A Failure Analysis of Data Distribution Middleware in a Mission-Critical System for Air Traffic Control

Domenico Cotroneo[1], Antonio Pecchia[1], Roberto Pietrantuono[1], and Stefano Russo[1,2]
[1] Dipartimento di Informatica e Sistemistica, Università degli Studi di Napoli Federico II
Via Claudio 21, 80125, Naples, Italy
[2] Laboratorio CINI-ITEM "Carlo Savy", Complesso Universitario Monte Sant'Angelo, Ed. 1
Via Cinthia, 80126, Naples, Italy
cotroneo@unina.it, antonio.pecchia@unina.it
roberto.pietrantuono@unina.it, stefano.russo@unina.it

ABSTRACT

Middleware plays a strategic role to reduce development cost and time to market. However, it raises significant dependability challenges when integrated in complex, mission-critical systems. Testing activities, carried out during the development of middleware platforms, may be not enough to assure a proper dependability level after their integration. Middleware failures and their impact on the system *as a whole* have to be carefully evaluated in critical scenarios.

This paper reports a practical experience with a real world, middleware-based Air Traffic Control (ATC) system, being developed in the context of an academic-industrial collaboration. Two equivalent middleware subsystems for data distribution have been compared from the dependability point of view. We identify internal dependencies and execution environment resources characterizing both the solutions. By means of an extensive failure modes emulation campaign, we show that these architectural features can significantly affect the middleware and the overall system dependability level.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*reliability, availability, and serviceability*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*

General Terms

Experimentation, Performance

Keywords

Software Dependability, Air Traffic Control, Data Distribution Service, Failure Mode Emulation

MW4SOC '09, November 30, 2009, Urbana Champaign, Illinois, USA

1. INTRODUCTION

Middleware solutions address relevant design challenges for developing software systems. They enable complex applications to be designed by integrating services and components rather then building them entirely *from scratch*. This in turn reduces development cost and time to market [10].

The use of middleware in mission critical systems (e.g., avionic and railway systems, monitoring/control of critical infrastructures) raises significant issues. Industry is required to devote a special care to dependability assessment activities [8]. Domain-specific standards for critical systems (e.g., [1, 5, 3, 2]) exhaustively describe tasks and evidences to be produced during all the phases of the system life-cycle, which aim to *assure* a proper dependability level for each system component.

Middleware designers are not always aware of this process, thus ignoring the actual needs of critical software development. Middleware platforms are usually delivered as Off-The-Shelf (OTS) items, which have to be *just* linked to the application code (e.g., binary objects for C/C++ programs) without any knowledge of their internals. These items are usually developed having no specific context in mind. Consequently, testing activities carried out during their development, may be not enough to guarantee a proper service during operations [20, 13], because of unforeseeable interactions with the system under development. Middleware failures and their impact on the rest of the system have to be carefully evaluated in critical scenarios.

This paper reports a practical experience with two middleware solutions for the Data Distribution Service (DDS), which have been compared from the dependability point of view. The experience has been conducted on a real-world case study in the field of Air Traffic Control (ATC). A world leading company, SELEX-SI, along with academic partners involved in the COSMIC [1] research project, have developed a novel ATC system allowing supporting personnel to monitor and to control planes in a specific air space.

The need to select and to integrate a suitable DDS middleware in this system is the reason why we compared two functionally equivalent solutions. To this aim, for each solution (i) we identified execution environment resources and their dependencies with the middleware; (ii) we performed

[1]COSMIC is a three-year Italian industrial research project aiming to create a public-private research laboratory for development of a open source middleware platform for mission critical systems

an extensive failure modes emulation campaign. We evaluated the impact of resources failures on middleware components and on the system as a whole. Results show how it has been possible to identify architectural dependencies and resources mainly affecting the overall dependability in both cases, thus driving the final choice taken by the project team.

The rest of this paper is organized as follows. Section 2 presents standards and related work in the area of dependability evaluation. Section 3 describes the system in hand and how experiments have been conducted. Section 4 provides experimentation results while Section 5 concludes the work.

