

HOLGER GIESE (ED.)

LNCS 6150

ARCHITECTING CRITICAL SYSTEMS

FIRST INTERNATIONAL SYMPOSIUM, ISARCS 2010
PRAGUE, CZECH REPUBLIC, JUNE 2010
PROCEEDINGS



Springer

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

TU Dortmund University, Germany

Madhu Sudan

Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max-Planck Institute of Computer Science, Saarbruecken, Germany

Holger Giese (Ed.)

Architecting Critical Systems

First International Symposium, ISARCS 2010
Prague, Czech Republic, June 23-25, 2010
Proceedings



Springer

Volume Editor

Holger Giese

Hasso Plattner Institute for Software Systems Engineering

Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany

E-mail: Holger.Giese@hpi.uni-potsdam.de

Library of Congress Control Number: 2010928429

CR Subject Classification (1998): C.3, K.6.5, D.4.6, E.3, H.4, D.2

LNCS Sublibrary: SL 4 – Security and Cryptology

ISSN 0302-9743

ISBN-10 3-642-13555-2 Springer Berlin Heidelberg New York

ISBN-13 978-3-642-13555-2 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

springer.com

© Springer-Verlag Berlin Heidelberg 2010

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper 06/3180

Preface

Architecting critical systems has gained major importance in commercial, governmental and industrial sectors. Emerging software applications encompass criticalities that are associated with either the whole system or some of its components. Therefore, effective methods, techniques, and tools for constructing, testing, analyzing, and evaluating the architectures for critical systems are of major importance. Furthermore, these methods, techniques and tools must address issues of dependability and security, while focusing not only on the development, but also on the deployment and evolution of the architecture.

This newly established ISARCS symposium provided an exclusive forum for exchanging views on the theory and practice for architecting critical systems. Such systems are characterized by the perceived severity of consequences that faults or attacks may cause, and architecting them requires appropriate means to assure that they will fulfill their specified services in a dependable and secure manner.

The different attributes of dependability and security cannot be considered in isolation for today's critical systems, as architecting critical systems essentially means to find the right trade-off among these attributes and the various other requirements imposed on the system. This symposium therefore brought together the four communities working on dependability, safety, security and testing/analysis, each addressing to some extent the architecting of critical systems from their specific perspective. To this end the symposium united the following three former events:

- Workshop on Architecting Dependable Systems (WADS)
- Workshop on the Role of Software Architecture for Testing and Analysis (ROSATEA)
- Workshop on Views on Designing Complex Architectures. (VODCA)

The 27 submissions and 11 published papers of this first ISARCS instance in 2010 show that we brought together as planned expertise from the different communities and therefore were able to provide a first overarching view on the state of research on how to design, develop, deploy and evolve critical systems from the architectural perspective.

The selected papers addressed issues such as rigorous development, testing and analysis based on architecture, fault tolerance based on the architecture, safety-critical systems and architecture, secure systems and architecture, combined approaches and industrial needs.

In the symposium the design of critical systems was addressed looking at issues such as analyzing the trade-offs between security and performance, architectural design decisions for achieving reliable software systems, and the integration of fault-tolerance techniques into the design of critical systems. In addition, also more rigorous approaches to design were discussed.

The assurance of critical systems was discussed for approaches that employ formal methods and testing for applications as well as underlying software layers. In addition, a number of results that target specific domains such as military systems, safety-critical product lines and peer-to-peer control and data acquisition systems were presented. These papers provided a good introduction into the specific requirements of these domains and presented specific solutions for their domain. Furthermore, the interplay of architecture modeling and existing domain-specific safety standards was discussed in the context of automotive systems.

The program was completed by two keynotes that were shared with the other events of the federated CompArch conference. The first one was on a component-based approach for adaptive user-centric pervasive applications from Martin Wirsing from the Ludwig-Maximilians-Universität Munich, Germany, and the second addressed how to make the definition of evolution intrinsic to architecture descriptions, by Jeff Magee from the Imperial College, London, UK.

I thank the authors of all submitted papers, and the PC members and external referees who provided excellent reviews. I am in particular grateful to Frantisek Plasil and the whole team in Prague as well as Stefan Neumann and Edgar Nähter for their help and support concerning organizational issues. I furthermore thank the ISARCS SC members for their support throughout the whole process and their strong commitment to making ISARCS 2010 a success.

April 2010

Holger Giese

Organization

ISARCS 2010 was organized by the Faculty of Mathematics and Physics of the Charles University, Prague, Czech Republic as one event of the federated conference Component-Based Software Engineering and Software Architecture (CompArch 2010).

General Chair

Frantisek Plasil Charles University, Prague, Czech Republic

Program Chair

Holger Giese Hasso Plattner Institute at the University of
Potsdam, Germany

Local Organization

Petr Hnětynka Charles University, Prague, Czech Republic
Milena Zeithamlova Action M Agency, Prague, Czech Republic

Steering Committee

Rogério de Lemos University of Coimbra, Portugal)
Cristina Gacek City University, London, UK
Fabio Gadducci University of Pisa, Italy
Lars Grunske Swinburne University of Technology, Australia
Henry Muccini University of L'Aquila, Italy
Maurice ter Beek ISTI-CNR, Pisa, Italy

Program Committee

Alessandro Aldini University of Urbino, Italy
Aslan Askarov Cornell University, USA
Brian Berenbach Siemens Corporate Research, USA
Stefano Bistarelli Università di Perugia, Italy
Michel R.V. Chaudron Leiden University, The Netherlands
Betty H. C.Cheng Michigan State University, USA
Nathan Clarke University of Plymouth, UK
Ricardo Corin Universidad Nacional de Cordoba (FAMAF),
Argentina

VIII Organization

Cas Cremers	ETH Zurich, Switzerland
Ivica Crnkovic	Mälardalen University, Sweden
Bojan Cukic	West Virginia University, USA
Eric Dashofy	The Aerospace Corporation, USA
Erik de Vink	Eindhoven University of Technology, The Netherlands
Heiko Dörr	Carpeq GmbH, Germany
Alexander Egyed	Johannes Kepler University, Austria
Sébastien Gérard	CEA LIST, France
Wolfgang Grieskamp	Microsoft Corporation, USA
Ethan Hadar	CA Inc., Israel
Paola Inverardi	University of L'Aquila, Italy
Valérie Issarny	INRIA, UR de Rocquencourt, France
Tim Kelly	University of York, UK
Marc-Olivier Killijian	LAAS-CNRS Toulouse, France
Philip Koopman	Carnegie Mellon University, USA
Patricia Lago	VU University Amsterdam, The Netherlands
Javier Lopez	University of Malaga, Spain
Nenad Medvidovic	University of Southern California, USA
Flavio Oquendo	European University of Brittany - UBS/VALORIA, France
Mauro Pezzè	University of Lugano, Switzerland
Ralf H. Reussner	Karlsruhe Institute of Technology / FZI, Germany
Roshanak Roshandel	Seattle University, USA
Ana-Elena Rugina	Astrium Satellites, France
Bradley Schmerl	Carnegie Mellon University, USA
Bran Selic	Malina Software, Canada
Judith Stafford	Tufts University, USA
Michael von der Beeck	BMW Group, Germany

External Referees

Rogério de Lemos
Lars Grunske
Aaron Kane
Giovanni Mainetto
Mohamad Reza Mousavi
Henry Muccini
Marinella Petrocchi
Justin Ray
Francesco Santini
Malcolm Taylor
Maurice H. ter Beek

Table of Contents

Design

An Architectural Framework for Analyzing Tradeoffs between Software Security and Performance	1
<i>Vittorio Cortellessa, Catia Trubiani, Leonardo Mostarda, and Naranker Dulay</i>	
Architectural Design Decisions for Achieving Reliable Software Systems	19
<i>Atef Mohamed and Mohammad Zulkernine</i>	
Integrating Fault-Tolerant Techniques into the Design of Critical Systems	33
<i>Ricardo J. Rodríguez and José Merseguer</i>	
Component Behavior Synthesis for Critical Systems	52
<i>Tobias Eckardt and Stefan Henkler</i>	

Verification and Validation

A Road to a Formally Verified General-Purpose Operating System	72
<i>Martin Děcký</i>	
Engineering a Distributed e-Voting System Architecture: Meeting Critical Requirements	89
<i>J. Paul Gibson, Eric Lallet, and Jean-Luc Raffy</i>	
Testing Fault Robustness of Model Predictive Control Algorithms	109
<i>Piotr Gawkowski, Konrad Grochowski, Maciej Ławryńczuk, Piotr Marusak, Janusz Sosnowski, and Piotr Tatjewski</i>	

Domain-Specific Results

Towards Net-Centric Cyber Survivability for Ballistic Missile Defense	125
<i>Michael N. Gagnon, John Truelove, Apu Kapadia, Joshua Haines, and Orton Huang</i>	
A Safety Case Approach to Assuring Configurable Architectures of Safety-Critical Product Lines	142
<i>Ibrahim Habli and Tim Kelly</i>	
Increasing the Resilience of Critical SCADA Systems Using Peer-to-Peer Overlays	161
<i>Daniel Germanus, Abdelmajid Khelil, and Neeraj Suri</i>	

Standards

ISO/DIS 26262 in the Context of Electric and Electronic Architecture Modeling	179
<i>Martin Hillenbrand, Matthias Heinz, Nico Adler, Klaus D. Müller-Glaser, Johannes Matheis, and Clemens Reichmann</i>	
Author Index	193

An Architectural Framework for Analyzing Tradeoffs between Software Security and Performance

Vittorio Cortellessa¹, Catia Trubiani¹, Leonardo Mostarda²,
and Naranker Dulay²

¹ Università degli Studi dell'Aquila, L'Aquila, Italy
{vittorio.cortellessa,catia.trubiani}@univaq.it

² Imperial College London, London, United Kingdom
{lmostard,nd}@doc.ic.ac.uk

Abstract. The increasing complexity of software systems entails large effort to jointly analyze their non-functional attributes in order to identify potential tradeoffs among them (e.g. increased availability can lead to performance degradation). In this paper we propose a framework for the architectural analysis of software performance degradation induced by security solutions. We introduce a library of UML models representing security mechanisms that can be composed with performance annotated UML application models for architecting security and performance critical systems. Composability of models allows to introduce different security solutions on the same software architecture, thus supporting software architects to find appropriate security solutions while meeting performance requirements. We report experimental results that validate our approach by comparing a model-based evaluation of a software architecture for management of cultural assets with values observed on the real implementation of the system.

Keywords: performance, security, UML, GSPN, tradeoff analysis.

1 Introduction

The problem of modeling and analyzing software architectures for critical systems is usually addressed through the introduction of sophisticated modeling notations and powerful tools to solve such models and provide feedback to software engineers.

However, non-functional attributes are often analyzed in isolation. For example, performance models do not usually take into account the safety of a system, as well as availability models do not consider security aspects, and so on. An early but relevant exception in this domain has been the definition of performability [19] that combines performance and availability aspects into the same class of models. With the increasing variety and complexity of computing platforms, we believe that the task of jointly analyzing non-functional attributes to study

possible dependencies is becoming a critical task for the successful development of software architectures.

This paper works towards this goal. It presents a framework to jointly model and analyze the security and performance attributes of software architectures. The critical aspect that we tackle is to quantify the performance degradation incurred to achieve the security requirements. The basic idea is that the solution of a performance model that embeds security aspects provides values of indices that can be compared to the ones obtained for the same model (i) without security solutions, (ii) with different security mechanisms and (iii) with different implementations of the same mechanism. Such comparisons help software architects to decide if it is feasible to introduce/modify/remove security strategies on the basis of (possibly new) requirements.

Security is, in general, a complex and cross-cutting concern, and several mechanisms can be used to impact on it. In this paper we focus on two common mechanisms that are: encryption and digital signature.

In order to easily introduce security and performance aspects into software models we have built a library of models that represent security mechanisms ready to be composed. Once an application model is built, in order to conduct a joint analysis of security and performance with our approach it is necessary for the software designer: (i) to specify the appropriate security annotations (e.g. the confidentiality of some data), and (ii) to annotate the model with performance related data (e.g. the system operational profile). Thereafter, such an annotated model can be automatically transformed into a performance model whose solution quantifies the tradeoff between security and performance in the architecture under design. The setting where our approach works is Unified Modeling Language (UML) [1] for software modeling and Generalized Stochastic Petri Nets (GSPN) [15] for performance analysis.

The starting point of this work can be found in [5], where we have introduced an earlier version of this approach and a preliminary security library of models expressed as performance models. As envisaged in [5], thereafter we have applied the approach to real world case studies, and the experimentation phase led us to modify the approach as well as the security library.

An important aspect lacking in [5] that we tackle in this paper is that the choice and the localization of the appropriate security mechanisms should be driven from the system architecture (e.g. features of the communication channels, physical environment, etc.). For example, a message exchanged between two components might require encryption depending on whether the communication channel between the components is a wireless one or not (see Section 5). Hence, the main progress, in comparison to [5], is that here we express security mechanisms as UML architectural models, and the model composition is moved on the designer's side. Thus, our approach allows designers to explicitly explore architectural alternatives for balancing security/performance conflicting concerns in the architectural phase of the development process.

To validate our approach we have conducted extensive experiments where we compare the results of our models with the data monitored on a real system.

The promising numerical results that we have obtained significantly support the prediction capabilities of our approach.

The paper is organized as follows: in Section 2 we present the related work and describe the novelty of the approach with respect to the existing literature; in Section 3 we introduce our approach and the types of analysis that it can support; in Section 4 we discuss the library of security models and how it is used at the application level; in Section 5 we present the experimental results that we have obtained on a real case study; finally in Section 6 we give concluding remarks and future directions.

2 Related Work

The literature offers a wide variety of proposals and studies on the performance aspects of security, but most of them analyze the performance of existing standards such as IPsec and SSL. Therefore the analysis conducted in these approaches remains confined to very specific problems and solutions. However, noteworthy examples in this direction can be found in [4] [9] [14].

A tradeoff analysis of security and scalability can be found in [16], which stresses the importance of understanding the security required while minimizing performance penalties. Our concern is similar to this one because we also target an analysis of how security solutions impact on system performance. However, in [16] the analysis is conducted using a specific security protocol (i.e. SSL) and a limited set of cryptographic algorithms, whereas our framework is intended to model and analyze more general solutions.

Estimating the performance of a system with different security properties is a difficult task, as demonstrated in [12], where they emphasize how difficult is to choose between secure and non-secure Web services, RMI and RMI with SSL. The authors solve the problem by performing different measurements on different platforms to elicit guidelines for security setting selections.

There are also several works that use aspect-oriented modeling (AOM) to specify and integrate security risks and mechanisms into a system model, such as the one in [8]. An interesting performance study, based on AOM, can be found in [22] where security solutions are modeled as aspects in UML that are composed with the application logics, and the obtained model is transformed into a performance model. This work uses an approach to the problem that is similar to ours, in that they are both based on model annotations and transformations. Our work, at some extent, refines the approach in [22] because we target the problem of representing elementary security mechanisms aimed at guaranteeing certain security properties, whereas the analysis in [22] is performed only on the SSL protocol and a set of properties embedded in it. Furthermore this paper differs from [22] because for each security mechanism we additionally consider the reasonable implementation options (e.g. the key length of an encryption algorithm) that significantly impact on the software performance.

The lack of a model-based architectural solution to this problem is the major motivation behind our work. This paper aims at overcoming the limitations of ad

hoc solutions that estimate the performance of specific security technologies. To achieve this goal, we propose a framework that manages and composes platform-independent models of security mechanisms, thus allowing designers to estimate and compare the performance of different architectural security solutions for critical systems.

3 Our Approach

In this section we present a framework that allows us to quantify the tradeoff between the security solutions introduced to cope with the application requirements and the consequent performance degradation.

In Figure 1 the process that we propose is reported. The process has been partitioned in two sides: on the left side all models that can be represented with a software modeling notation (e.g. UML) appear; on the right side all models represented with a performance modeling notation (e.g. GSPN) appear.

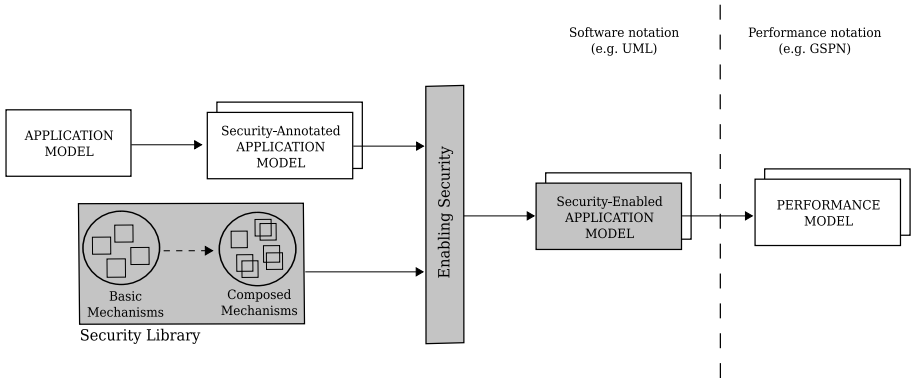


Fig. 1. A joint process for security and performance goals

The starting point of the process is an *Application Model* that is a static and dynamic representation of a software architecture. For sake of simplification we assume that such model is annotated with the performance parameters related to the application (e.g. the system operational profile¹).

A *Security-Annotated* model is obtained by introducing security annotations in the former. Such annotations specify *where* security properties have to be inserted, namely which software services have to be protected and how (e.g. the entity providing a certain service must be authenticated before using it).

¹ Note that the standard MARTE profile [2] has been adopted to specify performance parameters in our UML models. However, it is out of the scope of this paper to provide details of performance annotations, because well assessed techniques exist for this goal [20].

The contribution of this paper can be located among the shaded boxes of Figure 1. A *Security Library* of UML models is provided; in particular, models of *Basic Mechanisms* are combined to build *Composed Mechanisms*.

The task of *Enabling Security* consists in embedding the appropriate security mechanisms in the software architecture. This step is driven by the security annotations specified in the application model, and a *Security-Enabled Application Model* is finally obtained. As an example, if a security annotation specifies that data integrity must be guaranteed for a certain service, an additional pattern with the steps needed for the data integrity mechanism must be introduced in the architectural model wherever the service is invoked. Such a pattern is one of the mechanisms modeled in our security library.

A key aspect of this task is the composability of models, and this is achieved in our approach through two features: (i) entry points for security mechanisms are unambiguously defined by security annotations, and (ii) mechanism models in the security library have been designed to be easily composable with application models (see Section 4.2).

The security-enabled application model is finally transformed into a GSPN-based *Performance Model*. This step involves not only a transformation between modeling notations 2, but an additional task is necessary to appropriately instrument the target performance model, because security mechanisms inevitably introduce additional performance parameters to be set in the model.

The definition of such parameters is embedded in the security library where they are defined in an application-independent way. For example, the encryption mechanism introduces additional parameters affecting system performance, such as the complexity and resource requirements of the encryption algorithm, its mode of operation (e.g. CBC), the lengths of the keys, etc. However, the task of enabling security implies the usage of such mechanisms at the application level, thus they can be influenced by further application-dependent characteristics. For example, the encryption mechanism efficiency is influenced by the speed of the CPU executing the encryption algorithm, the length of the message to be encrypted, etc.

Hence, the GSPN performance model finally generated has to be carefully parameterized with proper performance data.

The GSPN performance models are solved by SHARPE [11] [21], and the model evaluation provides performance indices that jointly take into account both the security and the performance features required for a critical system.

Note that such tradeoff analysis can be conducted on multiple security settings by only modifying the security annotations and re-running the following steps of our approach. In fact, in Figure 1 we can define a certain multiplicity in the security annotations to emphasize that different strategies can be adopted for the same architecture according to different system settings (see Section 5.1).

² Well consolidated techniques have been exploited to transform software models (e.g. UML diagrams) into performance models (e.g. GSPN), and readers interested can refer to [3] for an extensive survey on this topic.

Finally we observe that several types of analysis can be conducted on the models built with this approach: (i) a performance model with a set of security requirements can be compared with one without security to simply study the performance degradation introduced from certain security settings; (ii) the performance estimates from different performance models can be compared to each other to study the tradeoff between security and performance across different architectural configurations.

Note that the latter analysis can be hierarchically conducted by assuming configurations that are ever more secure. This scenario leads to continuously raising the security settings, thus allowing us to quantify the amount of system performance degradation at each increase of security.

4 Enabling Security

The OSI (Open Systems Interconnection) Security Architecture standard [18] aims at defining, through basic mechanisms and their composition, various properties of a system belonging to a secure environment.

Based on OSI, in [5] we have introduced a set of performance models for Basic and Composed Security Mechanisms. An open issue of that preliminary work was the usage of those models on real applications. After experimenting on real case studies, we realized that the directives in [18] led us to produce, in some cases, models that were too abstract to be useful in practice. This consideration has brought to substantially modify our security models.

In Table 1 a refined set of Basic and Composed Mechanisms, and their dependencies, is illustrated. Each row refers to a Composed Mechanism and each column to a Basic one. An X symbol in a (i, j) cell means that Basic Mechanism j is used to build Composed Mechanism i .

The Basic Mechanisms that we consider are: *Encryption*, which refers to the usage of mathematical algorithms to transform data into a form that is unreadable without knowledge of a secret (e.g. a key); *Decryption*, which is the inverse operation of Encryption and makes the encrypted information readable again; the *Digital Signature* is a well-known security mechanism that has been split into *Generation* and *Verification*, in order to express finer grained dependencies among mechanisms.

Table 1. Dependencies between Basic and Composed Mechanisms

Composed \ Basic	Encryption	Digital Signature Generation	Digital Signature Verification	Decryption
Data Confidentiality	X			X
Data Integrity		X	X	
Peer Entity Authentication		X	X	
Data Origin Authentication		X		

The Composed Mechanisms have been defined as follows. *Data Confidentiality* refers to the protection of data such that only the authorised entity can read it. *Data Integrity* assures that data has not been altered without authorisation. *Peer Entity Authentication* is an identity proof between communicating entities. *Data Origin Authentication* supports the ability to identify and validate the origin of a message; it has been defined as a Composed Mechanism although it depends on only one Basic Mechanism (see Table II). This choice allows to interpret the generation of a digital signature as an high level mechanism that can be used by itself to enable the (possibly future) verification of data origin.

Models of Basic and Composed Mechanisms have been expressed as UML Sequence Diagrams (see Section 4.1).

In [13] an UML profile, called UMLsec, is presented for secure system development. Security properties (i.e. secrecy, integrity, authenticity and freshness) are specified as tagged values of a common stereotype (i.e. data security). We do not use the UMLsec profile because the set of models we consider act at a lower level of abstraction to provide a higher degree of freedom to architectural designers, e.g. about the encryption algorithm and the key lengths. Ultimately we intend to provide instruments for architecting security and performance critical systems on the basis of quantitative estimates.

4.1 Security Library

In this section we concentrate on the security mechanisms identified in Table I. Some preliminary operations, such as the generation of public and secret keys and the process of obtaining a certificate from a certification authority, are executed once by all software entities involved in the security annotations.

In Figure 2(a) the generation of public and private keys is illustrated: a component sets the key type (*setKeyType*) and the key length (*setKeyLength*) and generates the public (*generatePKey*) and the private (*generateSKey*) keys.

In Figure 2(b) the process of obtaining a certificate from a certification authority is shown: a component requiring a certificate (*reqCertificate*) sends its information and the public key; the certification authority checks the credentials (*checkEntityInfo*) and, if trusted, generates the certificate (*generateCertificate*) and sends it back (*sendCertificate*) to the software entity.

Figure 3 shows the UML Sequence Diagram modeling Encryption. Firstly the sender of the message decides the type of algorithm to use (*setAlgorithmType*) and the key length (*setKeyLength*). The encryption can be of two different types: *asymEncrypt* means asymmetric encryption (i.e. by public key), whereas *symEncrypt* indicates symmetric encryption (i.e. by a shared secret key).

For asymmetric encryption the sender sets the padding scheme it requires (*setPaddingScheme*) and verifies the receiver's certificate if it is not already known. Finally, the encryption algorithm (*encryptAlgorithm*) is executed on the message (*msg*) with the public key of the receiver ($P(R)$).

For symmetric encryption the sender sets the algorithm mode (*setAlgorithmMode*), performs a *key-exchange* protocol if a shared key is not already exchanged, and requires the exchange of certificates. We have modelled the ISO/IEC 11770-3

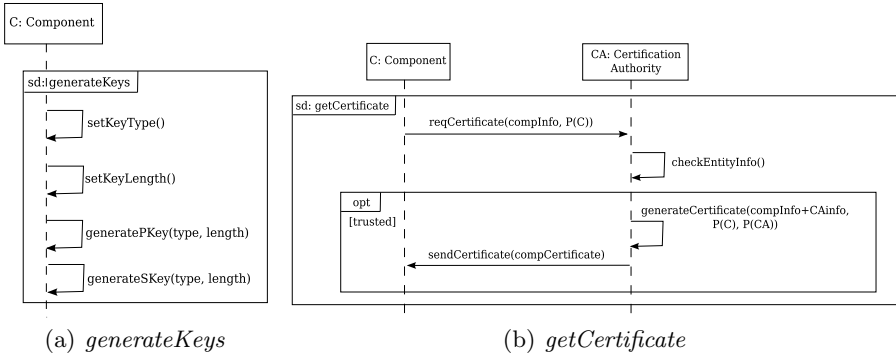


Fig. 2. UML Sequence Diagram of some preliminary operations

key exchange protocol that achieves mutual authentication and key exchange. This requires both parties to generate a key that can be combined to form a single session key. Three messages are exchanged. The first one is sent by the sender S and contains the sender information, the key generated by it, and a nonce; the message containing all this information is encrypted with the public key of the receiver. The second one is sent by the receiver R and contains the receiver information, the key generated by it, the nonce previously sent by S and a new nonce generated by R ; the message containing all this information is encrypted with the public key of the sender. The third one is sent by the sender and contains the nonce sent by the receiver. Finally, the encryption algorithm (i.e. *encryptAlgorithm*) is executed on the message (msg) with a session key obtained combining the keys generated by the sender and the receiver ($K(K_S, K_R)$).

Figure 4(a) shows the UML Sequence Diagram modeling the Digital Signature Generation. First, the hash function (*setHashFunction*) algorithm must be specified, then the digest generated (*generateDigest*) and finally the encryption algorithm (*encryptAlgorithm*), by using the entity private key, applied on the digest.

Figure 4(b) shows the UML Sequence Diagram modeling the Digital Signature Verification. A message (msg) and the digital signature ($digitSign$) are received as inputs. Two operations are performed: the first one is to calculate the digest (*execHashFunc*); the second one is the actual execution of the encryption algorithm applied on the input digital signature producing a forecast of the real signature (*encryptAlgorithm*). The last computation involves the verification of the digital signature (*verifyDigitSign*) which compares the forecast digital signature with the received one, in order to confirm the verification.

For sake of space the UML Sequence Diagram modeling the Decryption is not shown but it can be summarized as follows: after receiving the encrypted message, the algorithm type and the key length are extracted, and the decryption algorithm is executed to obtain the plain text.

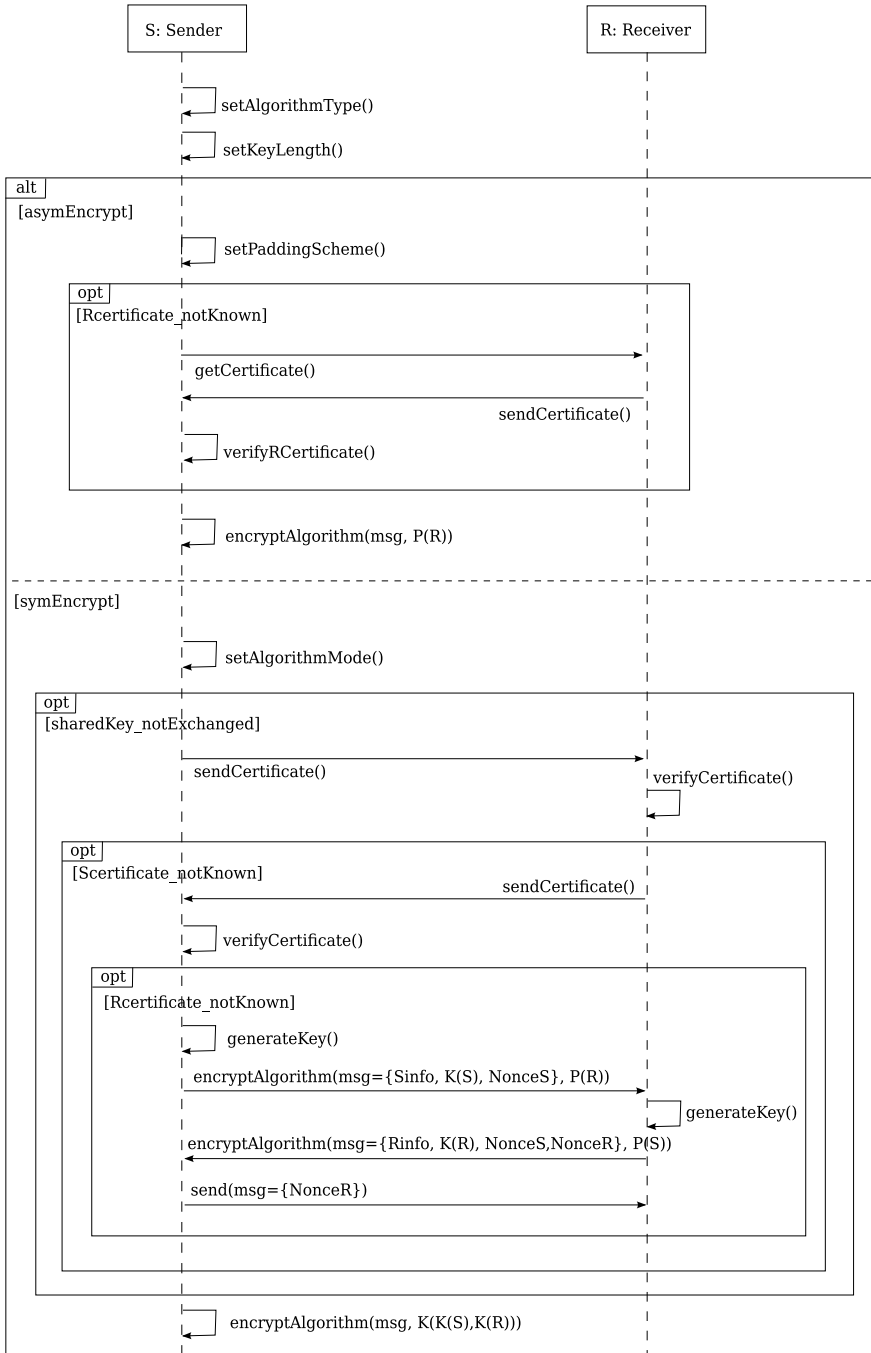


Fig. 3. UML Sequence Diagram of the *Encryption* mechanism

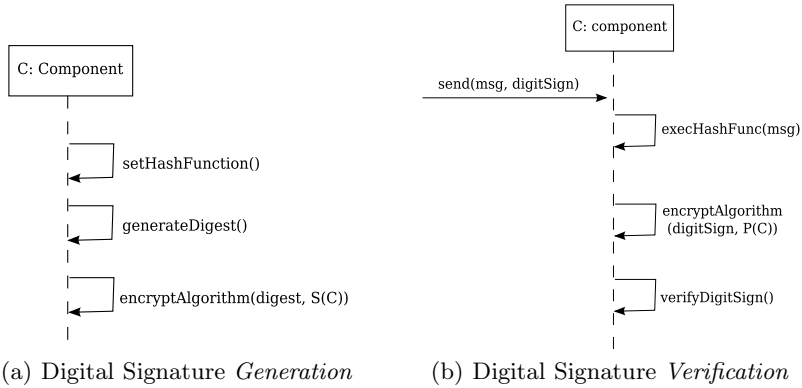


Fig. 4. UML Sequence Diagram of the *Digital Signature* mechanism

4.2 Security-Enabled Application Model

In this section we briefly discuss how the Composed Mechanisms of Table 1 are annotated and embedded in the application model to obtain a *Security-Enabled Application Model*.

