of malicious code
Integration, Test	bugs in test tool/test environment, insufficient test period, frequent changes in implementation	insufficient testing on the hardware functions, left over hardware bugs	insufficient test items (test cases, operation), errors in test result verification, errors in test items, errors in test reviews or test environment configuration	mixing of inferior parts, altering of test results, theft of test specification/results, intentional omissions
Distribution / Transportation	changes in the environment (natural environment, distribution systems, people in charge, distribution rules)	changes in environmental factors (temperature, shock and tilt during transport, submersion in water, other damages)	mixing of defective and counterfeit parts, occurrence of accidents during transport	mixing of counterfeit and stolen parts, damage, tampering
Operation	aging (changes over time), environment temperature, environment humidity, errors related to other systems' failures, system downtime due to unexpected data, input overload, shock/impact, power failures (power flicker, fluctuations, blackout, unplugging of electrical outlet), noise (electromagnetic ray, static electricity, cosmic ray), frequent version updates/patches, excessive operation cost, service termination or malfunction due to remaining bugs	deterioration of hardware (mechanical, chemical, physical), poor contact (connection, switch), excessive power consumption, noise, electromagnetic noise, heat	misunderstandings of specification, user error due to poor user interface design, incorrect operation (incorrect function selection, incorrect data entry or selection), errors in installation, errors in configuration, errors in data transfer	infiltration during operation (spyware, virus, attack), attaching of illegal modules, extraction of data, intrusion, information leakage
Maintenance, Update	irreproducible bugs/errors, bugs in maintenance/update tool , frequent (excessive) version updates, lack of version update history data, excessive maintenance cost	lack of features for collecting malfunction information, Inappropriate maintenance period of parts, incompatibility of new parts	insufficient notice of a malfunction, communication errors in malfunction information, version mismatch, backup failure, restoration failure, insufficient version upgrades (incomplete operation)	infiltration during maintenance/update (virus, attack), installation of illegal modules, theft of data
Disposal, Reuse	traces of deleted information	pollution, insufficient recycling or reuse plans	errors in deleting personal information, operation history, etc.	theft of information/parts

Table 2. Dependability Obstructions

To identify these obstructions in embedded OS and other peripheral technologies, the table can be divided into three areas:

(1) Areas that should be covered by a process and/or organization (company) utilized by the developer or service provider; as well as areas that should be covered in system integration.
(2) Areas where dependability technology is required for other OS-related components.
(3) Areas where large contributions to dependability can be expected from the OS and other peripheral technologies.

These areas are respectively colored yellow, white, and orange in Table 2.

New techniques and technologies, along with traditional technologies such as CMMI, modern PM (Project Management) techniques, etc. need to be developed to address the problems posed by the above faults in modern embedded devices in the 21st century. These are required for the proper setting of requirements/specifications for design, development and testing. This difficult task must be accomplished, because with the development of large-scale software, embedded systems are no longer considered mere stand-alone products but rather as fundamental parts of a whole infrastructure that provides service to users through a network. The training of engineers and the management for this purpose have become crucial.

10.2 Related Standards and Organizations

Standard

- 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

Process Guide
- 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 https://www.trustedcomputinggroup.org/home
- CELF: CE Linux Forum, an international open source software development community
  http://www.celinuxforum.org/
- 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 https://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/

10.3 DEOS Project Terminology

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

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

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

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

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

Dependability: Ability to deliver services that can justifiably be trusted. This is a composite of availability, reliability, safety, integrity and maintainability.

Open system: A system whose definition keeps changing during development or during operation and/or whose operation may change with environmental changes such as connection to external systems through a network.

Closed system: A system which is isolated from other systems during operation, and whose system requirements and configurations do not change during operation.

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

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

Incompleteness:　The property of a system with incomplete requirement specifications, so that it is difficult to fully understand as well as guarantee the system's behavior upon shipment.

Uncertainty: The possibility that a system's configuration will be changed by its usage environment during the lifecycle of the system, making it difficult to completely predict the behavior of the system during the design phase.

Open systems failure: Failure that is caused by incomplete and uncertain factors which are not clear in the design phase. The possibility of these failures is inherent in embedded systems.

Open Systems Dependability: The ability to continuously perform measures that remove the factors of failure before they cause failure, to provide appropriate and quick action when they occur, to manage the failure in order to minimize the damage so that the system can safely and continuously provide the services expected by users to as great a degree as possible, and to maintain accountability for the system operations and processes.

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

DEOS Process: Double helix process consists of the "requirements/environment change accommodating cycle" and the "failure reacting cycle" to achieve open systems dependability.

D-Case Growth Cycle: A cycle in which D-Case expands and improves over time, which indicates improvements in the system.

Requirement/Environment Change Accommodating Cycle: A cycle to respond and accommodate to changes in stakeholders' requirements or changes in environment.

Failure Reacting Cycle: A cycle to react to system failures.

System Change Requests based on Stakeholders' Agreement: Requests to change systems with stakeholders' agreement after discussion and resolution of the existing conflicts of change requests among stakeholders.

Failure Prevention: To avoid system failure by predicting system failure and anomalies.

Responsive Action: To respond to system failure or to a problem system state as soon as possible

Normal Operation: Daily operation with appropriate periodical inspection and preventive maintenance, and with effort to avoid the repeat of similar failures by performing continuous improvement activities (Kaizen).

Achievement of Accountability: To deal with system failure by determining and explaining the cause, making a recovery plan, and explaining the current status and procedures taken clearly to the stakeholders.

Cause Analysis: To identify the cause of failure or to identify areas of possible causes of failure based on evidence.

Embedded System: A system with software within a system whose primary purpose is not computational.

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

Elemental Technology: Technology to realize required functions.

Process: Steps for developing and operating systems or services, which include requirement formulation, design, development, implementation, testing, operation and maintenance phases.

Metrics: Qualitative or quantitative indices to evaluate objects. Quantitative values are preferable, because they facilitate comparisons and the certification of improvements.

Framework: D-fops is the framework of our project, which enables the construction of an architecture of Open Systems Dependability, to which are integrated the elemental technologies enabling the required processes.

Standard: The functions achieving an objective, with the required level of performance and quality defined.

Internationalization: To deploy certain concepts or technologies worldwide, not limiting the deployment to a country or predefined area.

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

Evidence: Valuable information which supports a claim through an argument.

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

Formal Verification: To prove the correctness or incorrectness of programs by formal or mathematical methods.

Model Checking: Software verification technique that exhaustively checks whether a program works correctly, without the occurrence of any critical error conditions such as deadlock or infinite loop.

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

Specification Description Language: A language which can describe the properties that a program needs to satisfy.

TAL: Typed Assembly Language: Assembly language annotating types to values, which enables type checking for proper memory management or flow count.

10.4 DEOS Project Members

Research Supervisor

Mario Tokoro, SONY Computer Science Laboratories, Inc.

Deputy Research Supervisor

Yoichi Muraoka, Waseda University

Area Advisors

Kazuo Iwano, IBM Japan, Ltd.
Tohru Kikuno, Osaka University
Kohichi Matsuda, Information-Technology Promotion Agency, Japan
Yoshiki Seo, NEC Laboratories America, Inc.
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
Kazuyasu Sasuga, Fuji Xerox Co., Ltd.
Yuzuru Tanaka, Hokkaido University
Seishiro Tsuruho, HAL Tokyo

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