2. RELATED WORK

Dependability assessment and evaluation of complex critical systems is a challenging issue. Different organizations carried out standards and methodologies supporting the development of dependable systems. They define tasks and evidences to be produced during all the phases of the development cycle. Examples are the CENELEC guidelines [1], in the field of railway applications, IEC 61508 [5], for electrical/electronic systems, and the DO-178B [3] for ATC systems, which has heavily influenced subsequent ones [12]. In the European scenario, EUROCONTROL combined several standards, such as DO-178B and IEC 61508, and defined a methodology to provide guidelines for the safety assessment of ATC systems, i.e., the Safety Assessment Methodology (SAM) [2].

Hazard analysis and risk assessment, such as failure modes and effects analysis (FMEA), hazard and operability (HAZOP) and fault tree analysis [18], are the most adopted techniques. In [11] and [15] authors described the hazard analysis methodology defined and used in railway dependable systems. Robustness testing is also commonly adopted to give insight about the system ability to tolerate exceptional environmental conditions [6, 4].

Several works proposed approaches based on the dynamic flow graph methodology (DFM) [9] for assessing risks associated with dynamic behaviors. Other works proposed methodologies and/or technologies for safety assessment of real complex critical system infrastructures. In [19] a case study for car security is presented.

Cited works neglect dependencies among system components and execution environment resources. Moreover, risk assessment is often performed by only examining faults at interface level. The focus of our experience is to evaluate how resource failures propagate within system components (i.e., the middleware layer) and affect the overall dependability level. Analyzing these phenomena is a *must* when coping with mission-critical scenarios.

3. CASE STUDY

3.1 ATC Application

Our case study consists of a real world scenario in the field of ATC. In particular, we consider a **Flight Data Plans (FPLs) Processor**, developed on the top of a middleware platform, named CARDAMOM [2]. A FPL provides information such as the flight route, the current trajectory, airplane-related information, and meteorological data.

CARDAMOM provides several services. These include, among others, the Load Balancer (LB), Replication (R), and System Management (SMG) services, used by the application in hand. The application also exploits the OMG-compliant[3] DDS [14]. DDS allows application components to share FPLs. This is done by means of `read` and `write` facilities provided by the DDS API, which allow to retrieve and to publish a FPL instance, respectively.

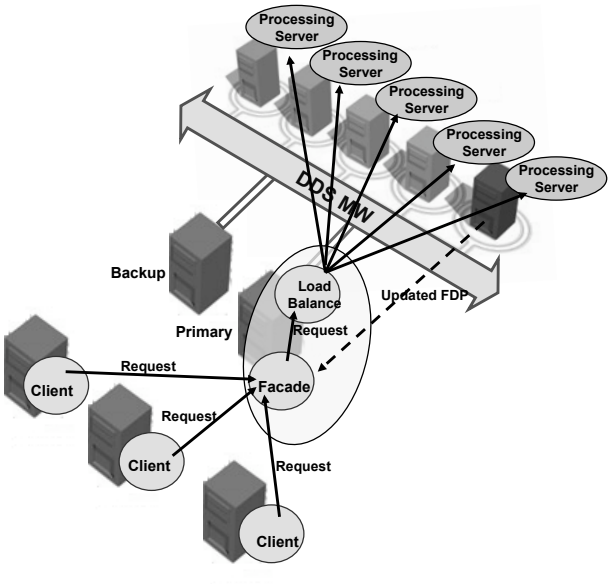

Figure 1: Experimental scenario

Figure 1 depicts the FPLs Processor. It is a CORBA-based distributed object system. It is composed by the **Facade** object and a set of **Processing Servers** managed by the LB service. Facade accepts FPL processing requests (i.e., insert, delete, update) supplied by external **Clients** and guarantees data consistency by means of mutual exclusion among requests accessing the same FPL instance. Facade subsequently redirects each allowed request to 1 out of N Processing Server, according to the *round robin* service policy. The selected server (i) retrieves the specified FPL instance from the DDS middleware (i.e., DDS MW in Figure 1) by means of the `read` facility (ii) executes request-related computations, and (iii) returns the updated FPL instance to the Facade object. Facade publishes the updated FPL instance by means of the DDS `write` facility and finalizes the request.

Machines composing the application testbed (Intel Pentium 4 3.2 GHz, 4 GB RAM, 1,000 Mb/s Network Interface equipped) run a RedHat Linux Enterprise 4. An Ethernet LAN interconnects these machines. Client objects continuously invoke Facade services with an average frequency of 50 requests per second. About 4,000 FPLs instances, each of them of 77,812 bytes, are shared with the DDS MW.