The Data Confidentiality mechanism can be annotated on a software connector, and it means that data exchanged between the connected components are critical and need to be kept secret.

In Figure 5 we illustrate how the Composed Mechanism is enabled in the application model. On the left side the security annotation is added in the static architectural model (i.e. UML Component Diagram) on the software connector, and it means that client and supplier components exchange critical data. On the right side all Basic Mechanisms used to build the composed one (see Table 1) are embedded in the dynamic architectural model (i.e. UML Sequence Diagram), hence data are encrypted by the client component before their exchange and later decrypted by the supplier component.

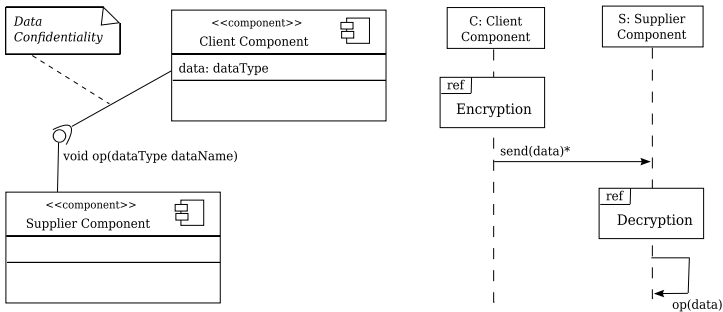


Fig. 5. Enabling *Data Confidentiality* mechanism

The Peer Entity Authentication and Data Integrity mechanisms are both obtained by the generation of the digital signature followed by its verification, as reported in Table 1. In fact they have the same modeling structure in terms of the sequence of operations. The difference is in the content of the message used to generate the digital signature. The Peer Entity Authentication can be annotated for software components and the content of the message is represented by its credentials, whereas Data Integrity can be annotated for attributes and the message is represented by application specific data.

Finally, the Data Origin Authentication mechanism can be annotated on attributes and means that data are critical and need to be authenticated. It depends on the digital signature generation, as reported in Table 1.

5 Experimental Validation

In this section we apply our approach to a real case study: a large distributed system in the domain of cultural asset management, built in the context of the CUSPIS project [7]. This experimental validation highlights the potential of our approach [3].

We denote by SC_i a *system configuration* that represents the required security settings to be included in the application model. It is obvious that the same application model may have multiple configurations, each leading certain security characteristics to the system.

For sake of experimentation we have numbered the *system configurations* in a hierarchical way SC_0, SC_1, \dots, SC_n , so that for configuration SC_i the required security settings properly include all the ones adopted for SC_j with $i > j$. SC_0 represents the system without any security setting. Such hierarchical organization of system configurations has been adopted in our case study in order to stress the progressive performance degradation introduced by the increasing of security.

However, our framework can be used to study the tradeoff between security and performance across different system configurations, not strictly ordered on raising security settings. For example, two generic configurations SC_i and SC_j may share certain settings but, at the same time, they may differ for other settings.

5.1 The CUSPIS System

The CUSPIS system aims to improve the protection of cultural assets (CA), such as sculptures and paintings, through the use of computer-based strategies (e.g. cryptography and satellite tracking). Our experimentation focuses on two services: *cultural asset authentication* and *cultural asset transportation* [4].

³ For sake of space we only report the most relevant results among all the evaluations carried out.

⁴ For sake of space we report only the *authentication* scenario, whereas we refer to [6] for the *transportation* one.

Cultural asset authentication aims to ensure that visitors to an exhibition, or potential buyers at an auction, can obtain cultural asset information and verify its authenticity (see [17] for details). Authentication is achieved by assigning a Geo Data (GD) tag containing information referring to each asset. A tag must be produced by a qualified organisation (e.g. the sculptures producer) to improve the asset protection.

Our experimentation in the cultural asset authentication service focuses on the *GD generation* scenario. It is performed in the following way: the qualified organisation generates the GD information (*genGDinfo*) and sends it (*send*) to a database that stores it (*storeGDinfo*).

The analysis of the *GD generation* scenario leads us to define two different system configurations (i.e. SC_1 and SC_2), as motivated in the following.

Figure 6(a) shows the *Security-Enabled Application Model* for the configuration SC_1 : the qualified organisation provides *Data Origin Authentication* of the *gd* data, that is uploaded to a database. The uploading does not require any security solution since we assume that it is performed through a secure channel. The operation that returns that value (*genGDinfo()*) is defined as a critical operation and tagged with a star in the UML Sequence Diagram of Figure 6(a). A “ref” fragment that points to the Digital Signature Generation mechanism is added, as stated in Table II.

The system configuration SC_1 is not security-wide when the qualified organisation device and the database communicate through an insecure network; the configuration SC_2 solves this problem by adding *Data Confidentiality* to the software connector requiring the *storeGDinfo()* operation. The exchange of data is performed through the *send()* operation that is defined as a critical operation, and it is tagged with a star in the UML Sequence Diagram of Figure 6(b). The references to the Encryption/Decryption mechanisms are added, as stated in Table II.

5.2 CUSPIS Implementation Details

Experiments for the CUSPIS system have been performed by running the same application code on two machines. The first machine has an Intel(R) Core2 T7250 CPU running at 2GHz with 2GB RAM, and runs the Windows Vista operating system. The second machine has an Intel Pentium4 3.4Ghz with 2GB RAM, and runs the Windows XP operating system.

In configuration SC_1 the digital signature was performed by using *SHAwithRSA* with different key sizes of 1024, 2048 and 4096 bits. In configuration SC_2 the encryption/decryption was performed by using an AES algorithm with a 256-bit key size in CBC mode.

5.3 Applying Our Approach to CUSPIS

In this Section we describe the experimental results that we have obtained from applying our approach to the CUSPIS system. From a performance analysis viewpoint, our experiments follow standard practices: construct the model, validate the model by comparing model results with real numerical values obtained from monitoring the implemented system while varying model parameters [10].

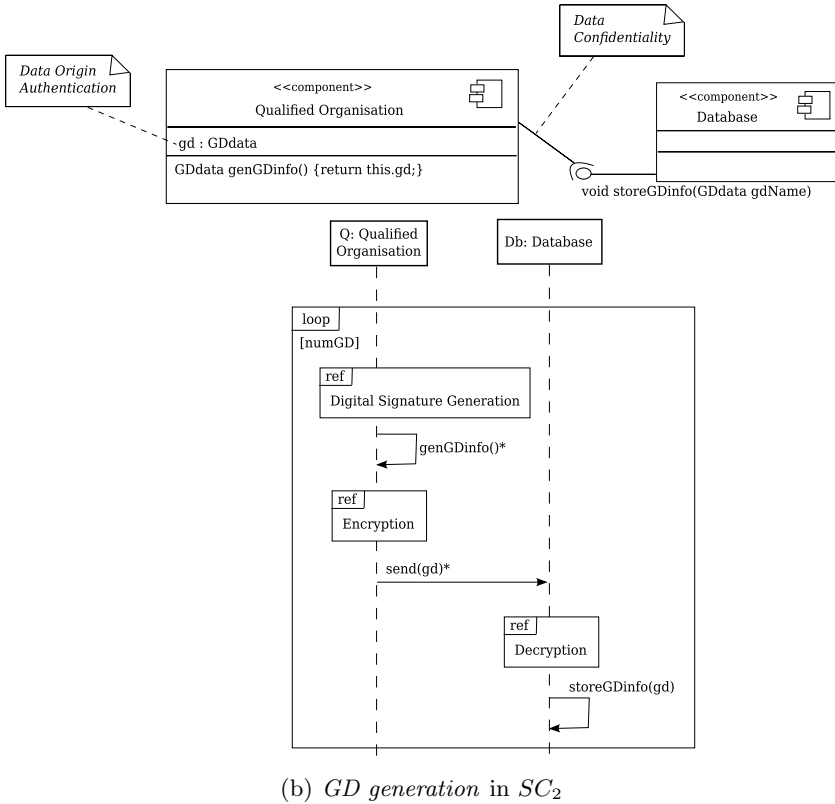
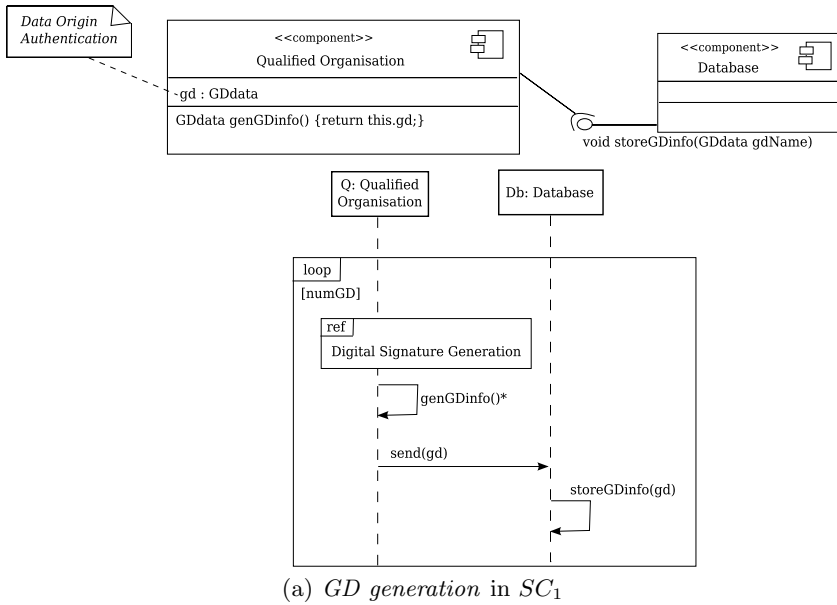


Fig. 6. Security-Enabled Application Models for *GD generation*

The validation of the *GD generation* scenario undergoes generic performance and security goals of a qualified organisation, which can be summarised as follows: (i) the number of tags generated per second must be as high as possible (a performance issue); (ii) tags must be hard to compromise (a security issue).

Based on the description in Section 5.1, two different GSPN performance models are built⁵, one for each configuration considered: SC_1 and SC_2 . Tables 2 and 3 report the results that we have obtained, respectively, for configurations SC_1 and SC_2 .

Table 2. *GD generation* - analysis of configuration SC_1

	KeySize (byte)	Model Solution Results (tags/sec)	Implementation Monitoring Data (tags/sec)	Model Prediction Error (%)
Platform 1	1024	17.45	17.3	0.86
	2048	9.32	9.11	2.25
	4096	1.98	1.92	3.03
Platform 2	1024	17.13	16.61	3.03
	2048	8.45	8.11	4.02
	4096	1.85	1.78	3.78

The columns of Table 2 can be divided into three sets. The first column reports the size of the key used in the encryption algorithm. The second set of columns reports the experimental results: the number of tags per second obtained from the model solution, the same values as monitored on the real implementation. Finally the last column reports the prediction error, expressed in percentage, of the model results in comparison to the monitored ones.

Table 3. *GD generation* - analysis of configuration SC_2

	KeySize (byte)	Model Solution Results (tags/sec)	Implementation Monitoring Data (tags/sec)	Model Prediction Error (%)
Platform 1	1024	3.43	3.29	4.08
	2048	2.93	2.85	2.73
	4096	1.35	1.33	1.48
Platform 2	1024	4.16	4.09	1.68
	2048	3.33	3.2	3.90
	4096	1.38	1.34	2.90

The rows of Table 2 can be divided into two sets, one for each platform considered (see Section 5.2). Within each set, numbers are reported for three values of the key size. For example, the first row of the Table 2 indicates the *GD generation* for the configuration SC_1 on Platform 1 with a 1024-bit key size: the model predicts that the system is able to generate 17.45 tags/second, the monitoring of the implementation reveals that the system actually generates 17.30 tags/second, and this leads to a gap between the model and the application of about 0.86%. Similarly

⁵ GSPN performance models are shown here 6.

promising results have been obtained for other key sizes and on both platforms, as shown in the last column of the Table, where the error never exceeds 4.02%.

Table 3 is similar to Table 2 for the organization of columns and rows; it collects the results for the configuration SC_2 . For example, the first row of Table 3 indicates the *GD generation* for the configuration SC_2 on Platform 1 with a 1024-bit key size: the model predicts that the system is able to generate 3.43 tags/second, the monitoring of the implementation reveals that the system actually generates 3.29 tags/second, and this leads to a gap between the model and the application of about 4.08%. In the Table this latter value is the worst one, in fact better predictive results have been obtained for other key sizes on both platforms, as shown in the last column of the Table 3.

In both tables the number of tags per second for the model solution was obtained by measuring the throughput value of the ending timed transition in both GSPN models. Besides, the corresponding metrics has been monitored on the actual implementation of the system. All these measures have been obtained with the system under workload stress, which occurs when the arrival rate is high enough to make the system always busy.

The analysis of workload for both SC_1 and SC_2 configurations is shown in Figure 7. The curves are obtained by solving the GSPN models under the configuration of Platform 1 with a key size of 1024 byte (i.e. the first row of Tables 2 and 3). In particular, on the x-axis the rate λ of arrivals to the system is reported; on the y-axis the throughput of the ending timed transition is shown. Note that in SC_1 the maximum throughput of 17.45 (see Table 2) is achieved for $\lambda = 23$ requests/second, whereas in SC_2 the maximum throughput of 3.43 (see Table 3) is achieved for $\lambda = 7$ requests/second.

From the comparison of Tables 2 and 3 some interesting issues emerge with regard to the performance degradation induced in SC_2 by raising security settings

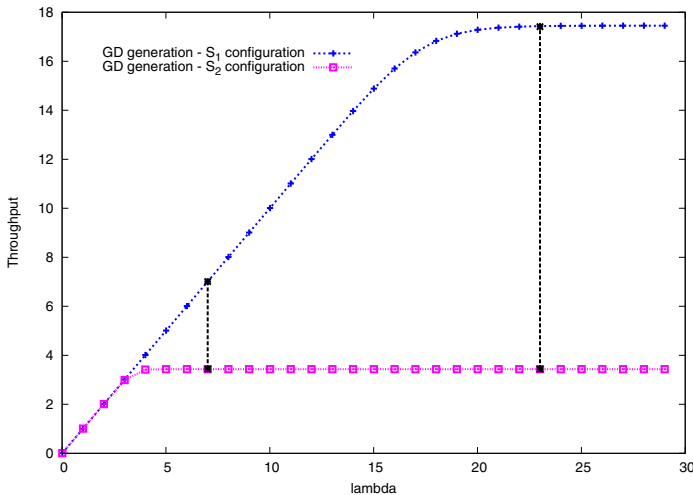


Fig. 7. *GD generation* throughput

(i.e. by introducing the additional Data Confidentiality mechanism). In Table 4 we have reported the percentage of performance degradation obtained (for model results and for monitored data) when moving from SC_1 to SC_2 configuration in both platforms while varying the key size. For example, the value 80.34% in the upper leftmost cell of the Table is obtained as $100 - (3.43 * 100)/17.45$ (see Tables 2 and 3).

Table 4. *GD generation: from SC_1 to SC_2*

KeySize (byte)	Model Results	Monitored Data	Model Results	Monitored Data
	Platform 1 (%)	Platform 1 (%)	Platform 2 (%)	Platform 2 (%)
1024	80.34	80.98	75.71	75.38
2048	68.56	68.71	60.59	60.54
4096	31.82	30.73	25.40	24.72

We note that our model consistently provides almost the same amount of performance degradation as the one observed in practice. This further supports the validity of our approach. An interesting consideration is that for smaller values of the key size the performance degradation is more dramatic. This is due to the fact that this key size affects the execution time of the Data Origin Authentication mechanism that is part of both configurations. Hence, while growing this size, the latter mechanism dominates in terms of execution time with respect to the Data Confidentiality, executed only in SC_2 , whose execution time does not vary with this key size.

6 Conclusions

In this paper we have introduced a framework to support the analysis of software architecture performance degradation due to the introduction of security mechanisms. Such a framework is that it is able to numerically quantify the system performance degradation while varying the adopted security solutions. This type of analysis can in fact support many decisions of software architects that span from simply evaluating if such performance degradation can be reasonably accepted from users, to choosing among different security solutions the one that provides the best tradeoff between security and performance properties.

A peculiar characteristic of our approach is the introduction of models for Basic Security Mechanisms. With this modular approach it is possible to study the performance degradation introduced by any meaningful combination of these (and possible newly built) security mechanisms. By pushing this concept ahead, a more complex type of analysis can be performed on models built by multiple Composed Mechanisms to represent the specification of an existing protocol, such as SSL. In this case an interesting study would be to observe how our models estimate the performance indices, and compare these results to what claimed in the corresponding protocol specifications.

As shown in the experiments, our architectural models very promisingly and quite accurately predict the performance of critical systems equipped with different security settings and implementation options. The results that we have obtained on our case study are somehow quite surprising in terms of percentage of degradation that can be introduced even from common security settings. Furthermore, we have been able to quantify the difference of degradation across platforms that, in some cases, achieves non-negligible thresholds.

The security mechanisms we consider (i.e. encryption and digital signature) can be seen as test beds for more complex security concerns. Modern applications may have to face with larger security vulnerabilities, and strategies for mitigating most of such vulnerabilities are cross-cutting and difficult to encapsulate (e.g., prevention of cross-site scripting errors). In this direction, our framework has been conceived to enable the modeling and analysis of security patterns that do not break the defined architectural abstraction level. In other words, the complexity of a pattern that implements a certain security strategy is not a problem on our framework as long as it can be (even piecewise) plugged into the application model. Moreover, such patterns can also spread from static to dynamic features of the system architectural model (e.g. see Figure 5).

In the near future we plan to automate the tradeoff analysis of the security configurations by automatically exploring the trade space. Such automation is feasible because performance models embedding security properties are generated once and the exploration of the trade space can be automatically performed by instrumenting the model with different numerical values for the input parameters. Besides, we devise to apply our approach to other real world examples in order to assess the scalability of the framework.

We consider this work as a starting point for studying even more sophisticated tradeoffs between security and performance. We plan to introduce into our evaluations the costs of security solutions as an additional attribute that very often affects software architects' decisions.

In the long term, it is of great interest to study the tradeoff between security and other non-functional attributes, such as availability. For example, addressing the problem of quantifying and locating data replicas for availability purposes without heavily affecting the security of the system would be crucial in certain domains.

Acknowledgments

This work has been partly supported by the italian project PACO (Performability-Aware Computing: Logics, Models, and Languages) funded by MIUR and by the UK EPSRC research grant EP/D076633/1 UbiVal (Fundamental Approaches to the Validation of Ubiquitous Systems).

References

1. UML 2.0 Superstructure Specification, OMG document formal/05-07-04, Object Management Group (2005), <http://www.omg.org/cgi-bin/doc?formal/05-07-04>

2. UML Profile for MARTE beta 2, OMG document ptc/08-06-09 (2008), <http://www.omgmarTE.org/Documents/Specifications/08-06-09.pdf>
3. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. *IEEE TSE* 30(5), 295–310
4. Blaze, M., Ioannidis, J., Keromytis, A.D.: Trust management for ipsec. *ACM Transactions on Information and System Security* 5(2), 95–118 (2002)
5. Cortellessa, V., Trubiani, C.: Towards a library of composable models to estimate the performance of security solutions. In: *WOSP*, pp. 145–156 (2008)
6. Cortellessa, V., Trubiani, C., Mostarda, L., Dulay, N.: An Architectural Framework for Analyzing Tradeoffs between Software Security and Performance - Extended results. Technical Report 001-2010, Dipartimento di Informatica - Università dell'Aquila (2010), <http://www.di.univaq.it/cortelle/docs/001-2010-report.pdf>
7. European Commission 6th Framework Program. Cultural Heritage Space Identification System (CUSPIS), www.cuspis-project.info
8. France, R.B., Ray, I., Georg, G., Ghosh, S.: Aspect-oriented approach to early design modelling. *IEE Proceedings - Software* 151(4), 173–186 (2004)
9. Gupta, V., Gupta, S., Shantz, S.C., Stebila, D.: Performance analysis of elliptic curve cryptography for SSL, pp. 87–94 (2002)
10. Harbitter, A., Menasce, D.A.: A methodology for analyzing the performance of authentication protocols. *ACM TISSEC* (2002)
11. Hirel, C., Sahner, R., Zang, X., Trivedi, K.: Reliability and performability modeling using sharpe 2000. In: Haverkort, B.R., Bohnenkamp, H.C., Smith, C.U. (eds.) *TOOLS 2000*. LNCS, vol. 1786, pp. 345–349. Springer, Heidelberg (2000)
12. Juric, M.B., Rozman, I., Brumen, B., Colnaric, M., Hericko, M.: Comparison of performance of web services, ws-security, rmi, and rmi-ssl. *Journal of Systems and Software* 79(5), 689–700 (2006)
13. Jurjens, J.: *Secure Systems Development with UML* (2004)
14. Kant, K., Iyer, R.K., Mohapatra, P.: Architectural impact of Secure Socket Layer on internet servers, pp. 7–14 (2000)
15. Marsan, M.A., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: *Modelling with Generalized Stochastic Petri Nets*, 4th edn. (November 1994)
16. Menascé, D.A.: Security performance. *IEEE Internet Computing* 7(3), 84–87 (2003)
17. Mostarda, L., Dong, C., Dulay, N.: Place and Time Authentication of Cultural Assets. In: 2nd Joint ITRUST and PST Conferences on Privacy, Trust and Security, IFIPTM 2008 (2008)
18. Stallings, W.: *Cryptography and network security: Principles and Practice*, 4th edn. Prentice-Hall, Englewood Cliffs (2006)
19. Tai, A.T., Meyer, J.F., Avizienis, A.: *Software Performability: From Concepts to Applications*. Kluwer Academic Publishers, Boston (1996)
20. Tawhid, R., Petriu, D.C.: Towards automatic derivation of a product performance model from a UML software product line model. In: *WOSP*, pp. 91–102 (2008)
21. Trivedi, K.: Sharpe interface, user's manual, version 1.01. Technical report (1999), <http://www.ee.duke.edu/~chirel/MANUAL/gui.doc>
22. Woodside, C.M., Petriu, D.C., Petriu, D.B., Xu, J., Israr, T.A., Georg, G., France, R.B., Bieman, J.M., Houmb, S.H., Jürjens, J.: Performance analysis of security aspects by weaving scenarios extracted from UML models. *Journal of Systems and Software* 82(1), 56–74 (2009)

Architectural Design Decisions for Achieving Reliable Software Systems

Atef Mohamed and Mohammad Zulkernine

School of Computing
Queen's University, Kingston
Ontario, Canada K7L 3N6
{atef,mzulker}@cs.queensu.ca

Abstract. Software architectural design decisions are key guidelines to achieve non-functional requirements of software systems in the early stages of software development. These decisions are also important for justifying the modifications of dynamic architectures during software evolution in the operational phase. Incorporating reliability goals in software architectures is important for successful applications in large and safety-critical systems. However, most of the existing software design mechanisms do not consider the architectural reliability (the impact of software architecture on system reliability). As a result, alternative software architectures cannot be compared adequately with respect to software system reliability. In this paper, we extend our previous work on failure propagation analysis to propose a selection framework for incorporating reliability in software architectures. The selection criterion in this framework exploits architectural attributes to appropriately select software architectures based on their reliabilities. We provide algorithms to derive the architectural attributes required by the model and to select the appropriate architecture using a quick and a comprehensive decision approach for minor and major architectural changes, respectively.

Keywords: Software architecture, architectural design decisions, architectural service routes, and architectural reliability.

1 Introduction

Software architecture is an important artifact that provides powerful means to incorporate quality attributes in software intensive systems [16]. Software practitioners recognize the importance of architectural design decisions to reason about their design choices [27]. These decisions are taken in the early design stages and their impacts are carried out to the later development stages. They are also taken in response to a failure, regular maintenance, repair, configuration, or other activities during the system operation. Therefore, “architectural design decisions deserve to be first class entities in the process of developing complex software-intensive systems” [23]. Appropriate architectural design decisions are critical for achieving software quality attributes.

Software system reliability (the continuity of correct service) is important for the successful applications in large scale, safety-critical, and mission-critical software systems. *Architectural reliability* is the impact of a software architecture on the software system

reliability. Reliable interactions among architectural entities is an important mean for ensuring the architectural reliability [3]. Unfortunately, reliability has not been sufficiently addressed, and the quantitative impact of a software architecture on this quality attribute has not been explicitly considered in the existing software architectural design methodologies. As a result, existing software development methods do not sufficiently explain the rationale behind the adoption of alternative architectures with respect to their impacts on the software system reliability [31]. One of the main aspects of unreliable interactions of system components is the failure propagation through the component interfaces. Failure propagation among system components has significant impact on the reliability of the whole system [24].

Our previous work [21] proposes a technique for analyzing the behavior of failure propagation, masking, and scattering among software system components. The analysis determines upper and lower bounds of failure masking and propagation in these systems. In this paper, we utilize the failure propagation analysis results of our previous work [21] to implement a framework for comparing and selecting the appropriate architectural modifications with respect to system reliability. Our selection criterion exploits the concept of Architectural Service Routes (ASRs) [21]. The concept of ASRs allows quantifying architectural quality attributes by viewing a software architecture as a set of components and a set of service routes connecting them. We exploit the architectural attributes derived by this concept to show the appropriate design decisions based on their impacts on system reliability. These attributes are the number and lengths of ASRs, shortest ASR, and longest ASR between every pair of system components. We show how to select a more reliable software architecture using two design decision approaches: a quick approach for deciding about a minor change of an architecture based on the failure propagation relationships to the architectural attributes and a comprehensive approach for deciding about a whole architecture based on its quantified architectural reliability.

The main contribution of this work is the incorporation of the reliability in software architectures based on the failure propagation among system components. Software architectures can be compared with respect to their impacts on system reliability. The major implication of this work is that the technique can be used in the early design stages and its design decision impacts are carried out across the later development stages. It can also be used during the operational stage to decide about the architectural modifications of dynamic software architectures. Moreover, it can be used to discover the weak points of an architecture.

The rest of this paper is organized as follows. Sec. 2 discusses some related work on software architecture reliability analysis. Sec. 3 discusses the concept and the identification of the ASRs. It also introduces some architectural attributes and shows their relationships to failure propagation. In Sec. 4, we show how to use the architectural attributes and the quantitative evaluations of reliability to select appropriate software architectures. We provide a quick approach and a comprehensive approach for deciding about an architecture. Finally, in Sec. 5, we present the conclusions and future work.

2 Related Work

There has been an ongoing drive for reasoning about reliability at the architectural level. Consequently, Architectural Description Languages (ADLs) introduce new notations,

methods and techniques that provide the necessary support for this reasoning [9]. Our work in this paper, provides architectural design decisions for incorporating reliability in software system architectures. The work allows comparing software architectures based on their reliabilities. It exploits a number of architectural attributes that can be considered as architectural notations and can be integrated with the existing ADLs to provide more reasoning about software system reliability.

Architectural design decisions can be used to achieve quality attributes of a software architecture during the initial construction [5,28,7] or during the evolution [30,10] of a software system [15]. Candea *et al.* [5] present a technique for determining the weak points of a software architecture by automatically capturing dynamic failure propagation information among its components. They use instrumented middleware to discover potential failure points in the application. By using a controlled fault injection mechanism and observing component failures, their technique builds a failure propagation graph among these components. Voas *et al.* [28] investigate an assessment technique for software architectural reliability based on “failure tolerance” among the components of a large scale software systems. Their approach is based on fault injection mechanism, in which, they corrupt the data passing through component interfaces and observe the system reaction. Cortellessa *et al.* [7] present an approach for analyzing the reliability of a software based on its component interactions. They present a failure propagation model in assessing software reliability. In this work, we utilize the failure propagation analysis results of [21] as a design rationale for incorporating system reliability in software architectures. We provide two decision approaches for comparing and selecting software architectures with respect to the software system reliability. Based on this decision approach, one can determine the weak points of a software architecture based on the relationships between the failure propagation and the architectural attributes. Our technique is more suitable for justifying the architectural design alternatives since it is easier to implement, and it avoids instrumented middleware and fault injections.

In dynamic architectures, components and connectors can be added or removed during the system operation. These architectural modifications could occur either as a result of some computations performed by the system or as a result of some configuration actions [10]. In response to system monitored events, dynamic (*e.g.*, self-organizing) software architectures use rules or constraints to decide about the appropriate modifications. Wang [30] provide a rule-based model to extract scattered rules from different procedures and enhance the self-adaptability of the software. Georgiadis *et al.* [10] examine the feasibility of using architectural constraints to specify, design, and implement software architectures. Our selection framework can also incorporate reliability in self-organizing software architectures based on failure propagation among system components. The selection criterion may enrich these self-adaptive models through its reliability consideration in specifying the dynamic rules and architectural constraints.