[2]CARDAMOM is a CORBA-based middleware platform developed by Thales and Selex-SI, tailored to support the development of software architectures for safety and mission critical systems

[3]OMG specification for the Data Distribution Service, `http://www.omg.org`

Table 1: Shared Memory

Failure-mode	Emulation
`access denied`	the `vm_area_struct` related to the target shared memory is deleted from the addressing space of the process
`read denied`	bits storing the memory access policies are modified by interacting with the OS paging sub-system [17]
`write denied`	bits storing the memory access policies are modified by interacting with the OS paging sub-system [17]
`corruption`	bit-flip technique. We perform distinct experiments by flipping a *single bit* or a *bit sequence* of different, increasing, sizes (i.e., 10, 100, 1,000, 10,000, 100,000)

3.2 Experiments

DDS MW plays a key role in the described system both because of (i) the critical scenario (i.e., the ATC domain) and (ii) the workload. Choosing a specific DDS implementation may result in a different overall system dependability level. We demonstrate this with a practical experience with two DDS implementations coming from different vendors. For reasons of confidentiality we do not disclose the actual middleware names used in the experiments. To this aim, let **DDS_1** and **DDS_2** denote the two DDS MW implementations. We also use the ***** wild card, when needed, to avoid any possible platform-related reference.

The aim of the experimentation is twofold (i) to evaluate how threats coming form the actual execution environment affect each considered DDS implementation (ii) to analyze if/how possible DDS outages compromise the mission of the ATC application.

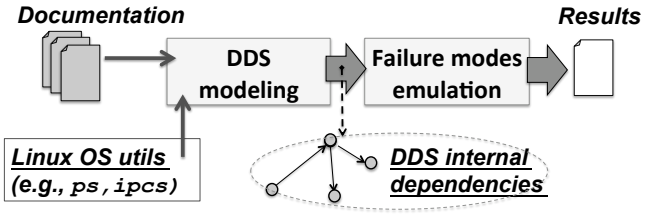

Figure 2: Experimentation phases

Figure 2 depicts how experiments have been conducted. We preliminary model each considered DDS implementation. Exploiting the available DDS documentation is the way we do this. Further insight has been achieved by using Linux command-line utils (e.g., `ps`, `ipcs`). Result of this modeling phase is a *high-level* picture of the DDS in terms of (i) operating system processes encapsulating DDS code, (ii) external environment resources (i.e., Inter-Process Communication (IPC), network) and (iii) functional dependencies among them.

This information is fed to the failure modes emulation phase. In this phase we investigate if/how external resources outages affect processes encapsulating the DDS code during operations. We also analyze if outages compromise the mission of the applicative ATC layer. We inject errors rather than faults to possibly reduce experimentation time [7]. In order to correctly evaluate the system dependability, we make proper failure assumptions on *how* a system component can fail [16]. Experiments take into account the following failure modes for resources:

- `access denied`: the resource becomes unavailable;
- `read denied`: the resource is accessed, but it can not satisfy a *reading* request;
- `write denied`: the resource is accessed, but it can not satisfy a *writing* request;
- `corruption`: the resource content is altered.

We tailor the described failure-modes to resources used by the considered DDS implementations in the context of the Linux OS. Tables 1 and 2 show how failure modes have been emulated for shared memories and semaphores, respectively. These failures have been injected during system operational time by *ad hoc* kernel modules. For network failures, the only `access denied` has been emulated. Details can be found in Table 3. We emulate a single failure mode for each experiment.

Table 2: Semaphore

Failure-mode	Emulation
`access denied`	target semaphore is deleted with `ipcrm` command-line util
`read denied`	access permissions are modified with `semctl`. `200` is set as new value
`write denied`	access permissions are modified with `semctl`. `400` is set as new value
`corruption`	target semaphore content is modified with `semctl` and the `SETVAL` flag

Table 3: Network

Failure-mode	Emulation
`access denied`	Network is made unavailable in two different ways (i) via the `/sbin/ifconfig eth0 down` command (ii) network interface disconnection.
`read denied`	*not meaningful for the case study.*
`write denied`	*not meaningful for the case study.*
`corruption`	negligible for the case study. A dedicated LAN environment interconnects testbed machines.

4. RESULTS

We evaluate if/how resource failure modes (emulated according to Tables 1, 2, 3) compromise the DDS MW and the overall system dependability level. To this aim we focus on a single pair Facade/Processing Server and we perform experiments as described in Section 3.2. Results are described in the following.

4.1 DDS_1

DDS_1 consists both of a shared library (i.e., `.so` extension in Linux OS) to be linked to the application and middle-

ware processes. Applicative processes (i.e., Facade and Processing Server, respectively) communicate with DDS internal ones by means of a shared memory (Figure 3). Middleware processes are responsible for the communication among the computing nodes of a domain. Let D_A and D_B be these processes. It is possible to identify three resources used by the DDS, i.e., the *shared memories* on the Facade and Processing Server nodes (named M_A and M_B, respectively) and the *network* (named N). Figure 3 depicts interactions needed to transmit data (i.e., FPLs) between the computing nodes.

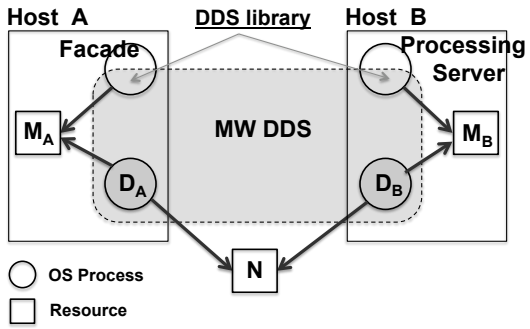

Figure 3: DDS_1 model

Facade and D_A processes communicate through $\underline{M_A}$. As we expected, Facade crashes with a "`segmentation fault`" message in case of an `access`, `read`, or `write denied`. This is due to the nature of the Linux OS paging subsystem. M_A corruption has different consequences, depending on the modified bits. In particular, Facade enters a hang state if the corruption affect lower M_A bits, a crash one, otherwise. Similarly, we evaluate the robustness of D_A with respect to M_A failure modes. D_A crashes for each emulated failure. Its crash does not directly affect Facade, which keeps on running, but updated FPL instances are not forwarded to Processing Servers anymore. No error notification is returned to the applicative layer. We conclude that *M_A failures always compromise the mission of the ATC system.* DDS_1 does not tolerate any of the injected failures.

Processing Server and D_B processes communicate through $\underline{M_B}$. As we expected, Processing Server crashes with a "`segmentation fault`" message in case of an `access`, `read`, or `write denied`. M_B corruption does not make Processing Server to hang or to crash. Anyway it losses every subsequent FPL update. Similarly, we evaluate the robustness of D_B process with respect to M_B failure modes. D_B crashes for each emulated failure. Its crash does not affect Processing Server, which keeps on running, but updated FPL instances are not forwarded to applicative layer anymore. No error notification is returned to Processing Server. We conclude that *M_B failures always compromise the mission of the ATC system.* DDS_1 does not tolerate any of the injected failures.

D_A and D_B processes communicate through $\underline{N}$. When we emulate network unavailability with both the proposed mechanisms, updated FPL instances are lost. Both D_A and D_B do not exhibit any error notification. Communication between nodes remains compromised even when network is restored. *N unavailability compromises the mission of the system*, and DDS_1 does not tolerate *transient* failures.

4.2 DDS_2

DDS_2 exhibits a different architecture (Figure 4). The overall DDS internal code is mapped into applicative processes (i.e., Facade and Processing Server, respectively) by means of a shared library. Internal threads communicate through *shared memories* and *semaphores*. Let M_A, S_{A1}, S_{A2} and M_B, S_{B1}, S_{B2} be shared memories and semaphores at Facade and Processing Server side, respectively. Let N be the *network*. Figure 4 depicts interactions needed to transmit data (i.e., FPLs) between the computing nodes.