3 Architectural Attributes

Software architecture of a system is the structure, which comprises software components, externally visible properties of those components, and the relationships among them [4]. In software architectures, a *component* is a unit of composition with

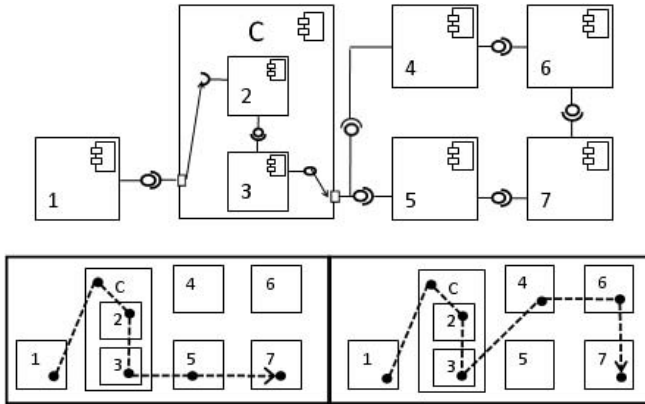


Fig. 1. Example Architectural Service Routes (ASRs)

contractually specified interfaces. A *component interface* is a mean by which a component connects to other components or to the environment [26]. A *component service* is a facility that a component provides to, or requires from other components as specified in the formal contracts [21][22]. Notation languages are used to simulate the dynamism in software architectures [2][17]. These languages describe the architectural changes of the dynamic architectures by using configuration elements and specify dynamic control of these configuration elements. An *architectural configuration* is an architecture that results by the control decisions of a dynamic architectural configuration element. Every configuration element selects an architectural modification among the possible changes that are specified in the dynamic architectural design artifact.

In this section, we introduce the concept of ASRs and the related architectural attributes. These architectural attributes are used to calculate failure propagation probabilities and relate the software architectures to the system reliability. Based on the failure propagation and the reliability quantification, these attributes are then used to derive the architectural design rationale and decisions.

3.1 Architectural Service Routes

Architectural Service Routes (ASRs) [21] are used to relate software architectures and system reliability. An *Architectural Service Route (ASR)* is a sequence of components in which every component provides a service to the next component through an interface connection. In Fig. 1, services are provided from a number of components to others. For example, component 2 provides service to component 3. Component 3, likewise, provides services to both component 4 and component 5. Therefore, component 2 provides its service to component 4 and 5 indirectly through component 3. In other words, a service failure of component 2 may affect the services provided by component 4 and 5. In the bottom part of Fig. 1, we show two example ASRs between components 1 and 7 of the provided UML 2.0 component diagram. The sequences of components are (1, C, 2, 3, C, 5, 7) and (1, C, 2, 3, C, 4, 6, 7) for the left and the right ASR, respectively.

Algorithm 1. Determining ASR sets**Input:** Software component diagram.**Output:** Sets of ASRs between each pair of components i and j ($\forall_{i,j}\Psi^{i,j}$).

```

01. FOR each set of ASRs  $\Psi^{i,j}$  DO
02.    $\Psi^{i,j} := Empty$ ;
03. END FOR
04. FOR each component  $u$  DO
05.   FOR each component  $v$  DO
06.     IF (interface connection from  $u$  to  $v$  exists) THEN
07.        $\psi_1^{u,v} := (u, v)$ ;
08.       FOR each set of ASRs  $\Psi^{i,j}$  DO
09.         FOR each ASR  $\psi_k^{i,j} \in \Psi^{i,j}$  DO
10.           IF ( $u = j$ ) THEN
11.              $w := |\Psi^{i,v}| + 1$ ;
12.              $\psi_w^{i,v} := \psi_k^{i,j}$ ;
13.             INSERT  $v$  at the end of  $\psi_w^{i,v}$ ;
14.           END IF
15.           IF ( $i = v$ ) THEN
16.              $w := |\Psi^{u,j}| + 1$ ;
17.              $\psi_w^{u,j} := \psi_k^{i,j}$ ;
18.             INSERT  $j$  at the beginning of  $\psi_w^{u,j}$ ;
19.           END IF
20.         END FOR
21.       END FOR
22.     END IF
23.   END FOR
24. END FOR
25. RETURN  $\forall_{i,j} \Psi^{i,j}$ ;

```

In UML 2.0 component diagrams, an ASR can be distinguished by a series of assembly and/or delegation connectors between two components. Any two different components x and y can have zero or more ASRs. For example, in Fig. [1](#) components 4 and 5 have zero ASR, components 2 and 6 have one ASR: $(2, 3, C, 4, 6)$, and components 3 and 7 have two ASRs: $(3, C, 4, 6, 7)$ and $(3, C, 5, 7)$. We refer to the set of ASRs from x to y as $\Psi^{x,y}$, and we denote an ASR in this set as $\psi_k^{x,y}$, where k is the index of the k -th ASR in $\Psi^{x,y}$. An ASR $\psi_k^{x,y}$ can have zero or more components in the sequence of components in addition to the pair of components x and y .

An architecture can be seen as a directed graph where components are the nodes and interface connections between components are the arrows connecting these nodes. Algorithm [1](#) determines the sets of ASRs between each pair of components in a software architecture. Lines 01-03 initialize the sets of ASRs to the empty sets. Lines 04-24 navigate through the whole architecture to access every interface connection between any two components and determine the ASRs based on these interface connections. We visit two components at a time and check for the existence of an interface connection

between them in Lines 06-22. We consider one of these two components as a service providing component u and the other as a service requesting component v , and we check for the interface connections between them. If a connection is found then in Line 07, the algorithm creates an ASR from u to v and marks this ASR as the first one in the set of ASRs between the two components u and v . Moreover, in Lines 08-21, the algorithm checks all the previously determined ASRs to find the ones that are adjacent to the current ASR $\psi_1^{u,v}$. Two ASRs $\psi_i^{w,x}$ and $\psi_j^{y,z}$ are *adjacent* if one ends with the source component of the other, *i.e.*, if $x = y$ or $z = w$. If an adjacent ASR $\psi_i^{t,u}$ or $\psi_j^{v,w}$ is found, the algorithm creates a new ASR between components (t and v) or (u and w) that corresponds to the combined ASR $\psi_k^{t,v}$ or $\psi_{k'}^{u,w}$, respectively. Lines 10-14 check the adjacent ASRs that ends with the service providing component u , create combined ASRs, and determine their indexes. In a similar procedure, Lines 15-19 process the adjacent ASRs that starts with the service requesting component v . Line 25 returns the sets of ASRs.

3.2 Service Route-Based Architectural Attributes

Between every pair of components in a software architecture, we may find zero or more ASRs. These ASRs can have zero or more components in the sequence of components in addition to this pair of components. The attributes of the ASRs between the pairs of components of an architecture are described in the following paragraphs.

For any two components x and y of a software architecture, we define the following architectural attributes.

- The number of ASRs, $|\Psi^{x,y}|$.
- The k -th ASR length for all $0 \leq k \leq |\Psi^{x,y}|$, $L_k^{x,y}$.
- The length of the shortest ASR, $L_S^{x,y}$.
- The length of the longest ASR, $L_L^{x,y}$.

The number of ASRs, $|\Psi^{x,y}|$ counts the ASRs from component x to component y , *i.e.*, it is the cardinality of the set $\Psi^{x,y}$. For example in Fig. 1 $|\Psi^{3,6}| = 1$ and $|\Psi^{3,7}| = 2$.

The length of the k -th ASR, $L_k^{x,y}$ (where $0 \leq k \leq |\Psi^{x,y}|$) is the number of components in the k -th ASR from components x to y . In Fig. 1 $L_1^{2,6} = 4$, $L_1^{3,5} = 2$, and $L_1^{4,5} = 0$. An example set of ASRs is $\Psi^{3,7} = \{(3, 4, 6, 7), (3, 5, 7)\}$, therefore, $L_1^{3,7} = 4$ and $L_2^{3,7} = 3$. It is obvious that, for any component x , $L_1^{x,x} = 1$, since $\psi_1^{x,x}$ is (x).

The length of the shortest ASR, $L_S^{x,y}$ is the number of components of the shortest ASR from component x to component y . For example in Fig. 1 the shortest ASR between component 3 and 7 is $\psi_S^{3,7} = (3, 5, 7)$ and its length is $L_S^{3,7} = 3$.

The length of the longest ASR, $L_L^{x,y}$ is the number of components of the longest ASR from component x to component y . For example, in Fig. 1 the longest ASR between component 3 and 7 is $\psi_L^{3,7} = (3, 4, 6, 7)$ and its length is $L_L^{3,7} = 4$.

3.3 Architectural Reliability

One of the major aspects of software system reliability is the propagation of failures among its components. Failure propagation from any component x to another component y in a software architecture indicates the probability that a failure of x propagates

to y through the set of ASRs $\Psi^{x,y}$. Symbolically, the probability of failure propagation from x to y through the set of ASRs $\Psi^{x,y}$ is expressed as P_{xy}^f [21].

$$P_{xy}^f = \bigcup_{k=1}^{|\Psi^{x,y}|} \left(\bigcap_{j=1}^{L_k^{xy}-1} E_{c_{j+1}}^f \right) E_y^f \quad (1)$$

where L_k^{xy} is the length of the k -th ASR from x to y , and $E_{c_{j+1}}^f$ is the probability that an input error e_j from component c_j causes a failure f in component c_{j+1} . Architectural reliability is evaluated in terms of failure propagation probabilities among system components as follows.

$$R_A = \bigcap_{h=1}^H \left(1 - \bigcup_{i=1}^I p_i^f P_{ih}^f \right) \quad (2)$$

where R_A is the architectural reliability of architecture A , I is the number of the system components, and H is the number of output interface components. p_i^f is the probability of failure occurrence in component i . P_{ih}^f is the failure propagation probability from component i to an output interface component h .

The failure propagation in Eq. (1) is computed based on the knowledge of the probabilities of input error causing a failure f ($E_{c_j}^f$) in component c_j . System reliability in Eq. (2) is also computed based on the knowledge of the failure occurrence probabilities of system components. Our model assumes the availability of these parameters. However, several ‘‘data change probability’’ models [11][4][29] and component reliability models [8][6][13] can be adopted to estimate these parameters for the individual components.

To decide about each individual attribute in a software architecture based on its impact on the system reliability, the relationships between failure propagation and these architectural attributes are defined [21] as follows.

Failure propagation and shortest ASR relationship. The probability that a failure f propagates from component x to component y is inversely proportional to the length of the shortest ASR from x to y .

$$P_{xy}^f \propto \frac{1}{L_S^{xy}} \quad (3)$$

Failure propagation and longest ASR relationship. The probability that a failure f propagates from x to y is inversely proportional to the length of the longest ASR from x to y .

$$P_{xy}^f \propto \frac{1}{L_L^{xy}} \quad (4)$$

Failure propagation and number of ASRs relationship. The probability that a failure f propagates from x to y is directly proportional to the number of ASRs from x to y .

$$P_{xy}^f \propto |\Psi^{x,y}| \quad (5)$$

In the following section, we show how to exploit the above relationships to decide about software architectural modifications based on their impacts on system reliability.

4 Architectural Design Decisions

An *architectural design decision* is the outcome of a design process during the initial construction or the evolution of a software system [15]. Architectural design decisions are classified into three major classes: existence, executive, and property [23]. An existence decision is related to the existence of new artifacts in a system design or implementation. Executive decisions are business-driven and they affect the methodology of the development process, the user training, and the choices of technologies and tools. A property decision controls the quality of the system. These decisions can be design rules, guidelines or design constraints. Our approach provides “property class” architectural design decisions.

We present our rationale behind the architectural design decisions based on the architectural attributes and their impacts on architectural reliability. The technique can be used to compare software architectures and appropriately select the more reliable architecture among them. The most radical changes of these architectures are removing or adding components and associated interface connections and a less radical changes are removing or adding interface connections without changing the components themselves [11]. We present a quick approach for deciding about a minor change of an architecture based on the failure propagation relationships to the architectural attributes. We also present a comprehensive approach for deciding about a whole architecture based on quantified architectural reliability. The choice of applying the quick approach or applying the comprehensive approach depends on the amount of changes that need to be decided for an architecture. If same components exist in both architectures, while the architectures differ only in few interface connections, then the quick approach can be used. The comprehensive approach can be used for deciding about any amount of changes between architectures. If the quick approach is not able to provide a decision based on those changes, the comprehensive approach must be selected. The comprehensive approach always succeeds to provide a decision.

4.1 Quick Decision Approach

Software designers often need to decide about a part of an architecture, *i.e.*, a minor change to a few number of architectural entities. For example, they may need to create and/or remove an interface connection between two existing components in an architecture. In this case, we do not have to re-evaluate the whole architecture. Instead, we can decide based on the direct relationships between the architectural attributes and failure propagation exclusively with respect to the changes in the specific part of the architecture as follows.

Consider a software architecture A contains n components with a set of ASRs between every pair of components, where $\Psi^{i,j}$ denotes the set of ASRs between components i and j . Some of these components are input interface, output interface, or internal components. Assume that components h_1 and h_2 are output interface components. Consider that a minor change to architecture A leads to a slightly different architecture A' ,

Table 1. Values of an architectural attribute: number of ASRs ($|\Psi^{i,j}|$) between each pair of components

(a) Architecture A			(b) Architecture A'		
Component	h_1	h_2	Component	h_1	h_2
1	$ \Psi^{1,h_1} $	$ \Psi^{1,h_2} $	1	$ \Psi'^{1,h_1} $	$ \Psi'^{1,h_2} $
2	$ \Psi^{2,h_1} $	$ \Psi^{2,h_2} $	2	$ \Psi'^{2,h_1} $	$ \Psi'^{2,h_2} $
3	$ \Psi^{3,h_1} $	$ \Psi^{3,h_2} $	3	$ \Psi'^{3,h_1} $	$ \Psi'^{3,h_2} $
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
i	$ \Psi^{i,h_1} $	$ \Psi^{i,h_2} $	i	$ \Psi'^{i,h_1} $	$ \Psi'^{i,h_2} $
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
n	$ \Psi^{n,h_1} $	$ \Psi^{n,h_2} $	n	$ \Psi'^{n,h_1} $	$ \Psi'^{n,h_2} $

where $\Psi^{i,j}$ denotes the set of ASRs between components i and j . The sets of ASRs for architecture A and A' can be determined by following Algorithm 1. Given these sets of ASRs for both the architectures, we can determine the values of the architectural attributes for each of them. For each attribute, we need to build a table of attribute values for each pair of components $\langle i, j \rangle$, where i is any component of the architecture and j is any output interface component. We only need to compare the architectural attributes of all ASRs that end with h_1 and h_2 where a system failure may occur.

Table 1 shows an example architectural attribute: number of ASRs. In this table, each of the n rows ($1, \dots, n$) represent a system component. The second and third columns are the output interface components h_1 and h_2 . For example, the value of $|\Psi^{1,h_1}|$ is the number of ASRs from component 1 to the output interface component h_1 . Similar to Table 1, we need to create two more tables for the length of the shortest ASR and the length of the longest ASR. Having the values for all architectural attributes, we may be able to select between architecture A and A' using Algorithm 2.

This algorithm takes one of the following actions: select an architecture (*select-A* or *select-A'*), select both architectures (*select-either*), or call the comprehensive decision approach. Selecting both architectures means that both of them have equal impacts on failure propagation. Algorithm 2 calls the comprehensive decision approach (Algorithm 3) when it fails to find the appropriate selection based on some contradictory impacts on failure propagation. For example, the algorithm fails to decide when the changes of an architecture lead to a shorter ASRs between a pair of components $\langle i', j' \rangle$, while it decreases the number of ASRs of $\langle i', j' \rangle$. In this case, a comprehensive comparison of these architectures is required using Algorithm 3.

In the beginning, Algorithm 2 considers that both architectures A and A' are equally reliable (Lines 01-02). Algorithm 2 selects an architecture only when all the changes of that architecture are not decreasing the reliability, otherwise, it calls the comprehensive decision approach to provide the proper selection. Therefore, the algorithm searches for cases where the changes of an architecture is negatively impacting reliability to exclude this architecture from the selection. Lines 03-21 access the ASR sets between every pair of a component and an output interface component to compare the architectural

Algorithm 2. Quick decision approach

Input: Architectural attribute values.

Output: Selected architecture.

```

01. select- $A := True$ ;
02. select- $A' := True$ ;
03. FOR each component  $u$  DO
04.   FOR each output interface component  $v$  DO
05.     IF ( $|\Psi^{u,v}| \leq |\Psi'^{u,v}|$  and  $L_S^{u,v} \geq L'_S{}^{u,v}$  and  $L_L^{u,v} \geq L'_L{}^{u,v}$ ) THEN
06.       IF ( $|\Psi^{u,v}| < |\Psi'^{u,v}|$  or  $L_S^{u,v} > L'_S{}^{u,v}$  or  $L_L^{u,v} > L'_L{}^{u,v}$ ) THEN
07.         select- $A' := False$ ;
08.       END IF
09.     ELSE-IF ( $|\Psi^{u,v}| \geq |\Psi'^{u,v}|$  and  $L_S^{u,v} \leq L'_S{}^{u,v}$  and  $L_L^{u,v} \leq L'_L{}^{u,v}$ ) THEN
10.       IF ( $|\Psi^{u,v}| > |\Psi'^{u,v}|$  or  $L_S^{u,v} < L'_S{}^{u,v}$  or  $L_L^{u,v} < L'_L{}^{u,v}$ ) THEN
11.         select- $A := False$ ;
12.       END IF
13.     ELSE
14.       select- $A := False$ ;
15.       select- $A' := False$ ;
16.     END IF
17.     IF (select- $A = False$  and select- $A' = False$ ) THEN
18.       BREAK;
19.     END IF
20.   END FOR
21. END FOR
22. IF (select- $A = True$  and select- $A' = True$ ) THEN
23.   RETURN select-either;
24. ELSE-IF (select- $A = True$  and select- $A' = False$ ) THEN
25.   RETURN select-A;
26. ELSE-IF (select- $A = False$  and select- $A' = True$ ) THEN
27.   RETURN select-A';
28. ELSE
29.   CALL Comprehensive decision approach (Algorithm 3);
31. END IF

```

attributes of architectures A and A' . By tracking only the ASRs from any component to any output interface component, we identify the architectural attributes that impact reliability based on the occurrences of failures at the output interface components. We visit one set of ASRs at a time and compare the architectural attributes (the number of ASRs, the length of the shortest ASR, and the length of the longest ASR) of the two architectures in Lines 05-16. The logical conditions of the comparison is formed based on the relationships of the architectural attributes and failure propagation in Eq. 3-5. We have three cases with respect to this comparison as discussed in the following paragraph.

First, architecture A' has less or equal impact on reliability based on the changes in the current set of ASRs than architecture A (Lines 05-08). In this case, if based

on any architectural attribute, architecture A positively influences the reliability more than A' (Lines 06-08), then we can conclude that, there is at least one change between architectures A and A' that causes A' to be less reliable than A . Here, the algorithm decides not to select architecture A' (Line 7). Second, architecture A has less or equal impact on reliability based on the changes in the current set of ASRs than A' (Lines 09-12). This is similar to case 1, while it checks whether architecture A will be selected or not. Third, in Lines 13-16, both architectures A and A' may have positive and negative impacts on reliability based on the attributes of the current set of ASRs. In this case, the algorithm chooses not to select any particular architecture (Lines 14-15). In Lines 22-31, the algorithm returns a decision based on all sets of ASRs. The decision can include a specific architecture (if one is more reliable), two architectures (if they are similar), or an application of the comprehensive decision approach (if the quick decision approach cannot select an architecture based on the provided changes).

4.2 Comprehensive Decision Approach

Software designers may also need to decide about the whole architecture whether it satisfies the reliability attribute or to compare between two architectures based on their impacts on reliability. This is the case when a major change to an architecture takes place. For example, some changes lead to a different number of components or connections between them. In this case, the quick architectural design decision approach fails to select an architecture. Therefore, we evaluate the whole architecture to measure the

Algorithm 3. Comprehensive decision approach based on failure propagation

Input: Architectural attribute values.

Output: Selected architecture.

```

01. FOR each component  $u$  DO
02.   FOR each output interface component  $v$  DO
03.     Calculate  $P_{uv}^f$  for architecture  $A$  using Eq. 11
04.   END FOR
05. END FOR
06. FOR each component  $u'$  DO
07.   FOR each output interface component  $v'$  DO
08.     Calculate  $P_{u'v'}^f$  for architecture  $A'$  using Eq. 11
09.   END FOR
10. END FOR
11. Calculate architectural reliability for architecture  $A$  using Eq. 12
12. Calculate architectural reliability for architecture  $A'$  using Eq. 12
13. IF (reliability of  $A >$  reliability of  $A'$ ) THEN
14.   RETURN select-A;
15. IF (reliability of  $A <$  reliability of  $A'$ ) THEN
16.   RETURN select-A';
17. ELSE
18.   RETURN select-either;
19. END IF

```

quantitative impacts of the new changes on the system reliability. In the comprehensive approach, we need to calculate the failure propagation probabilities among all components for both architectures. The failure propagation probabilities are used to quantify the reliability of each architecture. Based on the architecture reliabilities, a designer can select the appropriate architecture.

The comprehensive architectural design decision is taken based on the reliability evaluation using failure propagation (Algorithm 3). The algorithm selects the architecture that has higher reliability and performs more exhaustive comparison than the quick decision approach of Algorithm 2.

In Algorithm 3, Lines 01-05 access every pair of a component and an output interface component to calculate the failure masking probabilities between the pair of components for architectures A . Similarly, Lines 06-10 calculate the failure masking probabilities for architectures A' . Line 11 and Line 12 calculate the architectural reliability for architecture A and A' , respectively. Based on the architectural reliabilities of A and A' , Lines 13-19 select the architecture with higher reliability.

5 Conclusions and Future Work

Appropriate architectural design decisions are important for achieving quality attributes of software architectures. Reliability has not been sufficiently addressed and its quantitative impacts on software architectures have not been explicitly considered in existing software architectural design methodologies. As a result, current architectural strategies do not sufficiently consider the rationale behind the adoption of alternative architectures with respect to their reliabilities.

In this paper, we present a selection framework for incorporating reliability in software architectures based on the failure propagation among the software system components. The proposed technique makes appropriate design decisions based on the following architectural attributes: the number and lengths of ASRs, the shortest ASR, the and longest ASR between every pair of system components. Based on the failure propagation analysis [21], we show how to appropriately select a more reliable software architecture using two design decision approaches: a quick approach for deciding about a minor change of an architecture based on the failure propagation relationships to the architectural attributes and a comprehensive approach for deciding about a whole architecture based on quantified architectural reliability.

In our future work, we intend to involve the study of ASR intersections and cyclic ASRs in the architectural design decisions. An ASR intersection is a number of common component interfaces among multiple ASRs. A cyclic ASR is a closed sequence of connected components. By considering ASR intersections and cyclic ASRs, we may be able to measure the architectural impacts on the reliability of more complicated software architectures. Another direction of future research will incorporate failure severities in the architectural design decisions. Some systems are critical to specific failure types, while they are less critical to other failures [20]. Therefore, this research will allow new applications in safety-critical systems that distinguish among different failure severities. Further research will allow to estimate the failure severity of a component based on its location and connectivity in an architecture. This will help in identifying the components that are critical to system reliability.

Acknowledgments

This research work is partially funded by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

1. Abdelmoez, W., Nassar, D.M., Shereshevsky, M., Gradetsky, N., Gunnalan, R., Ammar, H.H., Yu, B., Mili, A.: Error propagation in software architectures. In: Proceedings of the 10th IEEE International Symposium on Software Metrics (METRICS 2004), Morgantown, WV, USA, September 2004, pp. 384–393 (2004)
2. Allen, R., Douence, R., Garland, D.: Specifying and Analyzing Dynamic Software Architectures. In: Astesiano, E. (ed.) ETAPS 1998 and FASE 1998. LNCS, vol. 1382, pp. 21–37. Springer, Heidelberg (1998)
3. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. In: Proceedings of the IEEE Transactions on Dependable and Secure Computing, January 2004, vol. 1, pp. 11–33 (2004)
4. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley, Reading (2003)
5. Candea, G., Delgado, M., Chen, M., Fox, A.: Automatic Failure-Path Inference: A Generic Introspection Technique for Internet Applications. In: Proceedings of the 3rd IEEE Workshop on Internet Applications (WIAPP 2003), California, USA, June 2003, pp. 132–141 (2003)
6. Cheung, L., Roshandel, R., Medvidovic, N., Golubchik, L.: Early Prediction of Software Component Reliability. In: Proceedings of the 30-th International Conference on Software engineering (ICSE 2008), Leipzig, Germany, May 2008, pp. 111–120 (2008)
7. Cortellessa, V., Grassi, V.: A Modeling Approach to Analyze the Impact of Error Propagation on Reliability of Component-Based Systems. In: Schmidt, H.W., Crnković, I., Heineman, G.T., Stafford, J.A. (eds.) CBSE 2007. LNCS, vol. 4608, pp. 140–156. Springer, Heidelberg (2007)
8. Everett, W.W.: Software component reliability analysis. In: Proceedings of the IEEE Symposium on Application - Specific Systems and Software Engineering and Technology, Washington, DC, USA, pp. 204–211 (1999)
9. Gacek, C., De Lemos, R.: Architectural Description of Dependable Software Systems. In: Besnard, D., Gacek, C., Jones, C. (eds.) Proceedings of Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective, pp. 127–142. Springer, Heidelberg (2006)
10. Georgiadis, I., Magee, J., Kramer, J.: Self-Organizing Software Architectures for Distributed Systems. In: Proceedings of the 1st workshop on Self-healing systems, Charleston, South Carolina, USA, pp. 33–38 (2002)
11. Goseva-Popstojanova, K., Trivedi, K.S.: Architecture based approach to reliability assessment of software systems. Proceedings of the International Journal on Performance Evaluation 45, 179–204 (2001)
12. Grunske, L.: Identifying “Good” Architectural Design Alternatives with Multi-Objective Optimization Strategies. In: Proceedings of the 28th International conference on Software engineering (ICSE 2006), China, pp. 849–852 (2006)
13. Hamlet, D., Mason, D., Woitman, D.: Theory of Software Reliability Based on Components. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001), Toronto, Ontario, Canada, May 2001, pp. 361–370 (2001)
14. Hiller, M., Jhumka, A., Suri, N.: An Approach for Analysing the Propagation of Data Errors in Software. In: Proceedings of the IEEE International Conference on Dependable Systems and Networks, Goteborg, Sweden, July 2001, pp. 161–170 (2001)

15. Jansen, A., Bosch, J.: Software Architecture as a Set of Architectural Design Decisions. In: The 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005), The Netherlands, pp. 109–120 (2005)
16. Komiya, S.: A model for the recording and reuse of software design decisions and decision rationale. In: Proceedings of the 3rd International Conference on Software Reuse: Advances in Software Reusability (ICSR 1994), Rio de Janeiro, Brazil, November 1994, pp. 200–201 (1994)
17. Kramer, J.: Configuration programming – A framework for the development of distributable systems. In: Proceedings of the IEEE International Conference on Computer Systems and Software Engineering (CompEuro 1990), Israel, pp. 1–18 (1990)
18. Leveson, N.G.: Software Safety: Why, What, and How. In: Proceedings of ACM Computing Surveys (CSUR) archive, June 1986, vol. 18, pp. 125–163 (1986)
19. Littlewood, B., Strigini, L.: Software Reliability and Dependability: A Roadmap. In: Proceedings of the 22-nd IEEE International Conference on Software Engineering on the Future of Software Engineering (ICSE 2000), Limerick, Ireland, pp. 175–188 (2000)
20. Mohamed, A., Zulkernine, M.: Improving Reliability and Safety by Trading Off Software Failure Criticalities. In: Proceedings of the 10-th IEEE International Symposium on High Assurance System Engineering (HASE 2007), Dallas, Texas, USA, November 2007, pp. 267–274 (2007)
21. Mohamed, A., Zulkernine, M.: On Failure Propagation in Component-Based Software Systems. In: Proceedings of the 8-th IEEE International Conference on Quality Software (QSIC 2008), Oxford, UK, August 2008, pp. 402–411 (2008)
22. Object Management Group, OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.1.2, OMG Available Specification without Change Bars, formal/2007-02-05 (November 2007)
23. Philippe, K.: An ontology of architectural design decisions in software intensive systems. In: The 2-nd Groningen Workshop on Software Variability, Groningen, The Netherlands, December 2004, pp. 54–61 (2004)
24. Popic, P., Desovski, D., Abdelmoez, W., Cukic, B.: Error propagation in the reliability analysis of component based systems. In: Proceedings of the 16-th IEEE International Symposium on Software Reliability Engineering (ISSRE 2005), Morgantown, WV, USA, November 2005, pp. 53–62 (2005)
25. Pullum, L.L.: Fault Tolerance Techniques and Implementation. Artech House (2001), ISBN 1-58053-470-8
26. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, Reading (1998), ISBN 0-201-17888-5
27. Tang, A., Ali Babar, M., Gorton, I., Han, J.: A Survey of Architecture Design Rationale. Proceedings of the Journal of Systems and Software 79, 1792–1804 (2006)
28. Voas, J., McGraw, G., Ghosh, A., Miller, K.: Glueing Together Software Components: How Good Is Your Glue? In: Proceedings of the Pacific Northwest Software Quality Conference, Portland, Oregon, USA, October 1996, pp. 338–349 (1996)
29. Walter, M., Trinitis, C., Karl, W.: OpenSESAME: An Intuitive Dependability Modeling Environment Supporting Inter-Component Dependencies. In: Proceedings of the Pacific Rim International Symposium on Dependable Computing, Seoul, Korea, December 2001, pp. 76–83 (2001)
30. Wang, Q.: Towards a Rule Model for Self-Adaptive Software. In: Proceedings of the ACM SIGSOFT Software Engineering Notes, vol. 30, pp. 8–12 (2005)
31. Weihang, W., Kelly, T.: Safety Tactics for Software Architecture Design. In: Proceedings of the 28-th Annual International Conference on Computer Software and Applications (COMP-SAC 2004), York University, United Kingdom, September 2004, pp. 368–375 (2004)

Integrating Fault-Tolerant Techniques into the Design of Critical Systems*

Ricardo J. Rodríguez and José Merseguer

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, Zaragoza, Spain
{rjrodriguez,jmerse}@unizar.es

Abstract. Software designs equipped with specification of dependability techniques can help engineers to develop critical systems. In this work, we start to envision how a software engineer can assess that a given dependability technique is adequate for a given software design, i.e., if the technique, when applied, will cause the system to meet a dependability requirement (e.g., an availability degree). So, the idea here presented is how to integrate already developed fault-tolerant techniques in software designs for their analysis. On the one hand, we will assume software behavioural designs as a set of UML state-charts properly annotated with profiles to take into account its performance, dependability and security characteristics, i.e., those properties that may hamper a critical system. On the other hand, we will propose UML models for well-known fault-tolerant techniques. Then, the challenge is how to combine both (the software design and the FT techniques) to assist the software engineer. We will propose to accomplish it through a formal model, in terms of Petri nets, that offers results early in the life-cycle.

1 Introduction

Software failures chronically occur and in most cases do not cause damage. However, a system is called *critical* when failures result in environmental damage (*safety-critical*), in a non-achieved goal compromising the system (*mission-critical*) or in financial losses (*business-critical*). Avizienis et al. [1] cleverly identified the fault-error-failure chain to support specification of intricacies occurring in critical systems.

Fault prevention and fault tolerance, as two of the means to attain dependability [1], have to be considered by designers of critical systems. The former, for example, by means of quality control techniques, while the latter may take the form of replication: distribution through replication confers tolerance to the system and allows to get a higher system availability.

This paper addresses the issue of integrating already developed fault-tolerant (FT) techniques into software designs for their analysis through automatically

* This work has been supported by the European Community's Seventh Framework Programme under project DISC (Grant Agreement n.INFSO-ICT-224498) and by the project DPI2006-15390 of the Spanish Ministry of Science and Technology.

obtained formal models. The aim is to evaluate the effectiveness of a given FT technique for a concrete software design, i.e., to verify if the design meets dependability requirements using such FT technique. FT techniques have to be efficaciously integrated with other system requirements, and this will be accomplished through the software design. In fact, we propose to represent a given FT technique as a UML (Unified Modelling Language [2]) model with well-defined interfaces. So, the software design of the critical system under analysis, also modelled with UML, will be equipped with the FT while appropriate interfaces are provided. The software design and the FT technique are eventually converted into blocks of Petri nets [3] (using well-known translation approaches [4,5]) and composed to get the desired analysable model that will report results about system dependability.

The analysis of dependability requirements compels to capture dependability properties (e.g., fault or failure description), which should be expressed in UML designs assuming that we desire to free the software engineer from the manual generation of the formal model. On the one hand, in this work we use DAM [6] (Dependability Analysis and Modelling profile) for this purpose. DAM is a MARTE [7] (Modelling and Analysis of Real-Time and Embedded systems profile) specialisation, that will be useful to complement the dependability properties with performance ones. On the other hand, since we focus our work in the context of intrusion-tolerant systems (i.e., those critical systems which apply FT techniques to tolerate intrusions), this implies also the necessity to report security requirements in the same UML designs. So, to avoid greater complexities, we rely on SecAM [8] (Security Analysis and Modelling profile), which is properly integrated in the MARTE-DAM framework. Although it may seem that the use of these profiles may bring some knottiness, in reality, a little part of the stereotypes proposed by the above mentioned profiles greatly helps designers of critical systems in their work.

The balance of the paper is as follows. Section 2 introduces the basis of the paper, i.e., some FT techniques we will use to illustrate the proposal and UML profiles. Section 3 presents UML and formal models for these FT techniques. Section 4 describes the UML design of an example and illustrates how the FT techniques can supplement it, moreover it shows how to obtain a final system formal model. Finally, related work and some conclusions are given in Section 5.

2 Previous Concepts

Before starting the contribution, we summarise in Section 2.1 the kind of FT techniques used along this work and in Section 2.2 we recall our proposal of a security profile in the context of MARTE-DAM, i.e., SecAM [8].

2.1 Proactive and Reactive Techniques

Modern critical systems (e.g., CRUTIAL [9]) incorporate fault prevention and fault-tolerant techniques to get a more robust system protected against faults.

They are known as *intrusion-tolerant systems* when protection mainly concerns with faults coming from intrusions. Fault-tolerant techniques can be subdivided [1] in several groups: fault detection, fault recovery, fault handling and fault masking. In this work, we focus on proactive and reactive fault-tolerant recovery techniques.

An extensive review of the application of fault-tolerant techniques to the security domain can be found in [10]. It is also worth mentioning the work in [11], which suggests an approach to the design of highly secure computing systems based on fault-tolerant techniques. An interesting example of application can be found in [12], where the authors propose a technique for fragmenting the data and then storing the pieces in different locations (as RAID technology actually works), which reduces losses of data in case of intrusion.

Proactive recovery transforms a system state containing one or more errors (or even visible faults) into a state without detected errors or faults. Proactive techniques were presented in [13] as a long-term protection against break-ins and implemented, for example, in the scope of an on-line certification-authority system [14]. These techniques borrow ideas from proactive discovery protocols (e.g., IPsec [15]), session-key refreshment (SSL/TLS [16]) or secret sharing algorithms [17]. Hence, proactive security is defined as a combination of periodic refreshment and distribution [18,19].

Following Avizienis et al.'s fault taxonomy [1], *reactive recovery* can be classified as a fault-tolerant technique: it does a concurrent error detection, that is, errors in the system are detected meanwhile it is working. Then, a detection implies some actions must be performed in order to recover the system to a free-error state.

Proactive and reactive recovery techniques should not be considered as mutually exclusive but as complementary. Briefly, proactive techniques are worried about fault prevention (passive part of the system), while reactive ones are concerned with fault removal (active part). Sousa et al. presented in [20] a real application of proactive and reactive recovery techniques to an existing critical system, which tolerates up to f failure nodes and is able to recover in parallel up to k nodes. The rationale behind this idea is a scheduled time-line, which will be modelled in Section 3 (Fig. 1, adapted from [20], depicts it) and it is explained in the following.

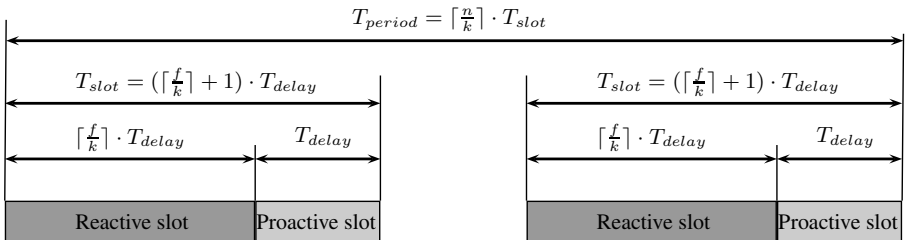


Fig. 1. Schedule time-line showing activations of reactive and proactive recoveries

A system with n distributed devices is initially divided into $\lceil \frac{n}{k} \rceil$ groups, containing each one up to k devices, being k the number of simultaneous recoveries the system can support. Assuming a period of time T_{period} , then each one is divided in $\lceil \frac{n}{k} \rceil$ slices (called T_{slot} from now on) where both (i.e., proactive and reactive) recoveries have to be performed. In a T_{slot} , one proactive recovery will be activated for a selected group which has a duration equal to T_{delay} , being T_{delay} the maximum expected time for recovering a device. Regarding reactive recovery, if we assume up to f failures and k simultaneously recoveries, that implies a maximum of $\lceil \frac{f}{k} \rceil$ reactive activations may happen in a T_{slot} . As can be inferred, T_{slot} has a duration equal to $(\lceil \frac{f}{k} \rceil + 1) \cdot T_{delay}$. There exists a relation [20] between values of n , f and k as is shown in Equation 1.

$$n \geq 2 \cdot f + k + 1 \quad (1)$$

A deeper description of the schedule time-line for proactive and reactive recoveries, as well as justification for inequality shown in Equation 1, can be found in [21].

2.2 The Security Analysis and Modelling (SecAM) Profile

The UML [2] (Unified Modelling Language) is a standard and comprehensive language that allows to specify functional software requirements through diagrams from architectural to deployment system views. UML can be tailored for analysis purposes through profiling. A *profile* defines stereotypes and tagged values for annotating design model elements extending its semantic. In particular, the Modelling and Analysis of Real-Time and Embedded systems (MARTE) [7] profile enables UML to support schedulability and performance analysis for real-time (RT) and embedded systems. Although focussed on RT, MARTE sub-profiles for performance and schedulability have also been proved useful in a wide range of other application domains. Performance as a Non-Functional Property (NFP) is specified in the MARTE context according to a well-defined Value Specification Language (VSL) syntax. Recently, the non-standard Dependability Analysis and Modelling (DAM) [6] profile was introduced to address dependability also as a NFP in UML design models. Indeed, as DAM is a MARTE specialisation, they can play together to specify performance and dependability NFPs in UML models. The entire set of MARTE stereotypes can be found in [7], while DAM stereotypes, including UML meta-classes that the stereotypes can be applied to, can be found in [6].

The close relation among dependability and security, cleverly disclosed by Avizienis et al. [1], was an argument in [8] for developing a new profile, called Security Analysis and Modelling (SecAM), to model and analyse security NFPs. Currently, the SecAM profile only addresses the topic of resilience, although its design favours easy integration of other security concerns. As the SecAM profile was constructed on top of DAM (indeed, as its specialisation), a joint DAM-SecAM specification on a UML design allows to accomplish a comprehensive dependability and security specification of system NFPs. The work here presented relies on the DAM-SecAM relation: we aim to specify fault-tolerant techniques, a dependability issue, for intrusion-tolerant systems, a security issue.

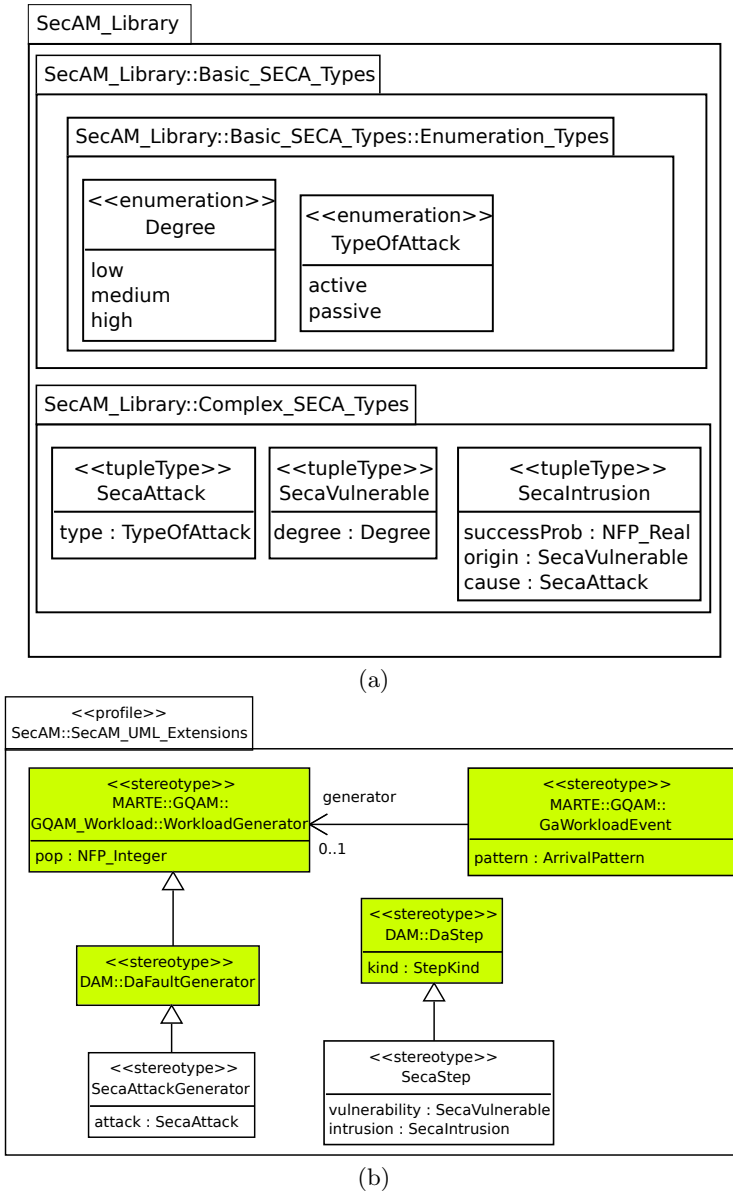


Fig. 2. (a) SecAM library and (b) SecAM UML extensions

Figure 2(b) depicts some SecAM stereotypes used in this work, concretely `secaStep` and `secaAttackGenerator`. The SecAM library (Fig. 2(a)) describes the types associated to the tagged values of these stereotypes. The `secaStep` stereotype inherits from `DAM::daStep` stereotype and is meant to describe a system vulnerability or an attack, being both security faults [8]. For description

of system malicious intrusions, the `secaAttackGenerator` stereotype is introduced, besides, the MARTE and DAM classes it specialises (Fig. 2(b)) allow to describe the occurrence probability pattern of the intrusion. `DaFault` DAM annotation, later used in this work, supports fault definition in [1] and means the basis for the actual SecAM annotations.

3 Modelling Proactive and Reactive Recovery Techniques

We develop, in this section, a generic and reusable model of proactive and reactive recovery techniques. In first term, we model them using UML state-machine (UML-SC) diagrams annotated with the previously discussed profiles. Then, we obtain a Coloured Petri Net (CPN) [22] which maps the behaviour of these UML diagrams. In fact, this CPN accurately represents proactive and reactive recovery techniques. Our intention is then to reuse such CPN through different software designs to conclude about the appropriateness of the techniques for the design, Section 4 will show an example. To accomplish this target, these software designs will also be modelled using UML-SC and each one will be eventually converted into a CPN. So, our proposal to reuse the “proactive-reactive” CPN within a given software design has to offer adequate “interfaces” to compose both CPNs. Then, we finally get a CPN that embeds both the proactive-reactive techniques and the software design as explained in Section 4.

3.1 UML Modelling

We have distinguished two components, one in charge of controlling the scheduled time-line presented in Section 2, and the other controlling the device to be recovered. The latter has been called Proactive and Reactive Recovery (PRR) component following terminology in [20].

Schedule controller UML state-chart is depicted in Figure 3. Initial analysis variables (`gaAnalysisContext` stereotype) are: `tDelay`, which determines the duration of each recovery; `f`, number of faulty devices allowed; and `k`, number of devices recovered in parallel. Only one controller will be placed in the system (tag `pop` of `gaWorkloadGenerator` stereotype). Once created, it calculates in `g` the first group which will be proactively recovered. Upon entrance into *Reactive slot* state, it invokes event `nextSchedule()` for PRR devices in `g` to inform them that the components they control will be proactively recovered in the next proactive slot, so their monitoring activity will not be necessary since for sure they will be recovered. Then, it starts the `countDown()` activity with duration `hostDemand` equals to $\lceil \frac{f}{k} \rceil \cdot tDelay$ seconds (that is, it makes room for up to $\lceil \frac{f}{k} \rceil$ parallel recoveries). Completion of `countDown()` activity means to schedule elements in `g` and to change to *Proactive slot* state, where all PRR devices are disabled and it starts the proactive `countDown()` activity, in this case with a duration equal to `tDelay` seconds. Once finished, it enables all PRR devices and before entering again in the *Reactive slot* state, it calculates the next proactive group.

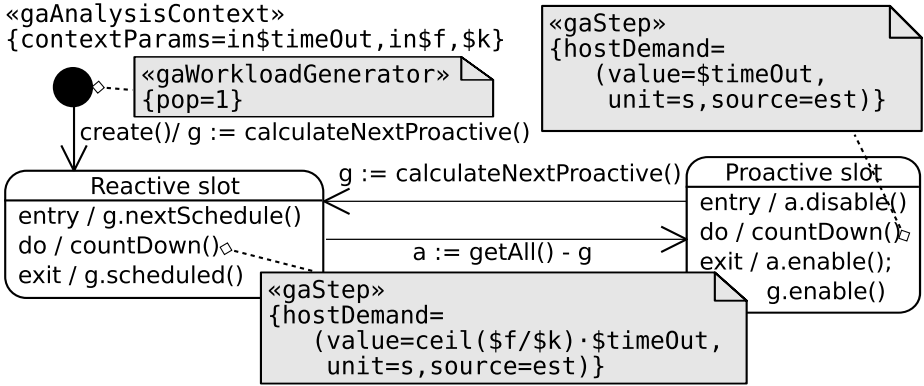


Fig. 3. Scheduler UML state-machine diagram

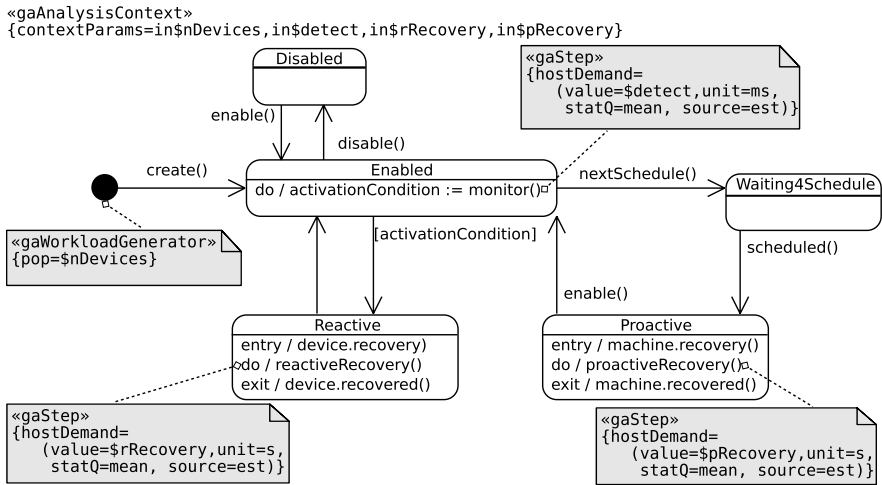


Fig. 4. PRR controller UML state-machine diagram

Figure 4 shows UML-SM for PRR component controller. Obviously, the population is equal to the number of effectively monitored devices, $nDevices$. It starts in *Enabled* state and executing the activity `monitor()`, which abstracts two processes: 1) detection of errors in the monitored device and 2) checking for room in current time slot for a reactive recovery. So, when it positively informs, then enters in *Reactive* state to perform a recovery (`reactiveRecovery()` activity), which has a duration of $rRecovery$ seconds on average. Once finished, it comes back to *Enabled* state. From there, event `nextSchedule()` evolves to *Waiting4Schedule* state, where the PRR will wait for event `scheduled()` invoked by the scheduler to start the proactive recovery. In both recovery states (*Reactive* or *Proactive*) the PRR invokes upon the entrance (`recovery()`) and on the exit

(`recovered()`) events in the monitored device switching it off/on, respectively. Finally, note that events `enable()` and `disable()` received from the scheduler effectively prevent the PRR to monitor its device.

3.2 Formal Modelling through Petri Nets

Following ideas in [5] we obtained two Generalized Stochastic Petri Nets (GSPNs) [23] by model transformation of UML design (Figs. 3 and 4). Considering that ideas proposed in [5] were given for performance analysis purposes, some minor changes have emerged. Indeed, we used the ArgoSPE [24] tool, which implements the algorithm given in [5], to perform the transformation of UML-SCs annotated with SPT [25] (Schedulability, Performance and Time profile, precursor of MARTE) into GSPNs. Seeing that ArgoSPE does not support MARTE, nor DAM nor SecAM profiles, the GSPNs obtained from the transformation have been manually modified to incorporate such annotations. In the following we summarise the algorithm implemented in ArgoSPE.

Each SC simple state is transformed in a PN place, which represents its entrance. The latter is followed by two causally connected PN transitions which represent, respectively, the entry action and the do-activity of the SC state. Entry actions are modelled by immediate PN transitions (assuming its execution time is negligible) while do-activities are represented by timed PN transitions, which are characterised by one output place (i.e., the completion place) modelling the SC completion state. If the SC state has outgoing immediate transition, this is translated into a PN immediate transition with the completion place as its input place. For conflicting outgoing transitions, the transformation adds immediate transitions with probabilities to stochastically resolve the choice. In this case, the probability values are taken from the annotations attached to the transitions.

In order to take advantage of the hierarchy and symmetries in the problem, and then gaining readability, we slightly modified these semi-automatically obtained Petri nets to gain an equivalent Coloured Petri Net (CPN) [22]. Figure 5(b,c) depicts these CPNs, while Figure 5(a) offers a hierarchical CPN [26]

Table 1. CPN initial marking, token colour definition and functions

Token colour definitions	Initial marking
<i>type</i> D is $\{1 \dots nDevices\}$	$m_0(Enable) = \sum_{i \in D} i$
<i>type</i> G is $\{G_1 \dots G_{\lfloor \frac{nDevices}{k} \rfloor}\}$	$m_0(nextGroup) = G_1$
<i>subtype</i> G_i is $\{(k \cdot (i - 1) + 1) \dots k \cdot i\}$	$m_0(Idle) = 1$
<i>var</i> $i : D, g : G$	$m_0(maxParallel) = k$
Functions definitions	
$belonging(g : G) = \sum_{i \in G} i$	
$cSubset(g : G) = \sum_{i \in D i \ni g} i$	
$allDevices() = \sum_{i \in D} i$	

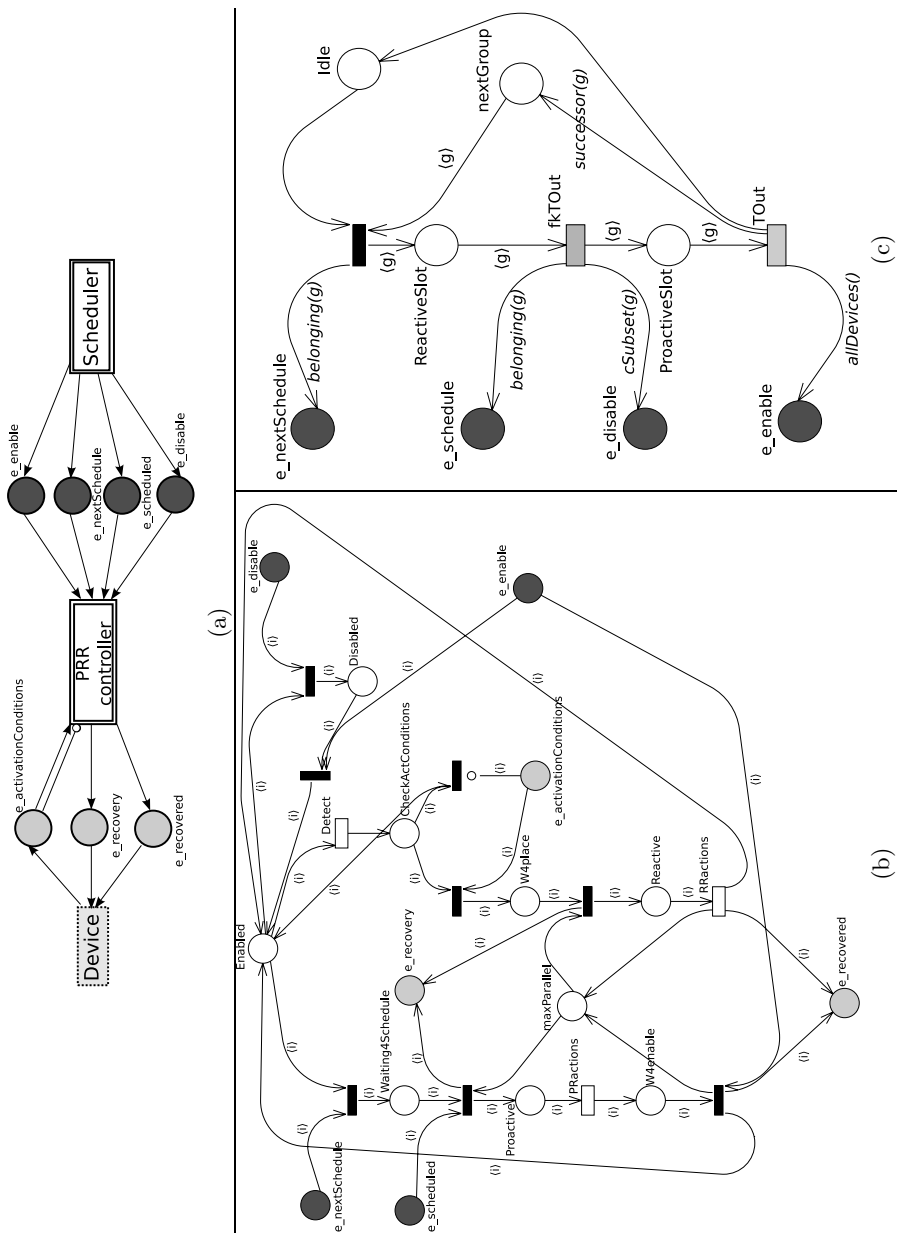


Fig. 5. (a) Hierarchical CPN, (b) CPN of PRR controller and (c) CPN of scheduler

view for an easy understanding of interactions between each subnet. Interactions occur via event places (*e_enable*, *e_nextSchedule*, *e_schedule* and *e_disable*), which model the real interfaces among components. Controlled devices are also depicted (light gray box) just to highlight their communication with the PRR controllers. CPNs in Figure 5(b,c) depict deterministic delays through grey transitions, stochastic delays through white transitions, while black transitions are immediate ones. The initial marking, token colour and functions definition are summarised in Table 1.

The CPN of scheduler SC (Figure 5(c)) has an initial marking represented in places *Idle* and *nextGroup*, with values indicated in Table 1, which come from *gaWorkloadGenerator* annotation and *calculateNextProactive()* function in the SM. Transitions *fkTOut* and *TOut* represent the reactive and proactive countdowns and are characterised by deterministic durations given in the corresponding *gaStep* SC annotations. Note that the firing of *TOut* enables all PRR controllers (through place *e_enable*), generates the next group which will be proactively recovered and starts up the cycle again.

Regarding CPN of PRR controller (Figure 5(b)), its initial marking in places *Enabled* and *maxParallel* represent the number of PRR devices in the system and the maximum number of devices the system can recover in parallel, according to annotations in the SC. Firing of transitions *PRactions* and *RRactions* respectively lead the activation of proactive and reactive recovery. Monitored devices are informed about the starting and ending of both recoveries (proactive and reactive) through places *e_recovery* and *e_recovered*. Transition *Detect* abstracts the activity *monitor()* in the SC, which once fired checks activation conditions to, in positive case, inform the device about the starting of the reactive recovery. Note that this can take place only if there exist room enough for a new parallel recovery.

4 Example: On-Line Shopping Website

Proactive and reactive recovery techniques such as the ones here described, but also many others, can be implemented in critical systems. However, it would be highly interesting to assess their actual convenience for a given system before to carry out them physically. Thereby, we think that the models developed in previous section can be useful for this purpose and with this thought in mind, we show in this section how a software design can offer interfaces through which eventually it will be combined, at UML level, with the FT target techniques. The example tries to be a blueprint about how to:

- (a) add proactive and reactive techniques to a critical system for improving its fault tolerance;
- (b) obtain an analysable formal model and
- (c) get results from such model that can assess system dependability.

We model a business-critical system of the kind of an on-line shopping website, where a balance loader is in charge of placing customers in several servers. Each server manages a defined number of customers in parallel. A physical view of the

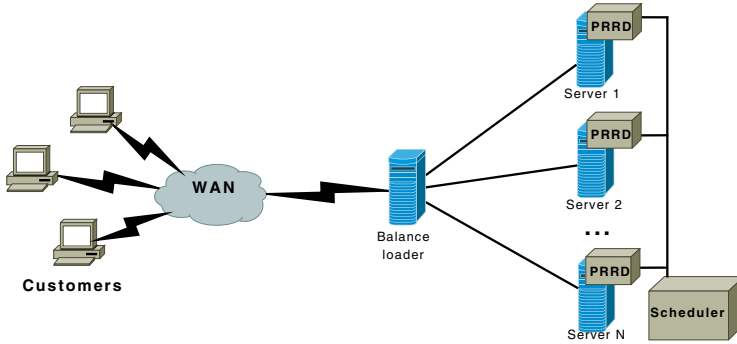


Fig. 6. Physical view of the system

system is depicted in Figure 6. Note that it incorporates an external PRR device (assumed tamper-proof and not subjected to failures) that embeds proactive and reactive recovery techniques. The system also features the scheduler device wired to PRRDs. Even achieving a complete failure prone device it is not a reality, this kind of device can be seen as an embedded tamper-proof device, that is, there is no possibility of deliberate altering or adulteration of the device. The addition of other FT techniques (e.g., replication, redundancy or diversity) to PRR devices will still fit within the techniques presented in this paper because the effort should be done in the modelling of the interaction between techniques.

4.1 UML Modelling

Figure 7 depicts the behaviour of the balance loader, of which only one copy exists (`pop` tagged value) and starts in *Idle* state. An open workload generates customer's requests (`gaWorkloadGenerator` stereotype) with an inter-arrival time defined as an exponential distribution. Customer arrivals provoke this component to execute an algorithm (`balanceLoader()`) that ends up asking a server to attend the new customer. This component does not interplay with the target techniques, so no interface is required. However, its behaviour is mandatory to get some system parameters such as workload (see `gaWokloadEvent` annotation).