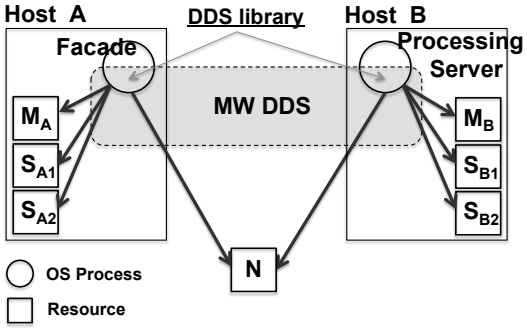

Figure 4: DDS_2 model

Facade process relies on $\underline{M_A}$, $\underline{S_{A1}}$ and $\underline{S_{A2}}$. As we expected, it crashes with a "`segmentation fault`" message in case of an `access`, `read`, or `write denied` . M_A corruption does not compromise both the Facade behaviour and data transmission (i.e., each subsequent DDS `write` invocation correctly succeeds). The improper memory change is notified by the following message "*`*_Transport_Shmem_attach_writer: incompatible shared memory segment found. All applications using * must use compatible shared memory protocols`*". This warning message is periodically triggered by the DDS library every 10 seconds and it is subsequently printed on the Facade console. We can conclude that *M_A failures do not always compromise the mission of the ATC system.* DDS_2 tolerates resource corruption and a notification of the faulty state is also experienced.

$\underline{S_{A1}}$ and $\underline{S_{A2}}$ `access`/`read denied`, and `corruption` do not affect Facade operations. S_{A1} and S_{A2} `write denied` are notified with the following messages "*`* Mutex_lock: OS semop() failure error OXD. *_send: !take semaphore`*", and "*`* Mutex_ive: OS semctl() failure error OXD. *_send:!give semaphore.`*", respectively. These warning messages are periodically triggered by the DDS library every 10 seconds and are subsequently printed on the Facade console. We conclude that *S_{A1} and S_{A2} failures do not compromise the mission of the ATC system.* DDS_2 tolerates, and possibly notifies, emulated failures.

Processing Server relies on $\underline{M_B}$, $\underline{S_{B1}}$ and $\underline{S_{B2}}$. Emulated failure modes result in a similar finding with respect to the Facade side. DDS_2 tolerates, and possibly notifies, emulated failures.

Facade and Processing Server communicate through $\underline{N}$. When we emulate network unavailability with any of the proposed mechanisms, updated FPL instances get lost. Both processes do not exhibit any explicit notification. Communication between them is restored when network is resumed. *N unavailability compromises the mission of the system*, anyway DDS_2 tolerates *transient* failures.

4.3 Comparison

The following findings come out from the experiments, with respect to the emulated failure modes. Both DDS MW use **shared memories**. In DDS_1 they have a *critical* role. Communications occur through the shared memory both at the Facade and Processing Server side. It is a *single point of failure* since each emulated failure compromises data transmission. DDS_2 uses shared memories too. Anyway, in this case they are only support facilities. Their corruption does not compromise data transmission.

DDS_1 does not use **semaphores**, thus avoiding the introduction of new potential failure sources. Anyway they do not represent an actual dependability threat in DDS_2. Even if they are used to access resources, their failures do not compromise the service completion.

Network unavailability affects both DDS MW. Anyway, we experienced that DDS_2 is robust to transient failures. As a matter of fact, when the network is restored, data transmission resumes functioning.

By concluding, the modularity of the daemon-based architecture of DDS_1 seems to not cope properly with critical scenarios. Furthermore, delegating FPLs transmission to external processes, may lead to an improper service without experiencing any applicative outage. DDS_2, even if less modular (i.e., the overall DDS code is mapped into the applicative process), exihibits a higher dependability level, thus making its choice suitable for our ATC scenario.

5. LESSONS LEARNED

Middleware solutions are often required for distributed critical applications, since they allow to notably lower development costs. However, as also shown by the reported experience, their adoption needs to be carefully evaluated from the *dependability perspective*. Even if a middleware platform is well-tested during its development, its use in a specific execution context may be not guaranteed to meet the system dependability goals, due to possible unexpected interactions with the execution environment. In particular, we observed that:

- *the overall system dependability level strongly depends on dependencies among middleware internal components.* Internal middleware architecture and interactions among middleware processes result in a different ability to react to failures;
- *inferring relationships (i) among middleware internal components and (ii) between these components and execution environment resources, is useful to provide a view of the system dependability level and to drive the middleware selection process.* Experiments show that distinct architectures result in different fault tolerance features with respect to the execution environment malfunctioning;
- our experience shows that the modular architecture of DDS_1, may be not the best choice from the dependability perspective. This highlights that *dependability may conflict, other than with performance requirements, also with features such as a higher degree of modularity and flexibility.*

In the future, we aim to extend experiments to other application fields. As for example, we aim to compare in terms of dependability, other than different architectures, different execution environments. Moreover, we aim to consider different kinds of OS/middleware resources as well as to expand the considered failure modes, in order to increase experimentation accuracy.