Server state-machine diagram is depicted in Figure 8. Initial pseudo-state indicates `nDevices` servers ready in the system. Each *available* server can concurrently attend up to `nThreads` started up through event `attendCustomer()`, which indeed initiates a sub-state machine specified in Figure 9. The server has been supplied with interfaces (`recovery()` and `recovered()`) to interact with the PRR component via events (note that here is where we incorporate the recovery techniques into the system). Consequently, the actual functional behaviour of a server is specified in the sub-state machine, which inherits the interfaces then allowing to abort normal behaviours. Besides, we have wanted to show how other kinds of faults, e.g. hardware faults, can also be expressed within this

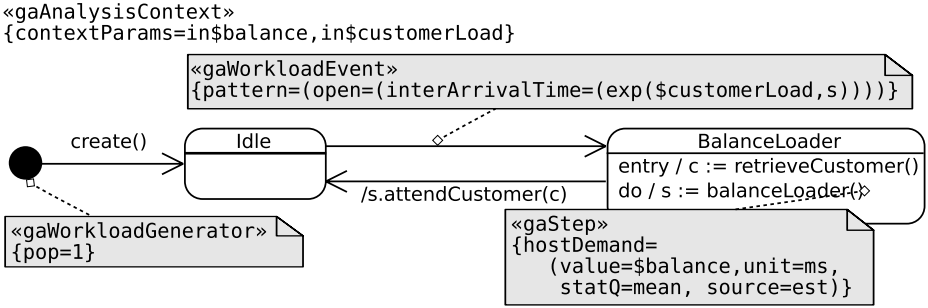


Fig. 7. Balance loader UML state-machine diagram

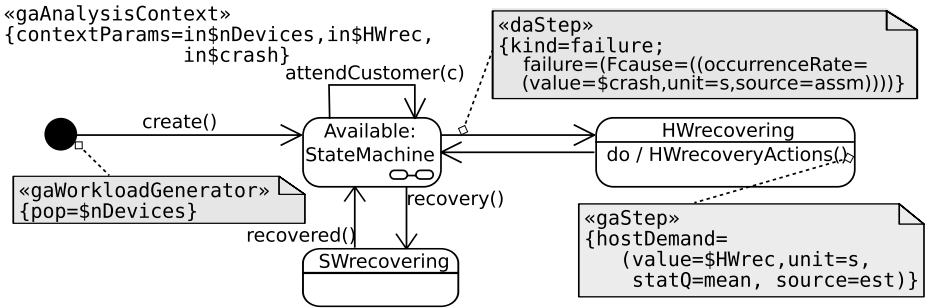


Fig. 8. Server UML state-machine diagram

modelling approach. So, when a hardware crash occurs (**daStep** annotation in Fig. 8) it will be properly handled, of course the resulting formal model will also embed this kind of fault.

During normal behaviour (Fig. 9) a server can be attacked and/or suffer intrusions. In the example we have reduced, as much as possible, the specification of system normal behaviour (*Processing*) to focus on the *critical* part. Hence, the customer’s requests (**attendCustomer**) can be a source of attack (see **secaAttackGenerator** annotation) and occasionally become an intrusion (**secaStep** annotation), i.e., the attack successes. Obviously, other kind of dependability faults could be here specified by means of DAM-SecAM. A final remark to point out that conflicting outgoing transitions of *Processing* state are evidently solved by the probabilities in the annotations.

4.2 Formal Modelling

Again, following ideas in [5] and assisted by ArgoSPE [24] tool, we obtained GSPNs by model transformation of UML design (Figs. 7, 8 and 9). Thereafter, the nets were composed by interface places (**e_attendCustomer**), simplified and converted into a CPN (depicted in Fig. 10) for readability purposes. The initial marking and transition rates are summarised in Table 2.

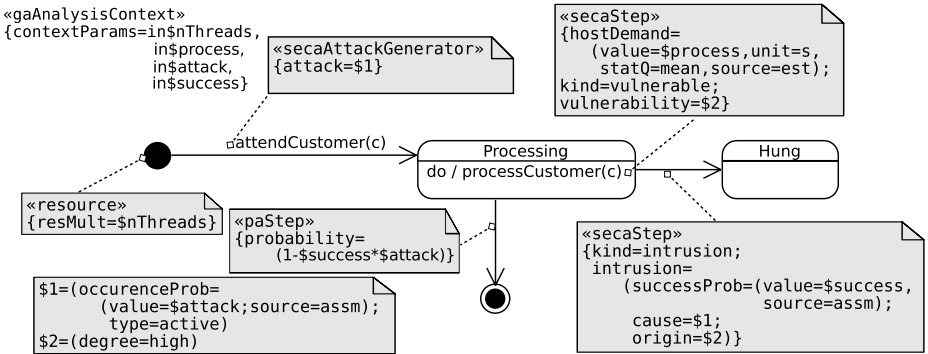


Fig. 9. Available UML sub-state machine diagram

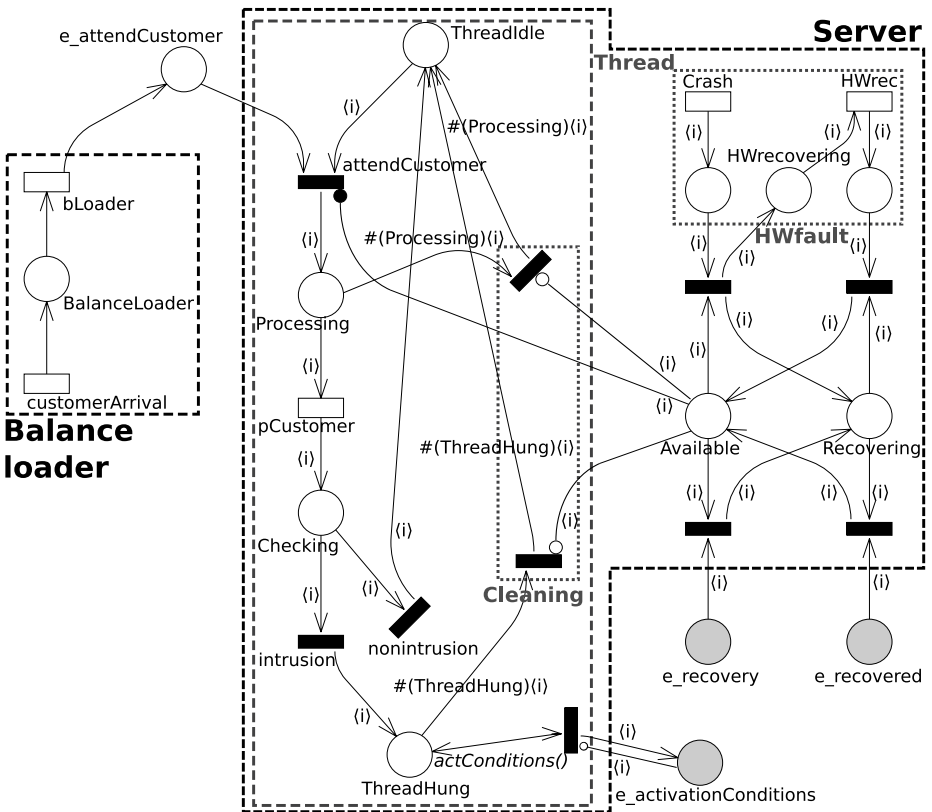


Fig. 10. CPN of case study

Interface light grey places allow combination with the PRR component as depicted in Figure 5(a), so to gain the target Petri net that models both: system behaviour and recovery techniques. Now we can discuss the role of interface

Table 2. (a) Example parameters and (b) experiments parameters

Initial marking		Parameters	Value
$m_0(\text{ThreadIdle}) = n\text{Threads} \cdot \sum_{i \in D} i$ $m_0(\text{Idle}) = \sum_{i \in D} i$		<i>nDevices</i>	12
		<i>k</i>	2, 3, 4
		<i>f</i>	1
		<i>timeOut</i>	120, 180s
		<i>detect</i>	100 ms
		<i>pRecovery</i>	120 s
		<i>rRecovery</i>	120 s
		<i>nThreads</i>	10
		<i>crash</i>	432000 s
		<i>HWrec</i>	43200 s
		<i>balance</i>	200 ms
		<i>customerLoad</i>	0.5 customers/s
		<i>process</i>	300 s
		<i>attack</i>	30%
		<i>success</i>	0% ... 75%

Transition	Parameter (type)
customerArrival	1/ <i>customerLoad</i> (rate)
bLoader	1/ <i>balance</i> (rate)
pCustomer	1/ <i>process</i> (rate)
Crash	1/ <i>crash</i> (rate)
HWrec	1/ <i>HWrec</i> (rate)
intrusion	<i>attack</i> · <i>success</i> (weight)
nonintrusion	1 – <i>attack</i> · <i>success</i> (weight)

(a)
(b)

place `e_activationConditions`, but firstly remember that a token in this place means for the PRR component to activate a reactive recovery. From the point of view of our system, we desire to activate the reactive recovery whenever 7 out of 10 threads (see *nThreads* variable in Table 2(b)) in a given server become hung. So, the `actConditions()` function in the test arc of place *ThreadHung* implements the algorithm that checks out such condition and when it is true then a token is placed in place `e_activationConditions`. Obviously, other systems should implement this function differently, but always preserving place `e_activationConditions` as an interface place.

In *Balance loader* area, the system open workload is represented by an exponentially distributed transition `customerArrival` of mean *customerLoad* (taken from `gaWorkloadEvent` stereotype, Fig. 7). In the *Thread* area, transitions `intrusion` and `nonintrusion` represent whether if an attack had success or not, respectively. Finally, we manually added the part of the net called *Cleaning* (indeed, composed only by two transitions) to remove tokens from `Processing` and `ThreadHung` as long as the server becomes not available (arc inscription $\#P_i$ means all tokens in place P_i).

4.3 Analysis and Assessment

The analysis was carried out using simulation programs of GreatSPN [27] tool. We actually simulated the original GSPNs obtained by ArgoSPE instead of the readable CPNs in Figures 5 and 10. Simulation parameters were set to a confidence level of 99%, accuracy of 1%, length of evolution phase equal to 604800 time units and a length of initialisation phase equal to 86400 time units. The Petri nets parameters and its values were summarised in Table 2(b) (note

Table 3. Analysis parameters required in UML statecharts

Input parameter	Provided by	Annotation (profile)
balance	Manufacturer	gaWorkloadEvent (MARTE)
customerLoad	Designer	gaStep (MARTE)
nDevices	Designer	gaWorkloadGenerator (MARTE)
Hwrec	Manufacturer	gaStep
crash	Manufacturer	daStep (DAM)
nThreads	Manufacturer	resource (MARTE)
process	Manufacturer	secaStep (SecAM)
attack	Designer	secaAttackGenerator (SecAM), paStep (MARTE)
success	Designer	secaStep (SecAM), paStep (MARTE)

that the number of servers was 12 (`nDevices`), each one able to attend up to 10 customers in parallel (`nThreads`)).

All values of the parameters can be known at design time: some of them, such expected customer load, probabilities of attack and success, should be estimated by the software engineer, while other parameters, such as time performing recovery actions should be given by manufacturer of PRR device. Table 3 summarises input analysis parameters and by whom they should be provided.

In the experiments, reactive recovery is always performed when the number of active threads for a device drops to 3 (function `activationConditions()`). Regarding proactive recovery, 12 servers allow setting several configurations that we have tested: *three proactive groups* of four servers (solid lines in Figure 11), *four proactive groups* of three servers (dot-dashed lines) and *six proactive groups* of two servers (dashed lines). Under these parameters, we have simulated the net to point out the best configuration among the previous ones w.r.t. throughput. In Figure 11 we show the relation between the incoming customer throughput (`customerLoad`) and the system throughput ($Thr(attendCustomer)$). The horizontal axis represents the percentage of successful attacks (0%..75%, variable `success`), having the percentage of system attacks (variable `attack`) set to 30% for all the experiments.

The results indicate that the more servers are simultaneously recovered, the more throughput the system obtains. In terms of absolute time, smaller groups recover more number of servers than bigger groups, which ensures higher availability for the formers and consequently better performance. In the example, it could be assessed that groups of three serves are the right choice regarding throughput. The computed measure is a *performability* one (i.e., performance in the presence of faults). Although some other interesting results were obtained (e.g., results of dependability nature) from this formal model, we do not present them since this is not the main focus of the paper.

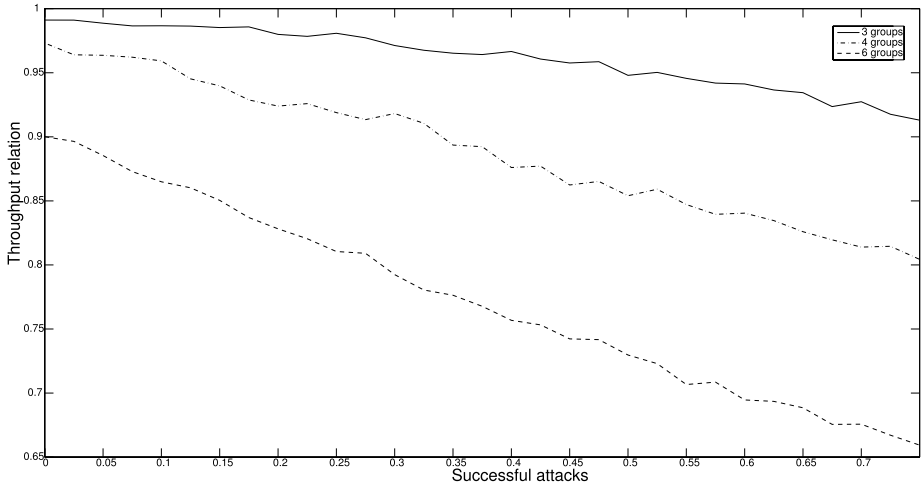


Fig. 11. Simulation results

5 Related Work and Conclusion

Several approaches [28,29,30] in the literature bring Petri nets for the design of critical systems. In [28] Heiner et al. used a combined model of Z and Petri Net formalisms, the first for specifying data and its evolution and the latter to validate the safety-critical system. The union of both formalisms allows to obtain an approach where data-combination and behaviour are described. Ghezzi et al. presented in [29] a high-level Petri Net formalism (TB nets, a particular case of *Timed Environment/Relationship* nets) to specify control, function and timing issues in time-critical systems. In [30], Houmb et al. quantified operational system integrity of security critical systems using the formalism of Coloured Petri Nets (CPN).

Regarding fault-tolerant techniques applied at software architectural level also several works can be found [31,32,33,4,34]. In [31] Harrison and Avgeriou studied how several fault-tolerant techniques can be carried out as best-known architectural patterns. By the use of architectural patterns they aim to directly create software architectures satisfying quality attributes. Nguyen-Tuong and Grimshaw presented in [32] a reflective model, called *Reflective Graph & Event* (RGE), which is applied for making failure-resistant applications. Using this reflective model they are able to express fault-tolerant algorithms as reusable components allowing composition with user applications. Rugina et al. propose in [33] an approach for system dependability modelling using AADL (Architecture Analysis and Design Language), being the design model transformed into GSPN. This approach was applied to an Air Traffic Control System. Bondavalli et al. [4,34] have a vast work in the area of translating UML diagrams into dependability models, having also used Petri nets as a target model in some of these works. Their proposal of translation could be used in this paper instead

of [5], but it should be taken into account that they propose an intermediate model as a first step.

In this paper, we have explored the idea of combining models that represent FT techniques and software behavioural designs. The combined model is useful for dependability assessment. Although the example has shown feasibility in the approach to integrate well-known recovery techniques into software designs, we are conscious that a long path has to be walked for the approach to reach applicability. So, we want to clearly establish that, from our point of view, the contribution of the paper is restricted to the achievements in the example, i.e., how to combine proactive and reactive techniques with a software design and their analysis. However, we are confident of the second one, i.e., reuse of the approach with other FT techniques. The key point is to gain a “library” of UML models representing FT techniques ready to use in critical designs. Being the crucial aspect for the UML model of a FT technique to have clearly defined its interfaces, we strongly believe that *events* and *conditions* are the means to attain it as we did in our proposal. Moreover, each technique has to define also how their interfaces play in the software design, for the case of the recovery techniques we have advocated for a superstate which offers suitable interfaces and embeds the system normal behaviour.

As a critic, we recognise that UML-SC should not be the only UML diagram used in this context, since for example, sequence diagrams may help the engineer in understanding system usage situations. So, we plan to extend our approach to take advantage of other UML diagrams. Another critic stems from the fact that the combination of FT techniques and software designs should be explicitly made at UML level, instead of deferring the combination to the Petri net models. This would bring advantage to the engineer for completely avoid the formal model. Being aware of this fact, we are working on a feasible solution to this problem.

The use of an approach such as the one here developed should otherwise bring several benefits from the point of view of a software engineer. The easy integration of FT techniques into software designs and the existence of such “library” may allow to test different techniques for the same design to find the ones fitting better. Such “library” will also free the engineer of worrying about how to model FT and concentrate on the problem domain. Finally, it is well-known that the use of formal models early in the life-cycle to prove requirements is less expensive than other approaches.

References

1. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. on Dependable and Secure Computing* 1, 11–33 (2004)
2. OMG: Unified Modelling Language: Superstructure. Object Management Group (July 2005) Version 2.0, formal/05-07-04
3. Murata, T.: Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE* 77, 541–580 (1989)

4. Bondavalli, A., Dal Cin, M., Latella, D., Majzik, I., Pataricza, A., Savoia, G.: Dependability Analysis in the Early Phases of UML Based System Design. *Journal of Computer Systems Science and Engineering* 16(5), 265–275 (2001)
5. Merseguer, J., Bernardi, S., Campos, J., Donatelli, S.: A Compositional Semantics for UML State Machines Aimed at Performance Evaluation. In: Giua, A., Silva, M. (eds.) *Procs. of the 6th Int. Workshop on Discrete Event Systems*, Zaragoza, Spain, October 2002, pp. 295–302. IEEE Computer Society Press, Los Alamitos (2002)
6. Bernardi, S., Merseguer, J., Petriu, D.: A Dependability Profile within MARTE. *Journal of Software and Systems Modeling* (2009), doi: 10.1007/s10270-009-0128-1
7. Object Management Group: A UML profile for Modeling and Analysis of Real Time Embedded Systems (MARTE) (November 2009), v1.0, formal/2009-11-02
8. Rodríguez, R.J., Merseguer, J., Bernardi, S.: Modelling and Analysing Resilience as a Security Issue within UML. In: *SERENE 2010: Procs. of the 2nd Int. Workshop on Software Engineering for Resilient Systems*. ACM, New York (2010) (accepted for publication)
9. Veríssimo, P., Neves, N.F., Correia, M., Deswarte, Y., Kalam, A.A.E., Bondavalli, A., Daidone, A.: The CRUTIAL Architecture for Critical Information Infrastructures. In: de Lemos, R., Di Giandomenico, F., Gacek, C., Muccini, H., Vieira, M. (eds.) *Architecting Dependable Systems V*. LNCS, vol. 5135, pp. 1–27. Springer, Heidelberg (2008)
10. Rushby, J.: *Critical System Properties: Survey and Taxonomy*. Technical Report SRI-CSL-93-1, Computer Science Laboratory, SRI International (1994)
11. Dobson, J., Randell, B.: Building Reliable Secure Computing Systems Out Of Unreliable Insecure Components. In: *IEEE Symposium on Security and Privacy*, p. 187. IEEE Computer Society, Los Alamitos (1986)
12. Fray, J.M., Deswarte, Y., Powell, D.: Intrusion-Tolerance Using Fine-Grain Fragmentation-Scattering. In: *IEEE Symposium on Security and Privacy*, p. 194. IEEE Computer Society Press, Los Alamitos (1986)
13. Canetti, R., Gennaro, R., Herzberg, A., Naor, D.: Proactive Security: Long-term Protection Against Break-ins. *CryptoBytes* 3, 1–8 (1997)
14. Zhou, L., Schneider, F.B., Van Renesse, R.: COCA: a Secure Distributed Online Certification Authority. *ACM Trans. on Computer Systems (TOCS)* 20(4), 329–368 (2002)
15. Tran, T.: Proactive Multicast-Based IPSEC Discovery Protocol and Multicast Extension. *MILCOM*, 1–7 (2006)
16. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, Internet Engineering Task Force (April 2006)
17. Shamir, A.: How to Share a Secret. *Communications of ACM* 22(11), 612–613 (1979)
18. Canetti, R., Halevi, S., Herzberg, A.: Maintaining Authenticated Communication in the Presence of Break-ins. In: *PODC 1997: Procs. of the 16th annual ACM symposium on Principles Of Distributed Computing*, pp. 15–24. ACM, New York (1997)
19. Ostrovsky, R., Yung, M.: How To Withstand Mobile Virus Attacks. In: *PODC 1991: Procs. of the 10th annual ACM symposium on Principles Of Distributed Computing*, pp. 51–59. ACM, New York (1991)
20. Sousa, P., Bessani, A., Correia, M., Neves, N., Verissimo, P.: Resilient Intrusion Tolerance through Proactive and Reactive Recovery. In: *Procs. of the 13th IEEE Pacific Rim Dependable Computing Conference*, pp. 373–380 (2007)

21. Kalan, A.A.E., Baina, A., Beitollahi, H., Bessani, A., Bondavalli, A., Correia, M., Daidone, A., Deconinck, G., Deswarte, Y., Garrone, F., Grandoni, F., Moniz, H., Neves, N., Rigole, T., Sousa, P., Verissimo, P.: D10: Preliminary Specification of Services and Protocols. Project deliverable, CRUTIAL: Critical Utility Infrastructural Resilience (2008)
22. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Monographs in Theoretical Computer Science. Springer, Heidelberg (1997)
23. Chiola, G., Marsan, M.A., Balbo, G., Conte, G.: Generalized Stochastic Petri Nets: A Definition at the Net Level and its Implications. *IEEE Trans. Soft. Eng.* 19(2), 89–107 (1993)
24. ArgoSPE: <http://argospe.tigris.org>
25. Object Management Group: UML Profile for Schedulability, Performance and Time Specification (January 2005), V1.1, f/05-01-02
26. Huber, P., Jensen, K., Shapiro, R.M.: Hierarchies in Coloured Petri Nets. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 313–341. Springer, Heidelberg (1991)
27. University of Torino: The GreatSPN tool (2002), <http://www.di.unitorino.it/~greatspn>
28. Heiner, M., Heisel, M.: Modeling Safety-Critical Systems with Z and Petri Nets. In: Felici, M., Kanoun, K., Pasquini, A. (eds.) SAFECOMP 1999. LNCS, vol. 1698, pp. 361–374. Springer, Heidelberg (1999)
29. Ghezzi, C., Mandrioli, D., Morasca, S., Pezzè, M.: A Unified High-Level Petri Net Formalism for Time-Critical Systems. *IEEE Trans. Softw. Eng.* 17(2), 160–172 (1991)
30. Houmb, S.H., Sallhammar, K.: Modelling System Integrity of a Security Critical System Using Colored Petri Nets. In: Proceedings of Safety and Security Engineering (SAFE 2005), Rome, Italy, pp. 3–12. WIT Press (2005)
31. Harrison, N.B., Avgeriou, P.: Incorporating Fault Tolerance Tactics in Software Architecture Patterns. In: Proc. of the 2008 RISE/EFTS Joint Int. Workshop on Software Engineering for Resilient Systems (SERENE), pp. 9–18. ACM, New York (2008)
32. Nguyen-Tuong, A., Grimshaw, A.S.: Using Reflection for Incorporating Fault-Tolerance Techniques into Distributed Applications. Technical report, University of Virginia, Charlottesville, VA, USA (1998)
33. Rugina, A.E., Kanoun, K., Kaâniche, M.: A System Dependability Modeling Framework Using AADL and GSPNs. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems IV. LNCS, vol. 4615, pp. 14–38. Springer, Heidelberg (2007)
34. Majzik, I., Pataricza, A., Bondavalli, A.: Stochastic Dependability Analysis of System Architecture Based on UML Models. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems. LNCS, vol. 2677, pp. 219–244. Springer, Heidelberg (2003)

Component Behavior Synthesis for Critical Systems^{*,**}

Tobias Eckardt and Stefan Henkler

Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn,
Warburger Str. 100, Paderborn, Germany
{tobie,shenkler}@uni-paderborn.de

Abstract. Component-based architectures are widely used in embedded systems. For managing complexity and improving quality separation of concerns is one of the most important principles. For one component, separation of concerns is realized by defining the overall component functionality by separated protocol behaviors. One of the main challenges of applying separation of concerns is the later automatic composition of the separated, maybe interdependent concerns which is not supported by current component-based approaches. Moreover, the complexity of real-time distributed embedded systems requires to consider safety requirements for the composition of the separated concerns. We present an approach which addresses these problems by a well-defined automatic composition of protocol behaviors with respect to interdependent concerns. The composition is performed by taking a proper refinement relation into account so that the analysis results of the separated concerns are preserved which is essential for safety critical systems.

1 Introduction

Component-based architectures are widely used in the domain of embedded real-time systems. The main benefits of using components are their support for information hiding and reuse. The interface of a component is well defined by structural elements and a collaboration of protocols (cf. [1]). The overall component behavior is defined by the (parallelly executed) protocol behaviors. Dependencies between components are reduced to the knowledge of interfaces or ports. Thus, a component can be exchanged if the specified port remains fulfilled.

* This work was developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

** This work was developed in the project “ENTIME: Entwurfstechnik Intelligente Mechatronik” (Design Methods for Intelligent Mechatronic Systems). The project ENTIME is funded by the state of North Rhine-Westphalia (NRW), Germany and the EUROPEAN UNION, European Regional Development Fund, “Investing in your future”.

The port and interface definitions of architectural components therefore facilitate the construction of complex functionality by the composition of components and protocols.

For managing complexity and improving quality of systems, separation of concerns [2] is one of the most important principles. It enables primary software engineering goals like adaptability, maintainability, extendability and reusability. Accordingly, advanced applications of separation of concerns have gained popularity like aspect-oriented programming (AOP) [3], for example. For one component, separation of concerns is realized by defining the overall component functionality by separated protocol behaviors [4].

One of the main challenges of applying separation of concerns is the later (application specific) composition of the separated, maybe interdependent concerns [5]. In general, we can distinguish between structural, data, and behavioral composition. In the area of structural composition, approaches exist for example, that consider the software architecture as well as architectural patterns [6,7]. For data composition approaches like [8] support the generation of suitable translators. In [9,4] approaches for the behavioral composition are presented. The overwhelming complexity of embedded real-time systems, however, requires to also consider safety and bounded liveness requirements for the composition which is not included in these approaches. On the other hand, component-based approaches for embedded real-time systems (e. g. [10,11]) suffer the support for interdependent concerns for the well-defined composition.

In this paper, we present an approach which addresses these problems by a well-defined automatic composition of protocol behaviors with respect to interdependent concerns specified as *composition rules*. The defined composition rules preserve timed safety properties which is inherently important for safety critical systems. The composition is performed by taking a proper refinement relation into account, which we call *role conformance*. This way also untimed liveness properties are preserved which is equally essential for safety critical systems. This work extends the fundamental work of [4] to the domain of critical systems (cf. Section 7).

In contrast to approaches which integrate interdependent behavior by an additional observer automaton (e.g., our former work as presented in [12]), our approach enables the explicit specification of interdependent concerns and the synthesis algorithm integrates the specified concerns automatically.

In general, the observer based approach is difficult to apply and error prone. Owned by the implicit specification of composition rules by the observer automata, the developer did not know if the composition rule in mind is really correctly implemented by the observer automata. To forbid for example that two protocol behaviors are at the same time in a specific state, all “relevant” events (timing and messages) have to be observed which lead to the forbidden states. After a corresponding observer automaton has been specified the developer did not know if all relevant events are observed, if too much behavior is observed (forbidden) or if timed safety properties and untimed liveness properties of the protocol behaviors are violated.

Additionally, the developer of the observer automaton has to instrument the protocol behaviors to enable the observation. This is not intended, however, as this may cause malfunctions originating from mistakes of the developer. Altogether the observer based approach is not well suited for safety critical systems.

For our synthesis approach, we extend our modeling approach MECHATRONIC UML which addresses the development of complex embedded real-time systems. MECHATRONIC UML supports the compositional specification and verification of real-time coordination by applying component-based development and pattern-based specification [12]. Furthermore, it also supports the integrated description and modular verification of discrete behavior and continuous control of components [13].

We evaluate our approach by the RailCab project of the University of Paderborn¹. The vision of the RailCab project is a mechatronic rail system where autonomous vehicles, called RailCabs, apply a linear drive technology, as used by the Transrapid system², for example. In contrast to the Transrapid, RailCabs travel on the existing track system of a standard railway system and only require passive switches. One particular problem (previously presented in [12]) is the convoy coordination of RailCabs. RailCabs drive in convoys in order to reduce energy consumption caused by air resistance and to achieve a higher system throughput. Convoys are established on-demand and require small distances between RailCabs. These small distances cause the real-time coordination between the speed control units of the RailCabs to be safety critical which results in a number of constraints that have to be addressed when developing the RailCabs control software.

In the following section, we present the relevant parts of MECHATRONIC UML and give an overview of our synthesis approach. For the formalization of the approach, we give fundamental definitions for the input behavioral specifications in Section 3. In Section 4, we present the concept of composition rules which formalize interdependent concerns. These composition rules are applied within the automatic composition of protocol behavior, as defined by the synthesis algorithm in Section 5. As the effect of the application of a set of composition rules cannot be anticipated, the result of the synthesis can violate properties of the protocol behavior. Therefore, we present the check for role conformance in Section 6. Related work is discussed in Section 7 and at last we conclude with a summary and future work in Section 8.

2 Approach

In MECHATRONIC UML separation of concerns is realized by applying *component-based development* and in accordance with that by rigorously separating inter-component from intra-component behavior. Following this concept, the system is decomposed into participating components and *real-time coordination patterns* [12], which define how components interact with each other.

¹ <http://www-nbp.uni-paderborn.de/index.php?id=2&L=1>

² <http://www.transrapid.de/cgi-tdb/en/basics.prg>

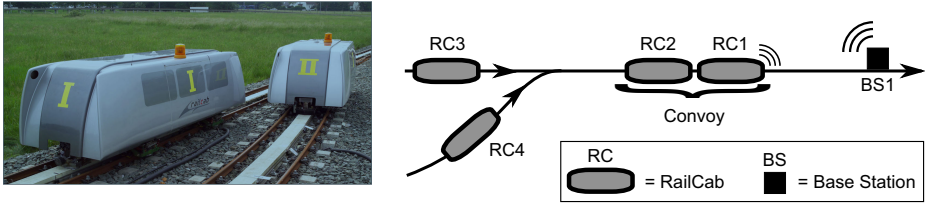


Fig. 1. RailCab example

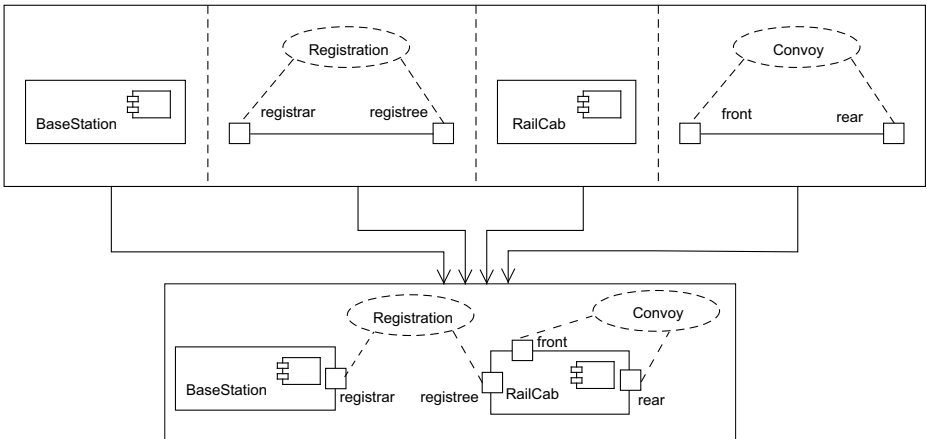


Fig. 2. Combining Separate Specifications in MECHATRONIC UML

To exemplify this, we use our RailCab case study. In Figure 1, we depict a situation of RailCabs driving in a convoy. The figure on the left shows this situation in the real test bed and on the right an abstraction is shown. In addition to the RailCabs, we depict a base station which is responsible for the power supply of the RailCabs and the management of track information for a specified section. The track information includes the data of all RailCabs in this section. RailCabs use this information to be aware of other RailCabs in their section in order to avoid crashes and possibly build convoys.

We specify two components **BaseStation** and **RailCab** (Figure 2) and two coordination patterns **Registration** and **Convoy**, which define the before described communication behavior between RailCabs and base stations.

In real-time coordination patterns, *roles* are used to abstract from the actual components participating in one coordination pattern. This way, it is possible to specify and verify coordination patterns independently from other coordination patterns and component definitions and therefore to reduce complexity. In Figure 2 the participating roles of the **Registration** pattern are **registrar** and **registree**; the roles of the **Convoy** pattern are **front** and **rear**. Each role behavior is

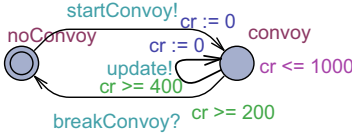


Fig. 3. Simplified Rear Role Timed Automaton

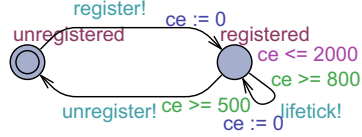


Fig. 4. Simplified Registree Role Timed Automaton

specified by one timed automaton [3, 15, 16]. The automata of the rear role and the registree role are depicted in Figure 3 and Figure 4. The automata for the front role and the registrar role only form corresponding counterparts and are therefore not depicted. We present only a simplified version of the behavior in order to present the complete approach by an example.

Initially, the rear role is in state `noConvoy` and sends a `startConvoy` event. The clock `cr` is set to zero before entering the `convoy` state. In the interval of 200 to 1000 time units the `breakConvoy` event has to be received as the location invariant of state `convoy` is $cr \leq 1000$ and the time guard of the transition is $cr \geq 200$ or in the interval of 400 to 1000 time units, periodically an `update` event is sent. The `registree` role is initially in the `unregistered` state, sends a `register` event and resets the clock. In the `registered` state in the interval of 800 to 2000 time units, periodically the `lifetick` event is sent or in the interval of 500 to 2000 time units the `unregistered` event is sent. The decision of sending the `lifetick` or `unregistered` event is at this point of nondeterministic choice.

To obtain an overall system specification later in the development process, the separated components and coordination patterns have to be combined again (Figure 2). The problem which inherently arises at this point is that separate parts of the system were specified as independent from each other when they are in fact not. This means that during the process of combining the separate parts of the system, additional dependencies between the particular specifications have to be integrated. At the same time, the externally visible behavior of the particular behavioral specifications may not be changed in order to preserve verification results [12].

In the overall system view of the RailCab example (Figure 2), the RailCab component takes part in both, the Registration and the Convoy pattern. While those patterns have been specified independently from each other, a system requirement states:

In convoy operation mode, each participating RailCab has to be registered to a base station.

Accordingly, a dependency between both patterns exists, when applied by the RailCab component. As a result, the behavior of the registree role and the

³ In MECHATRONIC UML, realtime statecharts [14] are used to describe role behavior. Realtime statecharts, however, are based on timed automata. Therefore, we define the complete synthesis procedure on the basis of timed automata in order to make the approach as general as possible.

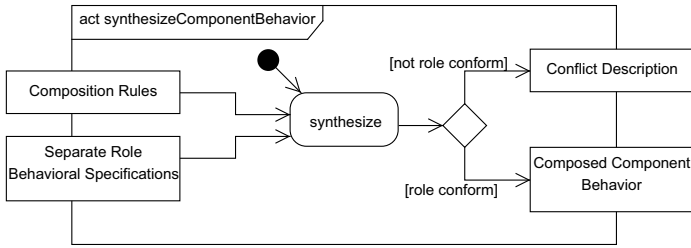


Fig. 5. Activity Diagram Illustrating the Basic Synthesis Approach

behavior of the rear role have to be refined and synchronized with each other when applied by the *RailCab* component in order to fulfill the system requirements. Still, it has to be regarded that the externally visible behavior of the *RailCab* component does not change. If this process of refinement and synchronization is performed manually, it is a time consuming and error-prone process. Consequently, this implies the necessity for automation in order to guarantee the required quality of safety critical systems.

In the proposed approach, we formalize the specification of inter-role dependencies and further separate this specification from the specification of pattern role behaviors in order to perform an automatic synthesis for the overall component behavior. Once the synthesis is performed, it is checked if the synthesized component behavior refines each of the particular pattern role behaviors properly.

The approach requires (1) the definition of a suitable refinement relation for (real) dense time systems and (2) the employment of a suitable and efficient abstraction of the timed behavioral models which is needed to perform the refinement check. The result is a fully automatic synthesis algorithm where dependencies between separate behavioral specifications are specified explicitly by so-called *composition rules* (cf. [5]). Accordingly, the input for the algorithm are composition rules and separate behavioral specifications (Figure 5) in the form of timed automata. If the synthesis is possible without violating the externally visible behavior of any of the input specifications, the output is one parallelly composed component behavior which combines all of the input behavioral specifications as well as the composition rules. If the synthesis is not possible, the algorithm returns a conflict description indicating the reason for the impossibility.

We continue with the basic definitions for the input behavioral specifications in the form of timed automata.

3 Prerequisites

For the verification of real-time coordination patterns, MECHATRONIC UML employs the model checker UPPAAL⁴. UPPAAL uses timed safety automata [16] as the input model [17]. Consequently, we also employ the concept of timed

⁴ <http://www.uppaal.com/>

safety automata for the entire approach and refer to them as timed automata in the following.

Within a timed automaton, we use clock constraints to make the behavior of the automaton dependent on the values of certain clocks of the automaton. A *general clock constraint* is a Boolean formula joining a set of equations and inequations describing the lower and upper bounds for clocks and clock differences.

Definition 1 (General Clock Constraint). *For a set C of clocks, the set $\Phi(C)$ of general clock constraints is inductively defined by the grammar $\varphi ::= x \sim n \mid x - y \sim n \mid \varphi \wedge \varphi \mid \text{true} \mid \text{false}$, where $x, y \in C$, $\sim \in \{\leq, <, =, >, \geq\}$, $n \in \mathbb{N}$.*

We further define *downwards closed clock constraints* as those constraints, which only define upper bounds for clock values. The lower bound of all clocks in a downwards closed clock constraint, consequently, is always zero.

Definition 2 (Downwards Closed Clock Constraint). *For a set C of clocks, the set $\Phi_{dc}(C) \subset \Phi(C)$ of downwards closed clock constraints is inductively defined by the grammar $\varphi ::= x \sim n \mid x - y \sim n \mid \varphi \wedge \varphi \mid \text{true}$, where $x, y \in C$, $\sim \in \{\leq, <\}$, $n \in \mathbb{N}$.*

With the definitions of clock constraints we can proceed with the definition of the syntax of a timed automaton. Note that this definition corresponds to the one given in [18], which is employed in UPPAAL.

Definition 3 (Timed Automaton). *A Timed Automaton A is a tuple $(L, l^0, \Sigma, C, I, T)$ where L is the set of locations, $l^0 \in L$ is the initial location, Σ is the finite set of events where the symbol τ is used for internal events (silent transitions), $I : L \rightarrow \Phi_{dc}(C)$ assigns each location a location invariant as a downwards closed clock constraint, C is the finite set of clocks, and $T \subseteq L \times \Sigma \times \Phi(C) \times 2^C \times L$ is the finite set of transitions $t = (l, e, g, r, l')$ with $l \in L$ the source location, $e \in \Sigma$ the related event, $g \in \Phi(C)$ the time guard as a general clock constraint, $r \subseteq C$ a set of clocks to be reset, and $l' \in L$ the target location.*

Examples of timed automata are depicted in Figure 3 and Figure 4 describing the communication behavior of the rear role and the registree role of the Convoy and the Registration real-time coordination pattern as described in Section 2. On the basis of the above given definitions, we formally define inter-role dependency specifications in the form of composition rules in the next section.

4 Composition Rules

With composition rules, interdependent concerns for the separate role behaviors can be specified as system properties which synchronize parts of the separated role behavioral models.

We divide composition rules into two distinct formalisms that are state composition rules and event composition automata. With *state composition rules*

we are able to synchronize the role behavior with respect to certain state combinations of the particular role automata. *Event composition automata*, on the other hand, provide the possibility to synchronize the role automata on the basis of events and event sequences. Both formalisms also include the specification of timing information for synchronization referring to the clocks of the role automata.

Generally speaking, system properties can be specified in terms of safety and liveness properties for a given behavioral specification [19,20]. *Safety properties* state that something bad will never happen during the execution of a program. *Liveness properties* state that something good will happen eventually. Transferring this to the context of automata synchronizations, these properties always concern two or more automata. Consequently, a *safety property for synchronization* states that something bad will never happen, when executing the corresponding automata in parallel, while a *liveness property for synchronization* expresses that something good will eventually happen during this parallel execution.

Transferring these properties to composition rules, we are able to specify both safety and liveness properties. Safety properties can be specified (1) by means of *state composition rules* in terms of forbidden state combinations of the parallel execution and (2) by means of *event composition automata* by adding further time constraints to time guards of selected transitions. Liveness properties in turn can be specified through state composition rules and event composition automata by adding further time constraints to location invariants of location combinations of the parallel execution.

State composition rules define forbidden state combinations, including timing information, in the parallel execution of the role automata. In order to make statements about forbidden state combinations of a component behavior, we need to define which clock values are forbidden in which automaton location. As the location invariant of an automaton location must be downwards closed (see Definition 3), the forbidden clock valuations can only be described by an *upwards closed clock constraint*. This is then used in a *location predicate* to connect the forbidden clock valuations to a certain location.

Definition 4 (Upwards Closed Clock Constraint). *For a set C of clocks, the set $\Phi_{uc}(C) \subset \Phi(C)$ of upwards closed clock constraints is inductively defined by the grammar $\varphi ::= x \sim n \mid x - y \sim n \mid \varphi \wedge \varphi \mid true$, where $x, y \in C$, $\sim \in \{\geq, >\}$, $n \in \mathbb{N}$.*

Definition 5 (Location Predicate). *For a timed automaton $A = (L, l^0, \Sigma, C, I, T)$, a location $l \in L$ and an upwards closed clock constraint $\varphi \in \Phi_{uc}(C)$ the set $\Gamma(A)$ of location predicates $\gamma = (l, \varphi)$ is defined by $\Gamma(A) = L \times \Phi_{uc}(C)$.*

With *state composition rules*, we want to restrict certain state combinations of the concerned role automata. Consequently, we define these state combinations by connecting location predicates by Boolean joins and meets in order to express which timed location combinations are not allowed in the composed component.

Definition 6 (State Composition Rule). *For two timed automata $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ and $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$ the set $R^S(A_1, A_2)$ of*

state composition rules ρ is defined by the grammar $\rho ::= \neg\rho_\gamma, \rho_\gamma ::= \rho_\gamma \wedge \rho_\gamma \mid \rho_\gamma \vee \rho_\gamma \mid \gamma$, where $\gamma \in \Gamma(A_1) \cup \Gamma(A_2)$.

An example of a state composition rule is the rule r_1 , given with:

$$r_1 = \neg((unregistered, true) \wedge (convoy, true)).$$

The state composition rule r_1 formalizes the pattern overlapping system requirement explained in Section 2. Correspondingly, it defines that a RailCab is not allowed to rest in states $(unregistered, true)$ and $(convoy, true)$ at the same time, where the clock constraint $true$ denotes that all clock values of the corresponding automata are concerned.

Event composition automata synchronize the parallelly executed role automata on the basis of events and event sequence by adding further timing constraints to the parallel execution.

For event composition automata, we also apply the syntax of timed automata themselves, as event composition automata are also used to describe possible event sequences of the component behavior. In contrast to pattern role automata, event composition automata do not add any further event occurrences, which means that they do not consume or provide any signals from the channels of the corresponding role automata. In other words, event composition automata are only monitoring event occurrences for a given set of role automata while they do not distinguish between sending or receiving events. They do, however, allow to add further timing constraints to the monitored event occurrences, also in terms of location invariants for the locations between the monitored events. This way, safety and liveness properties for the synchronization of several role automata can be specified. Formally, an event composition automaton is defined as follows.

Definition 7 (Event Composition Automaton). Let $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ and $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$ be two timed automata. An event composition automaton $A_E \in R^A(A_1, A_2)$ is again a timed automaton as a tuple $(L_E, l_E^0, \Sigma_E, C_E, I_E, T_E)$, where L_E is a finite non empty set of locations, $l_E^0 \subseteq L$ is the initial location, $\Sigma_E \subseteq \Sigma_1 \cup \Sigma_2$ is the finite set of events to be observed, $I : L \rightarrow \Phi_{dc}(C_E)$ assigns each location a downwards closed clock constraint, C_E is a finite set of clocks, with $C_E \cap (C_1 \cup C_2) = \emptyset$ $T_E \subseteq L_E \times \Sigma_E \times \Phi(C_E) \times 2^{C_E} \times L_E$ is a finite set of transitions $t = (l, e, g, r, l')$ $\in T_E$, $l \in L_E$ is the source location, $e \in \Sigma_E$ is the observed event, $g \in \Phi(C_E)$ is the time guard, $r \subseteq C_E$ is a set of clocks to be reset, and $l' \in L_E$ is the target location.

Semantically, an event composition automaton only observes event occurrences of the given role automata. Consequently, only those events can be used in an event composition automaton, as others can never be observed. Additionally, the set of clocks of the event composition automaton is restricted to be disjoint to the set of clocks of the role automata. This way, it is guaranteed that the event composition automaton cannot widen the time intervals of event sequences of the automata to be synchronized. This in turn guarantees that all verified deadlines of the role automata can still be met and, therefore, that all verified safety properties of the role automata are preserved (see section 6).

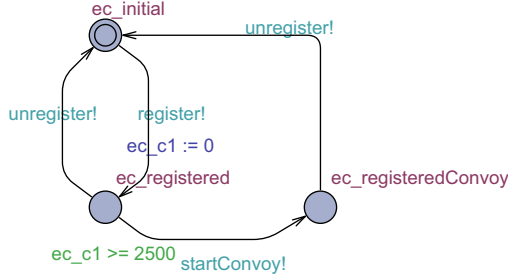


Fig. 6. Event Composition Automaton eca_1

To give an example for the pattern role automata of the rear role and the registree role (Figure 3 and Figure 4), assume a further pattern overlapping system requirement stating that a RailCab has to be registered to a base station for at least 2500 time units before starting a convoy. Observe that this requirement cannot be implemented using a state composition rule, as it is based on the occurrence of the `startConvoy!` event of the rear role automaton. Accordingly, we specify the event composition automaton eca_1 (Figure 6) to implement this requirement.

For implementing the requirement, the event composition automaton eca_1 monitors the `register!` event of the registree role automaton. Along with the occurrence of this event the clock `ec_c1` is reset. The time interval, in which the first following `startConvoy!` event may occur is then restricted by the time guard `ec_c1 >= 2500`. This means that a `startConvoy!` may not occur earlier than 2500 time units after the `register!` event which realizes that the RailCab has to be registered for at least this time to be able to start a convoy.

Once in `ec_registeredConvoy`, eca_1 changes its location only on the occurrence of the event `unregister!`, as in this situation the monitoring has to be started once more from the initial location. In all other situations, the component does not change its state of being registered and therefore this event composition rule does not have to add any further constraints.

With composition rules, we defined a suitable formalism to describe inter-role dependencies. We proceed with the definition of the synthesis algorithm in the next section, which includes the application of composition rules.

5 Synthesis Algorithm

The synthesis algorithm is divided into four distinct steps (see Figure 7). First, the parallel composition of the role automata is computed, which forms an explicit model for the parallel execution of the pattern role automata. On this parallelly composed timed automaton the composition rules are applied, by removing the forbidden system states specified by the state composition rules and by including the specified event composition automata in the parallelly composed

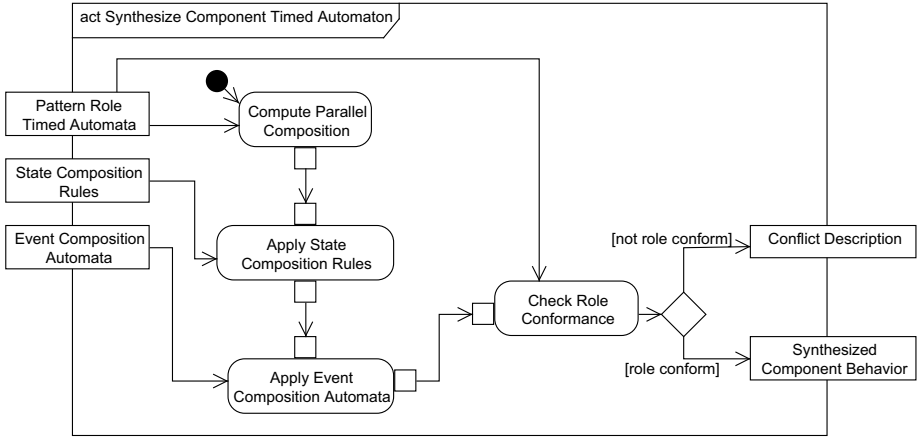


Fig. 7. Synthesis Algorithm for Timed Automata

automaton. In the last step, it is verified that the externally visible behavior of the particular role automata is preserved, as the changes made on the parallelly composed automaton by means of the application of composition rules might lead to violations of properties of the original role behaviors. Note that the overall procedure can also be applied iteratively in the development process.

The parallel composition applied in our approach is derived from the parallel composition operator of the process algebra *Calculus of Communicating Systems (CCS)* [9] as it has also been applied in *networks of timed automata* in [21][7]. In these approaches, the parallel composition allows for both synchronization and interleaving of events. The pattern role automata applied to one MECHATRONIC UML component, however, are defined such that they are independent from each other, in order to allow for compositional model checking. Consequently, we do not need to consider synchronizations in the parallel composition defined here. The parallel composition of the example automata of the rear role and the registree role (Figure 3 and Figure 4) is depicted in Figure 8.

Definition 8 (Parallel Composition). Let $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ and $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$ be two timed automata with $C_1 \cap C_2 = \emptyset$ and $\Sigma_1 \cap \Sigma_2 = \emptyset$. We define the parallel composition $A_1 \parallel A_2$ as a product automaton $A_P = (L_P, l_P^0, \Sigma_P, C_P, I_P, T_P)$, where $L_P = L_1 \times L_2$, $l_P^0 = (l_1^0, l_2^0)$, $\Sigma_P = \Sigma_1 \cup \Sigma_2$, $I_P : L_P \rightarrow \Phi(C_1) \cup \Phi(C_2)$ with $I_P((l_1, l_2)) = I_1(l_1) \wedge I_2(l_2)$, $C_P = C_1 \cup C_2$, $T_P \subseteq L_P \times \Sigma_P \times \Phi(C_P) \times 2^{C_P} \times L_P$, with $((l_1, l_2), e_1, g_1, r_1, (l_1', l_2)) \in T_P \Rightarrow ((l_1, l_2), e_1, g_1, r_1, l_1') \in T_1$, and $((l_1, l_2), e_2, g_2, r_2, (l_1, l_2')) \in T_P \Rightarrow ((l_2, l_2), e_2, g_2, r_2, l_2') \in T_2$.

The application of a state composition rule requires to evaluate each location predicate of that rule for a given parallelly composed automaton location.

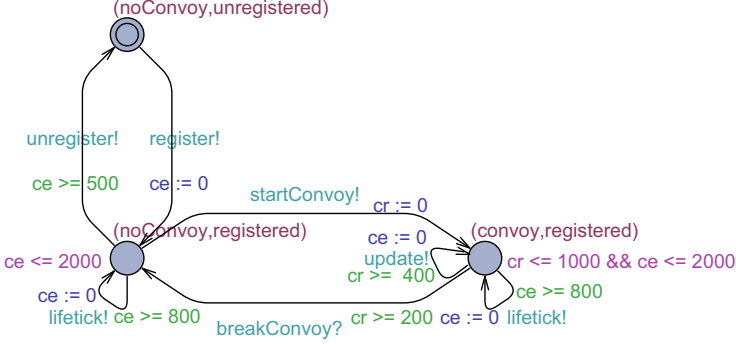


Fig. 9. Synthesized Component Behavior of the RailCab Component

but is modified such that the corresponding state composition rule has been applied to each automaton location. Automaton locations whose invariant is *false* are further removed from the automaton. The example state composition rule r_1 applied to the parallel composition of the rear and registree role automata (Figure 8) results in the automaton depicted in Figure 9.

Definition 11 (State Composition Conformance). Let $A_P = A_1 \parallel A_2 = (L_P, l_P^0, \Sigma_P, C_P, I_P, T_P)$ be the parallel composition of the timed automata A_1 and A_2 . Further let $R_1^S \subseteq R^S(A_1, A_2)$ be a set of state composition rules specified over A_1 and A_2 . The state composition conform, parallelly composed timed automaton $A_{SC} = (L_{SC}, l_{SC}^0, \Sigma_{SC}, C_{SC}, I_{SC}, T_{SC})$ is defined with $L_{SC} = L_P \setminus L_R$, where $L_R = \{l_p \mid l_p \in L_P \text{ and } \forall \rho_1, \dots, \rho_n \in R_1^S : I(l_p) \wedge \rho_1(l_p) \wedge \dots \wedge \rho_n(l_p) = \text{false}\}$, $l_{SC}^0 = l_P^0 \Leftrightarrow l_P^0 \in L_{SC}$, $\Sigma_{SC} = \Sigma_P$, $I_{SC} : L_{SC} \rightarrow \Phi(C_{SC})$ with $I_{SC}(l_p) = I_P(l_p) \wedge \rho_1(l_p) \wedge \dots \wedge \rho_n(l_p), \forall \rho_1, \dots, \rho_n \in R_1^S$, $C_{SC} = C_P$, $T_{SC} \subseteq L_{SC} \times \Sigma_{SC} \times \Phi(C_{SC}) \times 2^{C_{SC}} \times L_{SC}$, with $(l_p, e, g, r, l_p') \in T_{SC} \Leftrightarrow (l_p, e, g, r, l_p') \in T_P \wedge l_p, l_p' \in L_{SC}$.

Similar to the parallel composition used for the parallel execution of the role automata, applying event composition automata can also be compared to the parallel composition operator of the process algebra *Calculus of Communicating Systems (CCS)* [9] or the *networks of timed automata* formalism defined in [21]. Here, the resulting automaton is a composition of the event composition automaton and the parallel composition of the role automata.

The fundamental difference is that for the event composition automaton application only synchronization of events is taken into account, as event composition automata do not define any new event occurrences for the parallel execution. Furthermore, these synchronizations do not take the channel concept into account, which means that a sending event is synchronized with a sending event and also results in a sending event. This also holds for receiving events and originates from the fact that the event composition automaton only observes the event

occurrences of the parallel execution. We call this type of synchronization *silent synchronization*.

In the resulting automaton the additional time guards, clock resets and location invariants of the event composition automaton are added to the composed locations and synchronized transitions as defined in the following.

Definition 12 (Event Composition Conformance). Let $A_{SC} = (L_{SC}, l_{SC}^0, \Sigma_{SC}, C_{SC}, I_{SC}, T_{SC})$ be a state composition conform, parallelly composed timed automaton originating from the timed automata $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ and $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$ with $C_1 \cap C_2 = \emptyset$ and $\Sigma_1 \cap \Sigma_2 = \emptyset$. Furthermore, let $A_E = (L_E, l_E^0, \Sigma_E, C_E, I_E, T_E) \in R^A(A_1, A_2)$ be an event composition automaton for A_1 and A_2 . We define the event composition conform and state composition conform, parallelly composed timed automaton $A_{EC} = (L_{EC}, l_{EC}^0, \Sigma_{EC}, C_{EC}, I_{EC}, T_{EC})$ with $L_{EC} \subseteq L_1 \times L_2 \times L_E$, with $(l_1, l_2, l_e) \in L_{EC}$ iff $(l_1, l_2) \in L_{SC}$ and $I_{SC}((l_1, l_2)) \wedge I_E(l_e) \neq \text{false}$ and (l_1, l_2, l_e) is reachable through T_{EC} , $l_{EC}^0 = (l_1^0, l_2^0, l_e^0)$, iff $(l_1^0, l_2^0, l_e^0) \in L_{EC}$, $\Sigma_{EC} = \Sigma_1 \cup \Sigma_2$, $I_{EC} : L_{EC} \rightarrow \Phi(C_1) \cup \Phi(C_2) \cup \Phi(C_E)$ with $I_{EC}((l_1, l_2, l_e)) = I_{SC}((l_1, l_2)) \wedge I_E(l_e)$, $C_{EC} = C_1 \cup C_2 \cup C_E$, $T_{EC} \subseteq L_{EC} \times \Sigma_{EC} \times \Phi(C_{EC}) \times 2^{C_{EC}} \times L_{EC}$, with $((l_1, l_2, l_e), e_1, g_1, r_1, (l_1', l_2', l_e')) \in T_{EC} \Leftrightarrow ((l_1, l_2), e_1, g_1, r_1, (l_1', l_2')) \in T_{SC} \wedge \forall l_e' \in L_E : (l_e, e_1, g_e, r_e, l_e') \notin T_E$, $((l_1, l_2, l_e), e_2, g_2, r_2, (l_1, l_2', l_e')) \in T_{EC} \Leftrightarrow ((l_1, l_2), e_2, g_2, r_2, (l_1, l_2')) \in T_{SC} \wedge \forall l_e' \in L_E : (l_e, e_2, g_e, r_e, l_e') \notin T_E$, $((l_1, l_2, l_e), e_1, g_1 \wedge g_e, r_1 \cup r_e, (l_1', l_2', l_e')) \in T_{EC} \Leftrightarrow ((l_1, l_2), e_1, g_1, r_1, (l_1', l_2')) \in T_{SC} \wedge (l_e, e_1, g_e, r_e, l_e') \in T_E$, $((l_1, l_2, l_e), e_1, g_1 \wedge g_e, r_1 \cup r_e, (l_1, l_2', l_e')) \in T_{EC} \Leftrightarrow ((l_1, l_2), e_2, g_2, r_2, (l_1, l_2')) \in T_{SC} \wedge (l_e, e_2, g_e, r_e, l_e') \in T_E$.

To exemplify this, we apply the event composition automaton eca_2 to the parallel composition of the simplified rear role and registree role automaton, where the state composition rule r_1 has already been applied (Figure 9). This results in the timed automaton depicted in figure 10. Note that every location of this automaton refers to both the locations of the role automata as well as the locations of the event composition automaton eca_1 . Furthermore, observe that those composed locations which are not reachable from the initial composed location (*noConvoy, unregistered, ec.initial*) are omitted.

In the resulting automaton, the clock reset $ec_c1 := 0$ and the time guard $ec_c1 \geq 2500$ originating from the event composition automaton is added to the `register!` and to the `startConvoy!` transition respectively. Furthermore, it is now distinguished between the (`noConvoy, registered, . . .`) locations where the RailCab has just been registered (`noConvoy, registered, ec_registered`) and where the RailCab has already been in a convoy without being unregistered in-between (`noConvoy, registered, ec_registeredConvoy`)).

The resulting automaton describes the synthesized component behavior of the RailCab component. We have not yet ensured, however, that the externally visible behavior of each of the role automata is preserved. This is described in the next section.

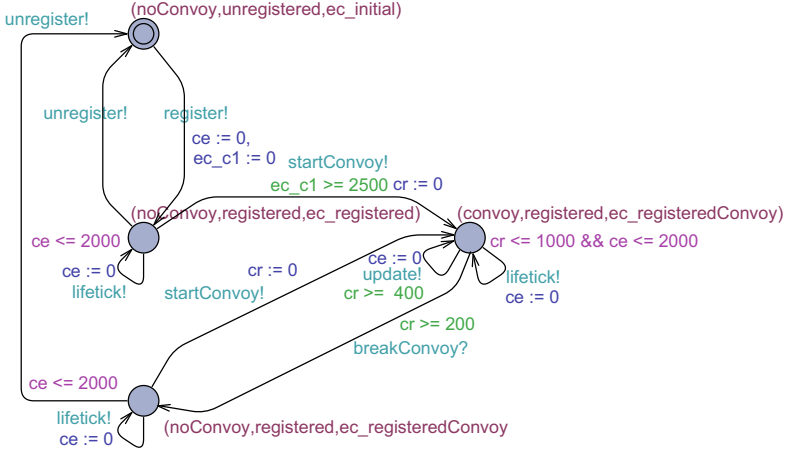


Fig. 10. Event Composition Rule eca_2 Applied to the Timed Automaton Depicted in Figure 9

6 Preserving Role Behavior

After composition rules have been applied to the parallelly composed timed automaton, it is not ensured anymore that the visible behavior of each of the particular role automata is still preserved. Assume for example, the application of an additional state composition rule $r_2 = \neg((registered, true) \wedge (convoy, cr > 100))$ to the composed timed automaton given in Figure 9. This results in a new location invariant $(cr \leq 100 \ \&\& \ ce \leq 2000)$ for the location $(convoy, registered)$. As a consequence, the outgoing $breakConvoy?$ transition can never be enabled, as its time guard $cr \geq 200$ can never evaluate to true. Accordingly, the relevant behavior of the convoy role is not anymore included in the composition conform automaton. Furthermore, note that this is not always trivial to see when specifying composition rules, as some relevant behavior is removed not before two or more rules are applied. The rule r_2 applied on the original parallel composition (see Figure 8), for example, would not remove the executability of the $breakConvoy?$ transition, as the automaton could switch to $(unregistered, convoy)$ to execute the $breakConvoy?$ transition.