6. ACKNOWLEDGMENTS

This work has been partially supported by the Consorzio Interuniversitario Nazionale per l'Informatica (CINI) and by the Italian Ministry for Education, University, and Research (MIUR) within the frameworks of the "Centro di ricerca sui sistemi Open Source per le applicazioni ed i Servizi MIssion Critical" (COSMIC) project (`www.cosmiclab.it`), the "Iniziativa Software" Project (`www.iniziativasoftware.it`), and the "Tecnologie Orientate alla Conoscenza per Aggregazioni di Imprese in InterneT" (TOCAI.IT) FIRB Project .

7. REFERENCES

[1] CENELEC: EN 50126 Railways Applications. The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS).

[2] SAF.ET1.ST03.1000-MAN-01. Air Navigation System Safety Assessment Methodology (v2-0). . *EUROCONTROL EATMP Safety Management*, Apr. 2004.

[3] DO-178B/ED12B. Software consideration in airborne systems and equipment certification. *RTCA and EUROCAE*, Dec. 1992.

[4] The Ballista Robustness Testing Service. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/edrcballista/www/index.html.

[5] Functional safety and IEC 61508. Functional safety of electrical/electronic/programmable electronic safety-related systems. *Produced by IEC/SC65A/WG14, The working group responsible for guidance on IEC 61508*, Sep. 2005.

[6] J. Campelo, J. Pardo, and J. Serrano. Robusteness study of an embedded operating system for industrial applications. In *Proc. of the 28th International Computer Software and Application Conference (COMPSAC 2004)*, pages 27–30, Sept. 2004.

[7] J. Christmansson and R. Chillarege. Generation of an error set that emulates software faults based on field data. In *Proc. of the 26th Annual International Symposium on Fault-Tolerant Computing (FTCS '96)*, 1996.

[8] K. Fowler. Mission-Critical and Safety-Critical Development. *IEEE Instrumentation and Measurement Magazine*, Dec. 2004.

[9] C. Garrett and G. Apostolakis. Automated hazard analysis of digital control systems. In *Reliability Engineering and System Safety*, pages Vol. 77, pp. 1–17, 2002.

[10] R. Hammett. Flight-Critical Distributed Systems: Design Considerations. *IEEE AESS Systems Magazines*, pages Vol. 18, Issue 6, 30–36, 2003.

[11] A. Hassami and A. Foord. Systems safety-a real example (european rail traffic management system, ertms). In *the Second IEEE International Conference on Human Interfaces in Control Rooms, Cockpits and Command Centres*, pages 327–334, 2001.

[12] E. Kesseler. Air Transport, from Privilege to Commodity. Technical report (NLR-TP-2003-300). *INLR (the National Aerospace Laboratory of the Netherlands)*, 2003.

[13] R. L. O. Moraes, J. Durães, R. Barbosa, E. Martins, and H. Madeira. Experimental Risk Assessment and Comparison Using Software Fault Injection. In *Proc. of the 37th Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 512–521. IEEE Computer Society, 2007.

[14] G. Pardo-Castellote. OMG data-distribution service: Architectural overview. In *ICDCS Workshops*, pages 200–206. IEEE Computer Society, 2003.

[15] T. Pasquale, E. Rosaria, M. Pietro, and O. Antonio. Hazard analysis of complex distributed railway systems. In *the 22nd IEEE International Symposium on Reliable Distributed Systems (SRDS'03)*, pages 283–292, Oct. 2003.

[16] D. Powell. Failure mode assumptions and assumption coverage. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing (FTCS '92)*, 1992.

[17] A. Rubini and J. Corbet. *Linux Device Drivers*. 2nd Edition, O'Reilly, 2001.

[18] N. Storey. Safety-Critical Computer Systems. *Pearson and Prentice Hall, 1996*.

[19] S. Supakkul and C. Lawrence. Applying a goal-oriented method for hazard analysis: A case study. In *the 4th International Conference on Software Engineering Research, Management and Applications (SERA'06)*, pages pp. 22– 30, Aug. 2006.

[20] E. Weyuker. Testing Component-Based Software: A Cautionary Tale. *IEEE Software*, 15(5):54–59, 1998.