In order to preserve the relevant role behavior, we need to ensure that in the refined component behavior all timed safety properties and all untimed liveness properties are preserved. This would imply that no deadlines of the original role automata are violated while still all events of the original automata are (in the correct order) visible within the original time interval. If both of these properties are preserved, we say that the refined component behavior is *role conform*. In the following, we give a sketch of a proof for role conform component behavior. A detailed proof is discussed in [22].

For *preserving timed safety properties*, we have defined the composition rule formalism exactly the way that neither any time interval can be widened nor

can additional events be added to the refined behavior. This means that composition rules can only restrict the time intervals of existing behavior or can remove certain state combinations completely. Thus, all timed safety properties are inherently preserved by the synthesis procedure.

For *preserving untimed liveness properties*, we have to ensure that each path of each single role automaton still exists in the refined (parallel composed) component automaton. This problem can be split up into analyzing the offered behavior of each refined component automaton location (cf. *protocol conformance* in [4]). This means that we verify that each refined location offers the same sending and receiving events as each of the corresponding role automaton locations. In the refined automaton of the RailCab component (Figure 9), for example, we have to verify for the (noConvoy,unregistered) location that it offers a startConvoy! event for the rear role automaton and a register! event for the registree role automaton.

The *offered behavior*, however, is defined such that it does not require the concerned location to have a direct outgoing transition with the corresponding event. Instead we also allow for transitions in-between, which are triggered by events of other role automata. This is possible because, for one particular role, the behavior of other roles is invisible. In the refined RailCab component automaton this means that the (noConvoy,unregistered) location also offers a startConvoy! event through the register! transition which originates from the registree automaton.

In addition to that we analyze timed systems. Therefore, we have to take the timing information in terms of clock values of the automaton into account. We cope with this by constructing the *zone automaton* [18,23] of the refined component automaton and verifying the offered behavior of each *zone location* instead of the automaton location. For this we also include the timing information of each original role automaton location in terms of location invariants and time guards of outgoing transitions in the analysis. The analysis is finally performed by applying operations on zones (cf. [18,23]) and comparing the offered events of each zone location with the offered events of each corresponding role automaton location in the time interval of the zone location.

If each zone location offers the required behavior, we also preserve all untimed liveness properties of the role behaviors and, thus, the refined component behavior is a correct refinement of the parallel composition of the particular role behaviors⁵. If this is not the case, one or more of the specified composition rules violate the externally visible behavior of at least one of the role behaviors. In this case the developer must either adapt the composition rules or go back to the specification of the corresponding real-time coordination patterns.

7 Related Work

Work which is related to our approach exists in the field of controller synthesis as well as in the field of component-based software development.

⁵ The correct refinement is defined by a weaker form of a (timed) bisimulation equivalence [24,25] which we call observational timed bisimulation.

The field of controller synthesis [26,27,28] deals with the problem of synthesizing a behavioral model for a *controller* which interacts with some *environment*. In a controller, interaction is specified through alternating actions between the controller and the environment. Consequently, for the behavioral model a special type of timed automaton, a *timed game automaton* [26], is applied. In a timed game automaton, transitions are partitioned into those controllable by the controller and those controllable by the environment.

There exist a number of approaches for the controller synthesis of system and component-level behavior models from system specifications which considers no time (e.g. [29,30]). Current work in this domain focuses on synthesis approaches based on modal transition systems (e.g. [31]). The motivation of these approaches is to capture the possible system or component implementations. In general, these approaches are also able to restrict the forbidden behavior by properties.

As we presented in [32], we divide the specification in two phases. First, we specify and analyze the protocol behavior independently from the concrete application of a component which results in independent pattern role automata. These behavior, which we can synthesize by our parameterized synthesis approach [33,34,32], is multiple applicable by different component implementations. In a second step, we specify the restrictions for the different component implementations and synthesize the component behavior by considering these restrictions and a refinement relation which preserves the formal verification results of the protocol behavior.

Therefore, the main difference to our synthesis approach is that the given behavioral model of controller synthesis does not take a compositional character of this model into account as this is not necessarily given in the underlying controller behavior. As the compositional character is mandatory for safety critical systems to be able to handle the complexity especially for the analysis, these approaches are rather not appropriate. In our approach this is given by the independent pattern role automata. In the controller synthesis approach, the compositionality can consequently also not be considered for the specification of the properties which have to be synthesized. Altogether, this results in a different equivalence relation between the original and the synthesized model which in turn results in different synthesis algorithms. Furthermore, none of these approaches takes all the relevant characteristics of safety critical systems into account that are time, safety and bounded liveness properties.

In [4], Giese and Vilbig present a synthesis procedure for the behavior of interacting components. While the basic idea of their approach and our approach is the same, the main goal of the synthesis differs. Giese and Vilbig propose to synthesize a maximal consistent component behavior which allows for representational non-determinism by the explicit use of τ -transitions representing internal component behavior. Our goal, on the other hand, is to synthesize a correct refined component behavior with respect to safety and liveness properties where the behavior of other ports is treated as internal component behavior. Furthermore, we employ real-time behavioral models as input specifications in order to suit the requirements of safety critical systems.

8 Conclusion and Future Work

In this paper we proposed an approach to automatically synthesize the behavior of components applied in critical systems. Therefore, we propose to specify dependencies between several role behaviors separately by means of composition rules. Additionally, we defined a procedure to automatically integrate the composition rules for a given set of role behaviors. Afterwards it is checked that the resulting component behavior refines each of the role behaviors properly. We exemplify the approach by extending the MECHATRONIC UML. A first prototype and an evaluation of our approach is presented in [32,35]. For future work we plan to perform an exhaustive evaluation of the approach in the RailCab project and industrial applications. This way, it could also be evaluated if the proposed composition rule formalism is sufficient to specify dependencies between several coordination roles in a multi-cast setting.

References

1. Bosch, J., Szyperski, C.A., Weck, W.: Component-Oriented Programming. In: Malenfant, J., Moisan, S., Moreira, A.M.D. (eds.) ECOOP 2000 Workshops. LNCS, vol. 1964, pp. 55–64. Springer, Heidelberg (2000)
2. Dijkstra, E.: A Discipline of Programming. Prentice-Hall Series in Automatic Computation (1976)
3. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
4. Giese, H., Vilbig, A.: Separation of Non-Orthogonal Concerns in Software Architecture and Design. *Software and System Modeling (SoSyM)* 5(2), 136–169 (2006)
5. Tarr, P., Ossher, H., Harrison, W., Sutton Jr., S.M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: ICSE 1999: Proceedings of the 21st International Conference on Software Engineering, pp. 107–119. ACM, New York (1999)
6. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: *Pattern-Oriented Software Architecture*, vol. 1. John Wiley & Sons, Chichester (1996)
7. Garlan, D., Perry, D.: (introduction to the) Special Issue on Software Architecture. *IEEE Transactions on Software Engineering* 21(4) (April 1995)
8. Gruber, T.R.: A Translation Approach to Portable Ontology Specifications. *Knowl. Acquis.* 5(2), 199–220 (1993)
9. Milner, R.: *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River (1989)
10. Selic, B.: Real-Time Object-Oriented Modeling (room). In: 2nd IEEE Real-Time Technology and Applications Symposium (RTAS 1996), Boston, MA, USA, June 10-12, p. 214. IEEE Computer Society, Los Alamitos (1996)
11. Jackson, E.K., Sztipanovits, J.: Using Separation of Concerns for Embedded Systems Design. In: EMSOFT 2005: Proceedings of the 5th ACM International Conference on Embedded Software, pp. 25–34. ACM, New York (2005)
12. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the Compositional Verification of Real-Time UML Designs. In: Proc. of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-11), September 2003, pp. 38–47 (2003)

13. Giese, H., Burmester, S., Schäfer, W., Oberschelp, O.: Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In: Proc. of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004), Newport Beach, USA, pp. 179–188. ACM Press, New York (2004)
14. Giese, H., Burmester, S.: Real-Time Statechart Semantics. Technical Report tr-ri-03-239, Lehrstuhl für Softwaretechnik, Universität Paderborn, Paderborn, Germany (June 2003)
15. Alur, R., Dill, D.L.: Automata for Modeling Real-time Systems. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 322–335. Springer, Heidelberg (1990)
16. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic Model Checking for Real-Time Systems. In: Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS), pp. 394–406. IEEE Computer Society Press, Los Alamitos (1992)
17. Pettersson, P.: Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice. PhD thesis, Department of Computer Systems, Uppsala University (February 1999)
18. Bengtsson, J.E., Yi, W.: Timed Automata: Semantics, Algorithms and Tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)
19. Lamport, L.: Proving the Correctness of Multiprocess Programs. IEEE Transactions on Software Engineering SE-3(2), 125–143 (1977)
20. Henzinger, T.A.: Sooner is Safer than Later. Information Processing Letters 43(3), 135–141 (1992)
21. Yi, W., Pettersson, P., Daniels, M.: Automatic Verification of Real-time Communicating Systems by Constraint-solving. In: Hogrefe, D., Leue, S. (eds.) Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Formal Techniques, Berne, Switzerland. IFIP Conference Proceedings, vol. 6, pp. 243–258. Chapman & Hall, Boca Raton (1994)
22. Eckardt, T., Henkler, S.: Synthesis of Reconfiguration Charts. Technical Report tr-ri-10-314, University of Paderborn, Paderborn, Germany (January 2010)
23. Alur, R.: Timed Automata. In: NATO-ASI 1998 Summer School on Verification of Digital and Hybrid Systems (1998)
24. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking (January 2000)
25. Tripakis, S., Yovine, S.: Analysis of Timed Systems Using Time-Abstracting Bisimulations. Formal Methods in System Design 18(1), 25–68 (2001)
26. Asarin, E., Maler, O., Pnueli, A.: Symbolic Controller Synthesis for Discrete and Timed Systems. In: Antsaklis, P.J., Kohn, W., Nerode, A., Sastry, S.S. (eds.) HS 1994. LNCS, vol. 999, pp. 1–20. Springer, Heidelberg (1995)
27. Altisen, K., Tripakis, S.: Tools for Controller Synthesis of Timed Systems. In: Pettersson, P., Yi, W. (eds.) Proceedings of the 2nd Workshop on Real-Time Tools (RT-TOOLS 2002) (August 2002)
28. Geist, S., Gromov, D., Raisch, J.: Timed Discrete Event Control of Parallel Production Lines with Continuous Outputs. Discrete Event Dynamic Systems 18(2), 241–262 (2008)
29. Harel, D., Kugler, H., Pnueli, A.: Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. In: Kreowski, H.-J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) Formal Methods in Software and Systems Modeling. LNCS, vol. 3393, pp. 309–324. Springer, Heidelberg (2005)
30. Whittle, J., Schumann, J.: Generating Statechart Designs from Scenarios. In: ICSE 2000: Proceedings of the 22nd International Conference on Software Engineering, pp. 314–323. ACM, New York (2000)

31. Uchitel, S., Brunet, G., Chechik, M.: Synthesis of Partial Behavior Models from Properties and Scenarios. *IEEE Transactions on Software Engineering* 35, 384–406 (2009)
32. Henkler, S., Greenyer, J., Hirsch, M., Schäfer, W., Alhawash, K., Eckardt, T., Heinzemann, C., Löffler, R., Seibel, A., Giese, H.: Synthesis of Timed Behavior from Scenarios in the Fujaba Real-Time Tool Suite. In: *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, Vancouver, Canada, Washington, DC, USA, May 16-24, pp. 615–618. IEEE Computer Society, Los Alamitos (2009)
33. Giese, H., Klein, F., Burmester, S.: Pattern Synthesis from Multiple Scenarios for Parameterized Real-Timed UML Models. In: Leue, S., Systä, T.J. (eds.) *Scenarios: Models, Transformations and Tools*. LNCS, vol. 3466, pp. 193–211. Springer, Heidelberg (2005)
34. Giese, H., Henkler, S., Hirsch, M., Klein, F.: Nobody's Perfect: Interactive Synthesis from Parametrized Real-Time Scenarios. In: *Proc. of the 5th ICSE 2006 Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM 2006)*, Shanghai, China, May 2006, pp. 67–74. ACM Press, New York (2006)
35. Eckardt, T., Henkler, S.: Synthesis of Component Behavior. In: Gorp, P.V. (ed.) *Proceedings of the 7th International Fujaba Days*, November 2009, pp. 1–5. Eindhoven University of Technology, The Netherlands (2009)

A Road to a Formally Verified General-Purpose Operating System

Martin Děký

Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics, Charles University
Malostranské náměstí 25, Prague 1, 118 00, Czech Republic
`martin.decky@d3s.mff.cuni.cz`

Abstract. Methods of formal description and verification represent a viable way for achieving fundamentally bug-free software. However, in reality only a small subset of the existing operating systems were ever formally verified, despite the fact that an operating system is a critical part of almost any other software system. This paper points out several key design choices which should make the formal verification of an operating system easier and presents a work-in-progress and initial experiences with formal verification of HelenOS, a state-of-the-art microkernel-based operating system, which, however, was not designed specifically with formal verification in mind, as this is mostly prohibitive due to time and budget constraints.

The contribution of this paper is the shift of focus from attempts to use a single “silver-bullet” formal verification method which would be able to verify everything to a combination of multiple formalisms and techniques to successfully cover various aspects of the operating system. A reliable and dependable operating system is the emerging property of the combination, thanks to the suitable architecture of the operating system.

1 Introduction

Operating systems (OSes for short) have a somewhat special position among all software. OSes are usually designed to run on bare hardware. This means that they do not require any special assumptions on the environment except the assumptions on the properties and behavior of hardware. In many cases it is perfectly valid to consider the hardware as *idealized hardware* (zero mathematical probability of failure, perfect compliance with the specifications, etc.). This means that it is solely the OS that defines the environment for other software.

OSes represent the lowest software layer and provide services to essentially all other software. Considering the principle of recursion, the properties of an OS form the assumptions for the upper layers of software. Thus the dependability of end-user and enterprise software systems is always limited by the dependability of the OS.

Finally, OSes are non-trivial software on their own. The way they are generally designed and programmed (spanning both the kernel and user mode,

manipulating execution contexts and concurrency, handling critical hardware-related operations) represent significant and interesting challenges for software analysis.

These are probably the most important reasons that led to several research initiatives in the recent years which target the creation of a formally verified OSES from scratch (e.g. [14]). Methods of formal description and verification provide fundamentally better guarantees of desirable properties than non-exhaustive engineering methods such as testing.

However, 98 %¹ of the market share of general-purpose OSES is taken by Windows, Mac OS X and Linux. These systems were clearly not designed with formal verification in mind from the very beginning. The situation on the embedded, real-time and special-purpose OSES market is probably different, but it is unlikely that the segmentation of the desktop and server OSES market is going to change very rapidly in the near future.

The architecture of these major desktop and server OSES is monolithic, which makes any attempts to do formal verification on them extremely challenging due to the large state space. Fortunately we can observe that aspects of several novel approaches from the OS research from the late 1980s and early 1990s (microkernel design, user space file system and device drivers, etc.) are slowly emerging in these originally purely monolithic implementations.

In this paper we show how specific design choices can markedly improve the feasibility of verification of an OS, even if the particular OS was not designed specifically with formal verification in mind. These design choices can be gradually introduced (and in fact some of them have already been introduced) to mainstream general-purpose OSES.

Our approach is not based on using a single “silver-bullet” formalism, methodology or tool, but on combining various engineering, semi-formal and formal approaches. While the lesser formal approaches give lesser guarantees, they can complement the formal approaches on their boundaries and increase the coverage of the set of all hypothetical interesting properties of the system.

We also demonstrate work-in-progress case study of a general-purpose research OS that was not created specifically with formal verification in mind from the very beginning, but that was designed according to state-of-the-art OS principles.

Structure of the Paper. In Section 2 we introduce the design choices and our case study in more detail. In Section 3 we discuss our approach of the combination of methods and tools. In Section 4 we present a preliminary evaluation of our efforts and propose the imminent next steps to take. Finally, in Section 5 we present the conclusion of the paper.

2 Operating Systems Design

Two very common schemes of OS design are *monolithic design* and *microkernel design*. Without going into much detail of any specific implementation, we can

¹ 98 % of client computers connected to the Internet as of January 2010 [13].

define the monolithic design as a preference to put numerous aspects of the core OS functionality into the kernel, while microkernel design is a preference to keep the kernel small, with just a minimal set of features.

The features which are missing from the kernel in the microkernel design are implemented in user space, usually by means of libraries, servers (e.g. processes/tasks) and/or software components.

2.1 HelenOS

HelenOS is a general-purpose research OS which is being developed at Charles University in Prague. The source code is available under the BSD open source license and can be freely downloaded from the project web site [11]. The authors of the code base are both from the academia and from the open source community (several contributors are employed as Solaris kernel developers and many are freelance IT professionals).

HelenOS uses a preemptive priority-feedback scheduler, it supports SMP hardware and it is designed to be highly portable. Currently it runs on 7 distinct hardware architectures, including the most common IA-32, x86-64 (AMD64), IA-64, SPARC v9 and PowerPC. It also runs on ARMv7 and MIPS, but currently only in simulators and not on physical hardware.

Although HelenOS is still far from being an everyday replacement for Linux or Windows due to the lack of end-user applications (whose development is extremely time-consuming, but unfortunately of no scientific value), the essential foundations such as file system support and TCP/IP networking are already in place.

HelenOS does not currently target embedded devices (although the ARMv7 port can be very easily modified to run on various embedded boards) and does not implement real-time features. Many development projects such as task snapshotting and migration, support for MMU-less platforms and performance monitoring are currently underway.

HelenOS can be briefly described as microkernel multiserver OS. However, the actual design guiding principles of the HelenOS are more elaborate:

Microkernel principle. Every functionality of the OS that does not have to be necessary implemented in the kernel should be implemented in user space. This implies that subsystems such as the file system, device drivers (except those which are essential for the basic kernel functionality), naming and trading services, networking, human interface and similar features should be implemented in user space.

Full-fledged principle. Features which need to be placed in kernel should be implemented by full-fledged algorithms and data structures. In contrast to several other microkernel OSes, where the authors have deliberately chosen the most simplistic approach (static memory allocation, naïve algorithms, simple data structures), HelenOS microkernel tries to use the most advanced and suitable means available. It contains features such as AVL and B+ trees, hashing tables, SLAB memory allocator, multiple in-kernel synchronization primitives, fine-grained locking and so on.

Multiserver principle. Subsystems in user space should be decomposed with the smallest reasonable granularity. Each unit of decomposition should be encapsulated in a separate task. The tasks represent software components with isolated address spaces. From the design point of view the kernel can be seen as a separate software component, too.

Split of mechanism and policy. The kernel should only provide low-level mechanisms, while the high-level policies which are built upon these mechanisms should be defined in user space. This also implies that the policies should be easily replaceable while keeping the low-level mechanisms intact.

Encapsulation principle. The communication between the tasks/components should be implemented only via a set of well-defined interfaces. In the user-to-user case the preferred communication mechanism is HelenOS IPC, which provides reasonable mix of abstraction and performance (RPC-like primitives combined with implicit memory sharing for large data transfers). In case of synchronous user-to-kernel communication the usual syscalls are used. HelenOS IPC is used again for asynchronous kernel-to-user communication.

Portability principle. The design and implementation should always maintain a high level of platform neutrality and portability. Platform-specific code in the kernel, core libraries and tasks implementing device drivers should be clearly separated from the generic code (either by component decomposition or at least by internal compile-time APIs).

In Section 3 we argue that several of these design principles significantly improve the feasibility of formal verification of the entire system. On the other hand, other design principles induce new interesting challenges for formal description and verification.

The run-time architecture of HelenOS is inherently dynamic. The bindings between the components are not created at compile-time, but during bootstrap and can be modified to a large degree also during normal operation mode of the system (via human interaction and external events).

The design of the ubiquitous HelenOS IPC mechanism and the associated threading model present the possibility to significantly reduce the size of the state space which needs to be explored by formal verification tools, but at the same time it is quite hard to express these constraints in many formalisms. The IPC is inherently asynchronous with constant message buffers in the kernel and dynamic buffers in user space. It uses the notions of uni-directional bindings, mandatory pairing of requests and replies, binding establishment and abolishment handshakes, memory sharing and fast message forwarding.

For easier management of the asynchronous messages and the possibility to process multiple messages from different peers without the usual kernel threading overhead, the core user space library manages the execution flow by so-called *fibrils*. A fibril is a user-space-managed thread with cooperative scheduling. A different fibril is scheduled every time the current fibril is about to be blocked while sending out IPC requests (because the kernel buffers of the addressee are full) or while waiting on an IPC reply. This allows different execution flows within the same thread to process multiple requests and replies. To safeguard proper

sequencing of IPC messages and provide synchronization, special fibril-aware synchronization primitives (mutexes, condition variables, etc.) are available.

Because of the cooperative nature of fibrils, they might cause severe performance under-utilization in SMP configurations and system-wide bottlenecks. As multicore processors are more and more common nowadays, that would be a substantial design flaw. Therefore the fibrils can be also freely (and to some degree even automatically) combined with the usual kernel threads, which provide preemptive scheduling and true parallelism on SMP machines. Needless to say, this combination is also a grand challenge for the formal reasoning.

Incidentally, the *full-fledged principle* causes that the size of the HelenOS microkernel is considerably larger compared to other “scrupulous” microkernel implementations. The average footprint of the kernel on IA-32 ranges from 569 KiB when all logging messages, asserts, symbol resolution and the debugging kernel console are compiled in, down to 198 KiB for a non-debugging production build. But we do not believe that the raw size of the microkernel is a relevant quality criterion per se, without taking the actual feature set into account.

To sum up, the choice of HelenOS as our case study is based on the fact that it was not designed in advance with formal verification in mind (some of the design principles might be beneficial, but others might be disadvantageous), but the design of HelenOS is also non-trivial and not obsolete.

2.2 The C Programming Language

A large majority of OSES is coded in the C programming language (HelenOS is no exception to this). The choice of C in the case of kernel is usually well-motivated, since the C language was designed specifically for implementing system software [10]: It is reasonably low-level in the sense that it allows to access the memory and other hardware resources with similar effectiveness as from assembler; It also requires almost no run-time support and it exports many features of the von Neumann hardware architecture to the programmer in a very straightforward, but still relatively portable way.

However, what is the biggest advantage of C in terms of run-time performance is also the biggest weakness for formal reasoning. The permissive memory access model of C, the lack of any reference safety enforcement, the weak type system and generally little semantic information in the code – all these properties do not allow to make many general assumptions about the code.

Programming languages which target controlled environments such as Java and C[‡] are generally easier for formal reasoning because they provide a well-known set of primitives and language constructs for object ownership, threading and synchronization. Many non-imperative programming languages can be even considered to be a form of “executable specification” and thus very suitable for formal reasoning. In C, almost everything is left to the programmer who is free to set the rules.

The reasons for frequent use of C in the user space of many established OSES (and HelenOS) is probably much more questionable. In the case of HelenOS, except for the core libraries and tasks (such as device drivers), C might be

easily replaced by any high-level and perhaps even non-imperative programming language. The reasons for using C in this context are mostly historical.

However, as we have stated in Section 1, the way general-purpose OSES are implemented changes only slowly and therefore any propositions which require radical modification of the existing code base before committing to the formal verification are not realistic.

3 Analysis

In this section, we analyze the properties we would like to check in a general-purpose OS. Each set of properties usually requires a specific verification method, tool or toolchain.

Our approach will be mostly bottom-up, or, in other words, from the lower levels of abstraction to the higher levels of abstraction. If the verification fails on a lower level, it usually does not make much sense to continue with the higher levels of abstraction until the issues are tackled. A structured overview of the formalisms, methods and tools can be seen on Figure 1.

Some of the proposed methods cannot be called “formal methods” in the rigorous understanding of the term. However, even methods which are based on semi-formal reasoning and non-exhaustive testing provide some limited guarantees in their specific context. A valued property of the formal methods is

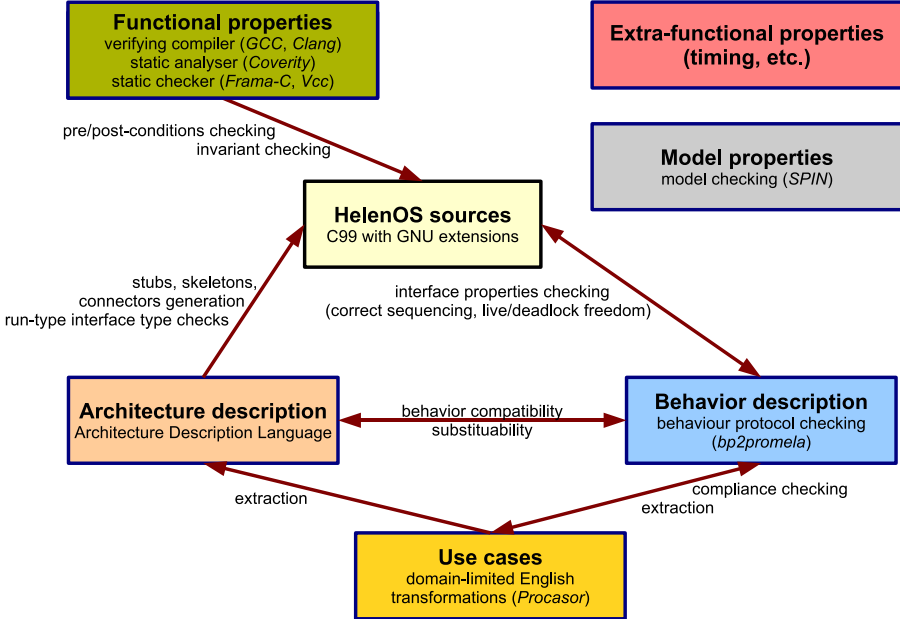


Fig. 1. Overview of the formalisms and tools proposed

to preserve these limited guarantees even on the higher levels of abstraction, thus allowing the semi-formal methods to complement the big picture where the formal methods do not provide any feasible verification so far. This increases the coverage of the set of all hypothetical interesting properties of the system (although it is probably impossible to formally define this entire set).

Please note that the titles of the following sections do not follow any particular established taxonomy. We have simply chosen the names to be intuitively descriptive.

3.1 C Language Compiler and Continuous Integration Tool

The initial levels of abstraction do not go far from the C source code and common engineering approaches. First, we would certainly like to know whether our code base is compliant with the programming language specification and passes only the basic semantic checks (proper number and types of arguments passed to functions, etc.). It is perhaps not very surprising that these decisions can be made by any plain C compiler. However, with the current implementation of HelenOS even this is not quite trivial.

Besides the requirement to support 7 hardware platforms, the system's compile-time configuration can be also affected by approximately 65 configuration options, most of which are booleans, the rest are enumerated types.

These configuration options are bound by logical propositions in conjunctive or disjunctive normal forms and while some options are freely configurable, the value of others gets inferred by the build system of HelenOS. The overall number of distinct configurations in which HelenOS can be compiled is at least one order of magnitude larger than the plain number of supported hardware platforms.

Various configuration options affect conditional compilation and linking. The programmers are used to make sure that the source code compiles and links fine with respect to the most common and obvious configurations, but the unforeseen interaction of the less common configuration options might cause linking or even compilation errors.

A straightforward solution is to generate all distinct configurations, starting from the open variables and inferring the others. This can be part of the continuous integration process which would try to compile and link the sources in all distinct configurations.

If we want to be really pedantic, we should also make sure that we run all higher level verification methods on all configurations generated by this step. That would certainly require to multiply the time required by the verification methods at least by the number of the distinct configurations. Constraining the set of configurations to just the most representative ones is perhaps a reasonable compromise to make the verification realistic.

3.2 Regression and Unit Tests

Running regression and unit tests which are part of HelenOS code base in the continuous integration process is fairly easy. The only complication lies in the

technicalities: We need to setup an automated network of physical machines and simulators which can run the appropriate compilation outputs for the specific platforms. We need to be able to reboot them remotely and distribute the boot images to them. And last but not least, we need to be able to gather the results from them.

Testing is always non-exhaustive, thus the guarantees provided by tests are strictly limited to the use cases and contexts which are being explicitly tested. However, it is arguably easier to express many common use cases in the primary programming language than in some different formalism. As we follow the bottom-up approach, filtering out the most obvious bugs by testing can save us a lot of valuable time which would be otherwise waisted by a futile verification by more formal (and more time-consuming) methods.

3.3 Instrumentation

Instrumentation tools for detecting memory leaks, performance bottlenecks and soft-deadlocks are also not usually considered to be formal verification tools (since it is hard to define exact formal properties which are being verified by the non-exhaustive nature of these tools). They are also rarely utilized on regular basis as part of the continuous integration process. But again, it might be helpful to just mention them in the big picture.

If some regression or unit tests fail, they sometimes do not give sufficient information to tell immediately what is the root cause of the issue. In that case running the faulting tests on manually or automatically instrumented executable code might provide more data and point more directly to the actual bug.

3.4 Verifying C Language Compiler

C language compilers are traditionally also not considered to be formal verification tools. Many people just say that C compilers are good at generating executable code, but do not care much about the semantics of the source code (on the other hand, formal verification tools usually do not generate any executable code at all). However, with recent development in the compiler domain, the old paradigms are shifting.

As the optimization passes and general maturity of the compilers improve over time, the compilers try to extract and use more and more semantic information from the source code. The C language is quite poor on explicit semantic information, but the verifying compilers try to rely on vendor-specific language extensions and on the fact that some semantic information can be added to the source code without changing the resulting executable code.

The checks done by the verifying compilers cannot result in fatal errors in the usual cases (they are just warnings). Firstly, the compilers still need to successfully compile a well-formed C source code compliant to some older standard (e.g. C89) even when it is not up with the current quality expectations. Old legacy source code should still pass the compilation as it did decades ago.

Secondly, the checks run by the verifying compilers are usually not based on abstract interpretation. They are mostly realized as abstract syntax tree transformations much in the line with the supporting routines of the compilation process (data and control flow graph analysis, dead code elimination, register allocation, etc.) and the evaluation function is basically the matching of antipatterns of common programming bugs.

The checks are usually conservative. The verifying compilers identify code constructs which are suspicious, which might arise out of programmer's bad intuition and so on, but even these code snippets cannot be tagged as definitive bugs (since the programmer can be simply in a position where he/she really wants to do something very strange, but nevertheless legitimate). It is upon the programmer to examine the root cause of the compiler warning, tell whether it is really a bug or just a false positive and fix the issue by either amending some additional semantic information (e.g. adding an explicit typecast or a vendor-specific language extension) or rewriting the code.

Although this level of abstraction is coarse-grained and conservative, it can be called semi-formal, since the properties which are being verified can be actually defined quite exactly and they are reasonably general. They do not deal with single traces of methods, runs and use cases like tests, but they deal with all possible contexts in which the code can run.

3.5 Static Analyzer

Static analyzers try to go deeper than verifying compilers. Besides detecting common antipatterns of bugs, they also use techniques such as abstract interpretation to check for more complex properties.

Most commercial static analyzers come with a predefined set of properties which cannot be easily changed. They are coupled with the commonly used semantics of the environment and generate domain-specific models of the software based not only on the syntax of the source code, but also based on the assumptions derived from the memory access model, allocation and deallocation rules, tracking of references and tracking of concurrency locks.

The biggest advantage of static analyzers is that they can be easily included in the development or continuous integration process as an additional automated step, very similar to the verifying compilers. No manual definition of code-specific properties is needed and false positives can be relatively easily eliminated by amending some explicit additional information to the source code within the boundaries of the programming language.

The authors of static analyzers claim large quantities of bugs detected or prevented [1], but static analyzers are still relatively limited by the kind of bugs they are designed to detect. They are usually good at pointing out common issues with security implications (specific types of buffer and stack overruns, usage of well-known functions in an unsafe way, clear cases of forgotten deallocation of resources and release of locks, etc.). Unfortunately, many static analyzers only analyze a single-threaded control flow and are thus unable to detect concurrency issues such as deadlocks.

3.6 Static Verifier

There is one key difference between a static analyzer and a static verifier: Static verifiers allow the user to specify one's own properties, in terms of preconditions, postconditions and invariants in the code. Many static verifiers also target true multithreaded usage patterns and have the capability to check proper locking order, hand-over-hand locking and even liveness.

In the context of an OS kernel and core libraries two kinds of properties are common:

Consistency constrains. These properties define the correct way how data is supposed to be manipulated by some related set of subroutines. Checking for these properties ensures that data structures and internal states will not get corrupt due to bugs in the functions and methods which are designed to manipulate them.

Interface enforcements. These properties define the correct semantics by which a set of subroutines should be used by the rest of the code. Checking for these properties ensures that some API is always used by the rest of the code in a specified way and all reported exceptions are handled by the client code.

3.7 Model Checker

On the first sight it does not seem to be reasonable to consider general model checkers as relevant independent tools for formal verification of an existing OS. While many different tools use model checkers as their backends, verifying a complete model of the entire system created by hand seems to be infeasible both in the sense of time required for the model creation and resources required by the checker to finish the exhaustive traversal of the model's state space.

Nevertheless, model checkers on their own can still serve a good job verifying abstract properties of key algorithms without dealing with the technical details of the implementation. Various properties of synchronization algorithms, data structures and communication protocols can be verified in the most generic conditions by model checkers, answering the question whether they are designed properly in theory.

If the implementation of these algorithms and protocols do not behave correctly, we can be sure that the root cause is in the non-compliance between the design and implementation and not a fundamental flaw of the design itself.

3.8 Architecture and Behavior Checker

All previously mentioned verification methods were targeting internal properties of the OS components. If we are moving to a higher-level abstraction in order to specify correct interaction of the encapsulated components in terms of interface compatibility and communication, we can utilize *Behavior Protocols* [2] or some other formalism describing correct interaction between software components.

To gain the knowledge about the architecture of the whole OS in terms of software component composition and bindings, we can use *Architecture Description Language* [12] as the specification of the architecture of the system. This language has the possibility to capture interface types (with method signatures), primitive components (in terms of provided and required interfaces), composite components (an architectural compositions of primitive components) and the bindings between the respective interfaces of the components.

It is extremely important to define the right role of the behavior and architecture description. A flawed approach would be to reverse-engineer this description from the source code (either manually or via some sophisticated tool) and then verify the compliance between the description and the implementation. However, different directions can give more interesting results:

Description as specification. Behavior and architecture description created independently on the source code serves the role of specification. This has the following primary goals of formal verification:

Horizontal compliance. Also called *compatibility*. The goal is to check whether the specifications of components that are bound together are semantically compatible. All required interfaces need to be bound to provided interfaces and the communication between the components cannot lead to *no activity* (a deadlock), *bad activity* (a livelock) or other communication and synchronization errors.

Vertical compliance. Also called *substituability*. The goal is to check whether it is possible to replace a set of primitive components that are nested inside a composite component by the composite component itself. In other words, this compliance can answer the question whether the architecture description of the system is sound with respect to the hierarchical composition of the components.

Specification and implementation compliance. Using various means of generating artificial environments for an isolated component a checker is able to partially answer the question whether the implementation of the component is an instantiation of the component specification.

Description as abstraction. Generating the behavior and architecture description from the source code by means of abstract interpretation can serve the purpose of verifying various properties of the implementation such as invariants, preconditions and postconditions. This is similar to static verification, but on the level of component interfaces.

Unfortunately, most of the behavior and architecture formalisms are static, which is not quite suitable for the dynamic nature of most OSes. This limitation can be circumvented by considering a relevant snapshot of the dynamic run-time architecture. This snapshot fixed in time is equivalent to a statically defined architecture.

The key features of software systems with respect to behavior and architecture checkers are the granularity of the individual primitive components, the level of isolation and complexity of the communication mechanism between them. Large

monolithic OSES created in semantic-poor C present a severe challenge because the isolation of the individual components is vague and the communication between them is basically unlimited (function calls, shared resources, etc.).

OSES with explicit component architecture and fine-grained components (such as microkernel multiserver systems) make the feasibility of the verification much easier, since the degrees of freedom (and thus the state space) is limited.

Horizontal and vertical compliance checking can be done exhaustively. This is a fundamental property which allows the reasoning about the dependability of the entire component-based OS. Assuming that the lower-level verification methods (described in Sections 3.1 to 3.7) prove some specific properties of the primitive components, we can be sure that the composition of the primitive components into composite components and ultimately into the whole OS does not break these properties.

The feasibility of many lower-level verification methods from Sections 3.1 to 3.7 depends largely on the size and complexity of the code under verification. If the entire OS is decomposed into primitive components with a fine granularity, it is more likely that the individual primitive components can be verified against a large number of properties. Thanks to the recursive component composition we can then be sure that these properties also hold for the entire system.

The compliance between the behavior specification and the actual behavior of the implementation is, unfortunately, the missing link in the chain. This compliance cannot be easily verified in an exhaustive manner. If there is a discrepancy between the specified and the actual behavior of the components, we cannot conclude anything about the properties holding in the entire system.

However, there is one way how to improve the situation: *code generation*. If we generate implementation from the specification, the compliance between them is axiomatic. If we are able to generate enough code from the specification to run into the hand-written “business code” where we check for the lower-level properties, the conclusions about the component composition are going to hold.

3.9 Behavior Description Generator

To conclude our path towards higher abstractions we can utilize tools that can generate the behavior descriptions from *textual use cases* written in a domain-constrained English. These generated artifacts can be then compared (e.g. via vertical compliance checking) with the formal specification. Although the comparison might not provide clean-cut results, it can still be helpful to confront the more-or-less informal user expectations on the system with the exact formal description.

3.10 Summary

So far, we have proposed a compact combination of engineering, semi-formal and formal methods which start at the level of C programming language, offer the possibility to check for the presence of various common antipatterns, check for

generic algorithm-related properties, consistency constrains, interface enforcements and conclude with a framework to make these properties hold even in the case of a large OS composed from many components of compliant behavior.

We do not claim that there are no missing pieces in the big picture or that the semi-formal verifications might provide more guarantees in this setup. However, state-of-the-art OS design guidelines can push further the boundaries of practical feasibility of the presented methods. The limited guarantees of the low-level methods hold even in the composition and the high-level formal methods do not have to deal with unlimited degrees of freedom of the primitive component implementation.

We have spoken only about the functional properties. In general, we cannot apply the same formalisms and methods on extra-functional properties (e.g. timing properties, performance properties, etc.). And although it probably does make a good sense to reason about component composition for the extra-functional properties, the exact relation might be different compared to the functional properties.

The extra-functional properties need to be tackled by our future work.

4 Evaluation

This section copies the structure of the previous Section 3 and adds HelenOS-specific evaluation of the the proposed formalisms and tools. As this is still largely a work-in-progress, in many cases just the initial observations can be made.

The choice of the specific methods, tools and formalisms in this initial phase is mostly motivated by their perceived commonality and author’s claims about fitness for the given purpose. An important part of further evaluation would certainly be to compare multiple particular approaches, tools and formalisms to find the optimal combination.

4.1 Verifying C Language Compiler and Continuous Integration Tool

The primary C compiler used by HelenOS is *GNU GCC 4.4.3* (all platforms) [3] and *Clang 2.6.0* (IA-32) [4]. We have taken some effort to support also *ICC* and *Sun Studio C* compilers, but the compatibility with these compilers is not guaranteed.

The whole code base is compiled with the `-Wall` and `-Wextra` compilation options. These options turn on most of the verification checks of the compilers. The compilers trip on common bug antipatterns such as implicit typecasting of pointer types, comparison of signed and unsigned integer values (danger of unchecked overflows), the usage of uninitialized variables, the presence of unused local variables, NULL-pointer dereferencing (determined by conservative local control flow analysis), functions with non-void return typed that do not return any value and so on. We treat all compilation warnings as fatal errors (`-Werror`), thus the code base must pass without any warnings.

We also turn on several more specific and strict checks. These checks helped to discover several latent bugs in the source code:

- `Wfloat-equal` Check for exact equality comparison between floating point values. The usage of equal comparator on floats is usually misguided due to the inherent computational errors of floats.
- `Wcast-align` Check for code which casts pointers to a type with a stricter alignment requirement. On many RISC-based platforms this can cause runtime unaligned access exceptions.
- `Wconversion` Check for code where the implicit type conversion (e.g. from float to integer, between signed and unsigned integers or between integers with different number of bits) can cause the actual value to change.

To enhance the semantic information in the source code, we use GCC-specific language extensions to annotate some particular kernel and core library routines:

`__attribute__((noreturn))` Functions marked in this way never finish from the point of view of the current sequential execution flow. The most common case are the routines which restore previously saved execution context.

`__attribute__((returns_twice))` Functions marked in this way may return multiple times from the point of view of the current sequential execution flow. This is the case of routines which save the current execution context (first the function returns as usual, but the function can eventually “return again” when the context is being restored).

The use of these extensions has pointed out to several hard-to-debug bugs on the IA-64 platform.

The automated continuous integration building system is currently work-in-progress. Thus, we do not test all possible configurations of HelenOS with each changeset yet. Currently only a representative set of 14 configurations (at least one for each supported platform) is tested by hand by the developers before committing any non-trivial changeset.

From occasional tests of other configurations by hand and the frequency of compilation, linkage and even run-time problems we conclude that the automated testing of all feasible configurations will be very beneficial.

4.2 Regression and Unit Tests

As already stated in the previous section, the continuous integration building system has not been finished yet. Therefore regression and unit tests are executed occasionally by hand, which is time consuming and prone to human omissions. An automated approach is definitively going to be very helpful.

4.3 Instrumentation

We are in the process of implementing our own code instrumentation framework which is motivated mainly by the need to support MMU-less architectures in the future. But this framework might be also very helpful in detecting memory and generic resource leaks. We have not tried *Valgrind* [17] or any similar existing tool because of the estimated complexity to adopt it for the usage in HelenOS.

4.4 Static Analyzer

The integration of various static analyzers into the HelenOS continuous integration process is underway. For the initial evaluation we have used *Stanse* [16] and *Clang Analyzer* [5]. Both of them showed to be moderately helpful to point out instances of unreachable dead code, use of language constructs which have ambiguous semantics in C and one case of possible NULL-pointer dereference.

The open framework of Clang seems to be very promising for implementing domain-specific checks (and at the same time it is also a very promising compiler framework). Our mid-term goal is to incorporate some of the features of Stanse and VCC (see Section 4.5) into Clang Analyzer.

HelenOS was also scanned by *Coverity* [7] in 2006 when no errors were detected. However, since that time the code base has not been analyzed by Coverity.

4.5 Static Verifier

We have started to extend the source code of HelenOS kernel with annotations understood by *Frama-C* [9] and *VCC* [18]. Initially we have targeted simple kernel data structures (doubly-linked circular lists) and basic locking operations. Currently we are evaluating the initial experiences and we are trying to identify the most suitable methodology, but we expect quite promising results.

As the VCC is based on the Microsoft C++ Compiler, which does not support many GCC extensions, we have been faced with the requirement to preprocess the source code to be syntactically accepted by VCC. This turned out to be feasible.

4.6 Model Checker

We are in the process of creating models of kernel wait queues (basic HelenOS kernel synchronization primitive) and futexes (basic user space thread synchronization primitive) using *Promela* and verify several formal properties (deadlock freedom, fairness) in *Spin* [15]. As both the Promela language and the Spin model checker are mature and commonly used tools for such purposes, we expect no major problems with this approach. Because both synchronization primitives are relatively complex, utilizing a model checker should provide a much more trustworthy proof of the required properties than “paper and pencil”.

The initial choice of Spin is motivated by its suitability to model threads, their interaction and verify properties related to race conditions and deadlocks (which is the common sources of OS-related bugs). Other modeling formalisms might be more suitable for different goals.

4.7 Architecture and Behavior Checker

We have created an architecture description in ADL language derived from *SOFA ADL* [12] for the majority of the HelenOS components and created the Behavior Protocol specification of these components. Both descriptions were created independently, not by reverse-engineering the existing source code. The architecture

is a snapshot of the dynamic architecture just after a successful bootstrap of HelenOS.

Both the architecture and behavior description is readily available as part of the source code repository of HelenOS, including tools which can preprocess the Behavior Protocols according to the architecture description and create an output suitable for *bp2promela* checker [2].

As the resulting complexity of the description is larger than any of the previously published case studies on Behavior Protocols (compare to [6]), our current work-in-progress is to optimize and fine-tune the *bp2promela* checker to process the input.

We have not started to generate code from the architecture description so far because of time constrains. However, we believe that this is a very promising way to go and provide reasonable guarantees about the compliance between the specification and the implementation.

4.8 Behavior Description Generator

We have not tackled the issue of behavior description generation yet, although tools such as *Procasor* [8] are readily available. We do not consider it our priority at this time.

5 Conclusion

In this paper we propose a complex combination of various verification methods and tools to achieve the verification of an entire general-purpose operating system. The proposed approach generally follows a bottom-up route, starting with low-level checks using state-of-the-art verifying C language compilers, following by static analyzers and static verifiers. In specific contexts regression and unit tests, code instrumentation and model checkers for the sake of verification of key algorithms are utilized.

Thanks to the properties of state-of-the-art microkernel multiserver operating system design (e.g. software component encapsulation and composition, fine-grained isolated components), we demonstrate that it should be feasible to successfully verify larger and more complex operating systems than in the case of monolithic designs. We use formal component architecture and behavior description for the closure. The final goal – a formally verified operating system – is the emerging property of the combination of the various methods.

The contribution of this paper is the shift of focus from attempts to use a single “silver-bullet” method for formal verification of an operating system to a combination of multiple methods supported by a suitable architecture of the operating system. The main benefit is a much larger coverage of the set of all hypothetical properties.

We also argue that the approach should be suitable for the mainstream general-purpose operating systems in the near future, because they are gradually incorporating more microkernel-based features and fine-grained software components.

Although the evaluation of the proposed approach on HelenOS is still work-in-progress, the preliminary results and estimates are promising.

Acknowledgments. The author would like to express his gratitude to all contributors of the HelenOS project. Without their vision and dedication the work on this paper would be almost impossible

This work was partially supported by the Ministry of Education of the Czech Republic (grant MSM0021620838).

References

1. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM* 53(2), 66–75 (2010)
2. Kofron, J.: Checking Software Component Behavior Using Behavior Protocols and Spin. In: *Proceedings of Applied Computing 2007*, Seoul, Korea, pp. 1513–1517 (2007)
3. GCC, the GNU Compiler Collection, <http://gcc.gnu.org/>
4. Clang: a C language family frontend for LLVM, <http://clang.llvm.org/>
5. Clang Static Analyzer, <http://clang-analyzer.llvm.org/>
6. Bulej, L., Bures, T., Coupaye, T., Decky, M., Jezek, P., Parizek, P., Plasil, F., Poch, T., Rivierre, N., Sery, O., Tuma, P.: CoCoME in Fractal. In: Rausch, A., Reussner, R., Miranda, R., Plášil, F. (eds.) *The Common Component Modeling Example*. LNCS, vol. 5153, pp. 357–387. Springer, Heidelberg (2008)
7. Coverity, <http://scan.coverity.com/>
8. Mencl, V.: Deriving Behavior Specifications from Textual Use Cases. In: *Proceedings of Workshop on Intelligent Technologies for Software Engineering (WITSE 2004)*, Linz, Austria, September 21, part of ASE 2004, pp. 331–341. Oesterreichische Computer Gesellschaft (2004)
9. Frama-C, <http://frama-c.cea.fr/>
10. Lawlis, P.K.: Guidelines for Choosing a Computer Language: Support for the Visionary Organization. *Ada Information Clearinghouse* (1998), <http://archive.adaic.com/docs/reports/lawlis/k.htm>
11. HelenOS Project, <http://www.helenos.org/>
12. Oplustil, T.: Inheritance in Architecture Description Languages. In: *Reviewed section of Proceedings of the Week of Doctoral Students 2003 conference (WDS 2003)*, Charles University, Prague, Czech Republic, vol. 2003, pp. 124–131 (2003)
13. Operating System Market Share, <http://marketshare.hitslink.com/report.aspx?qprid=8&qptimeframe=M&qpsp=132> (retrieved on 2010-02-28)
14. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA (2009)
15. Spin, <http://spinroot.com/>
16. Stanse: Static Analysis Framework for C code, <http://stanse.fi.muni.cz/>
17. Valgrind, <http://valgrind.org/>
18. VCC, <http://vcc.codeplex.com/>

Engineering a Distributed e-Voting System Architecture: Meeting Critical Requirements

J. Paul Gibson, Eric Lallet, and Jean-Luc Raffy

Département Logiciels-Réseaux (LOR),
Telecom & Management SudParis,
9 rue Charles Fourier, 91011
Évry cedex, France

Abstract. Voting is a critical component of any democratic process; and electronic voting systems should be developed following best practices for critical system development. E-voting has illustrated the importance of formal software engineering in the development of complex systems: poorly engineered and poorly documented voting systems have had serious negative consequences for all system stakeholders. It is clear that the formal verification of e-voting system models would help to address problems associated with certification against standards, and would improve the trustworthiness of the final systems. However, it is not yet clear how best to carry out such formal modelling and verification in order to leverage the compositional nature of the problem, and manage the complexity of the task.

The choice of modelling language - for expressing the high level design and architecture of an e-voting system - poses many problems due to the complex mix of requirements that such a system is required to meet. Different modelling languages are more-or-less suited to the verification of different critical requirements. Thus, we report on a mixed model approach: where we address 3 different types of critical requirements using 3 different modelling languages and development strategies. Firstly, we report on network quality-of-service issues that are analyzed through simulation models. Secondly, we report on functional correctness of a counting process that can be validated through algebraic techniques. Finally, we report on the use of formal refinement to reason about the correctness of design steps when adding detail to an architecture model. To conclude, we acknowledge the main problem that arises from such a mixed-model approach to architecture verification: how can we be sure that the different models are coherent when we integrate them in a final implementation?

1 Introduction

1.1 Overview

The work presented in this paper is part of an applied research project in which the objective is to develop a prototype for an innovative e-voting system for use

in France¹. The project is constrained by existing rules, regulations, laws and standards that the specific elections are supposed to meet, including European recommendations^[25]. A main goal is that the prototype demonstrates that such a system can be manufactured at reasonable cost, and that it meets the needs of the electorate. A secondary objective is to demonstrate the application of formal methods — as in ^[5,6] — in the engineering of the software in the e-voting system, which we consider to be critical^[24].

The software process that we followed was that of rapid-prototyping, as dictated by the limited time frame of the project. However, we applied formal modelling techniques where we felt they would add rigour to the development process without compromising the time limits. The development was intended as a learning process where we would use different formal techniques as the need arose. Thus, as well as developing a prototype e-voting system that would help us to build a final version in the future, we would also develop a better understanding of the role of the different formal methods that would help us to follow a more formal development process in future development.

The main innovation in the system is concerned with allowing the voter to choose to vote at any official voting location (and not to be restricted to a single voting station). The challenge with this innovation is to design a distributed architecture which is robust against denial of service attacks during the voting process. Currently, in France, in order to meet the requirement that no person can have more than one vote counted during an election, a person can vote at most one time. This is enforced by having a list of all people who have voted stored locally at each voting station. By allowing voters to *VoteAnywhere*, we chose not to enforce the restriction on voting only one time; as this would require either use of a network during voting in order to allow sharing of information between voting locations, or use of some complex protocol involving physical tokens that voters would “pay” in order to vote. Rather, we chose to allow re-voting and to guarantee that only a single vote for each voter is counted after the voting process is terminated. *Revoting* is not strictly necessary to permit a voter to *VoteAnywhere*, but — as we demonstrate later in this paper — it simplifies the development of the system, as well as offering some advantages to the voter.

The solution that we propose does not completely remove the need for some sort of global functionality during the voting process: we require the use of clocks that are synchronized between election locations; but demonstrate that this solution is much more robust against a denial of service attack.

1.2 Structure of Paper

In section 2 we review the specific innovative features of our chosen system and comment on the main architectural concerns. In section 3 we provide a brief summary of previous research on distributed and remote e-voting system architectures. Section 4 focuses on the key requirement that the e-voting system

¹ The system documentation is in French and we have translated the main concepts and components into English. Where multiple translations are equally reasonable, we have noted this in the text.

should be — as far as possible — robust against denial-of-service attacks. In particular, section 4 shows that support for simulation is a major advantage when choosing a modelling language with operational semantics; in our study we used Estelle[17]. In section 5, we report on the formal specification of the fundamental data (and data transformations) that are used in the counting (tabulation) process. In particular, we illustrate how the simple algebraic specification of invariant properties can aid validation[12,7] and help developers avoid types of tabulation error that are common to e-voting systems. Section 6 illustrates the use of refinement (with Event-B and the RODIN toolset[1]) for the specification and verification of a design transformation step. Section 7 reviews the development of a prototype implementation where the main difficulty was a coherent integration of multiple views as specified by our different modelling languages. We conclude the paper in section 8.

2 Revote Anywhere (By Procuration): Our Specific Requirements and Architecture Concerns

We consider the requirements for secrecy and accuracy[8] to be fundamental to all voting systems of interest to the research community. In all discussions that follow with regards to voting systems, it is implicit that no additional requirement should compromise the need for secrecy and accuracy.

We also consider *quality-of-service* to be a critical property in any voting system - the effort required to vote must not discourage electors from engaging in the voting process. In particular, the time that is required to vote must not be *unreasonable*.

In the following, we introduce 2 voting innovations (for France) — permitting voters to vote at numerous different locations and at numerous different times — and illustrate some of the problems that may arise when these innovations have to be integrated with existing features, such as allowing a third party to vote on behalf of an elector.

2.1 VoteAnywhere: A First Innovation

Restricting each elector to vote at a single specific location can have a significant negative impact on voter turnout. Providing flexibility in where electors can go to vote should improve voter turnout, and this is the major high-level objective of the *VoteAnywhere* innovation.

This paper is not proposing remote electronic voting where electors are able to vote over the internet — in the next section we review many of the problems that can arise if such unconstrained remote voting is allowed. We agree with the conclusions in a review paper — *The Development of Remote E-Voting Around the World: A Review of Roads and Directions*[21] — that: “Overall remote electronic voting has not reached the maturity to be applied in large-scale elections of major importance.” Rather, we are proposing that electors be allowed to vote at any authorised polling station. This *VoteAnywhere* requirement provides many

of the advantages of remote voting whilst not being vulnerable to most of the weaknesses [13,30]. We note that a less general variation on *VoteAnywhere* functionality is the use of *remote voting centers* [30], for “voters far from their home precincts”. This approach does not meet our objective of allowing all electors to vote at any authorised voting center (but it does illustrate that the need for remote voting is well acknowledged.)

Two other requirements are key to the development of our prototype system: *Revote* and *Procuration*. In the subsections that follow we summarise the potential interactions between each of these features [14].

2.2 VoteAnywhere with ReVote

Revote facilitates the implementation of a system that meets our *VoteAnywhere* requirement without risk of denial of service attacks. However, it also provides additional benefits to the voter when we consider elections run over a long time period. Restricting electors to vote during a narrow time frame can reduce turnout. However, widening the times when electors can vote (as with early voting in the USA) introduces the problem that electors may be discouraged from voting early because they do not have an opportunity to change their vote at a later time (while the voting process is still open.) The main objective of the *ReVote* innovation is to encourage early voting (and consequently improve turnout) by permitting an elector to revote if they wish to change a previous recorded ballot. A fundamental requirement is that only a single vote is counted for each elector. In our chosen system we refine this fundamental requirement into a rule that states that if an elector votes multiple times then only the last vote recorded by this elector will be counted.

Provided that an elector has to vote at the same polling station then there should be no problem in identifying which vote was the last recorded when a *ReVote* occurs. Some obvious options are:

1. Use a local clock to stamp each signature.
2. Use a local counter to stamp each signature.
3. Use a “destructive-write” so that a signed bulletin² added to the local urn automatically results in the destruction of any bulletin that shares the same signature already in the urn³.

However, integrating *VoteAnywhere* with *ReVote* poses problems in all three of the optional designs above:

1. Local clocks would need to be synchronised or replaced by a global clock.
- 2.&3. Require a reliable non-local network for communication of data between distributed polling booths.

In section 4 we show that option 1 is the only acceptable (and feasible) solution to meeting all our requirements.

² A bulletin is also known as a ballot.

³ An urn is also known as a ballot box.

2.3 Procuration, ReVote and VoteAnywhere: A Feature Interaction

Procuration⁴ is the feature that permits one elector (the elector-by-procuration⁵) to vote on behalf of another elector. In many elections, procuration does not necessarily prohibit an elector from voting. In France, for example, the elector may be able to go to a polling station and vote, provided that the elector-by-procuration has not already done so. Given a reliable non-local communication network then there are no undesirable interactions between *Procuration* and *VoteAnywhere* as a central voter list could guarantee that the elector and the elector-by-procuration cannot record two suffrages “at the same time” at different polling stations; in the same way that this is currently guaranteed by local voter lists at each polling station.

There is a clear undesirable interaction between *Procuration* and *ReVote* during the election process. In the first instance, an elector may be denied the right to record a suffrage whilst in the second instance an elector must never be denied the right to record a suffrage. Consequently, to provide the *ReVote* feature it may be necessary to change the existing regulations with respect to *Procuration*.

Using local clocks to implement *ReVote Anywhere* can lead to additional requirements when combined with *Procuration*. Without *procuration*, a design which uses global clocks to time-stamp ballots can safely make the assumption that a “single elector” cannot be in two places at once. As a consequence, the accuracy of the clocks is not critical and inexpensive solutions should be considered. However, with *Procuration* and *VoteAnywhere* it is possible that an elector is in two different polling stations at the same time⁶. This scenario may require much more accurate (and much more expensive) global clocks.

In our chosen system, we address these potential problems by adhering to the spirit of procuration - a vote from the original (non-procured) elector should take priority over a procured vote, irrespective of the time at which they are recorded.

2.4 Audits and Recounts

A main weakness of our proposed system is that the votes cannot be recounted by hand. The voter does have a paper record of their vote but it is impossible for them to be decrypted and hand counted — the encrypted votes must be counted as a whole before the result is decrypted.

The paper record of the vote allows a limited type of verifiability (or audit) — a voter can verify that their vote was counted after the election, but they cannot verify that it was correctly counted. The voter can also verify that the encryption process correctly records votes (during the voting process).

⁴ Procuration is also known as proxy voting or vote delegation.

⁵ The elector-by-procuration is also known as the proxy voter.

⁶ This arises if the elector goes to one polling station and the elector-by-procuration goes to another.

3 Distributed/Remote E-Voting Systems: Architecture and Design Issues

In all distributed voting systems, denial-of-service of the underlying communication architecture is a major threat. In remote voting there are also increased threats of voter coercion and/or the voting machine being untrustworthy. In the *VoteAnywhere* system, because electors vote in a controlled polling station, voter coercion should be no greater an issue than with traditional voting. Trusting e-voting machines is a major concern for all voting systems, but one which is much more serious for remote voting where the machines are not under the direct control of the voting authorities.

The design of remote electronic voting machines — requiring a network for communication between machines — is clearly a much more complex problem than the design of standalone machines. In the remainder of this section we review some of the most relevant previous research in these areas.

3.1 Denial-of-Service

In 1998, Susan King Roth identified voter disenfranchisement as a main risk of poorly designed e-voting systems [28]. Her analysis raised interesting questions with respect to poorly designed machines discouraging voter participation. This is particularly relevant when we consider the requirement that voting takes a reasonable amount of time.

In 2000, Hoffman asked *Internet Voting: Will it Spur or Corrupt Democracy?* [16], and commented on the perceived risk of denial-of-service attacks: “Imagine what a concerted denial of service attack might do to an election with Internet/Web-based voting . . .”.

In 2003 the design of an internet voting system is proposed in *REVS — A Robust Electronic Voting System* [19]. The authors write that they have designed: “a robust electronic voting system . . . that tolerates failures in communications and servers while maintaining all desired properties of a voting system.” However, the key issue of anonymity is mentioned only briefly in the conclusions, where the authors state that “REVS can benefit from a more sophisticated anonymity mechanism”. In 2004, further analysis of the REVS architecture identified weaknesses inherent in the design due to voter information being centralised [33], which introduces additional dependency on the underlying communication network.

In the same year, Chen et al. proposed the *Design of a secure anonymous Internet voting system* [9] and claim that their “scheme does not require a special voting channel and communications can occur entirely over the current Internet”. They do consider the robustness of their system with respect to election disruption (through voter behaviour): “Even if a voter intends to disrupt the election, there is no way to do it. The only way to disrupt the elections is for the voter to keep sending ballots to the TC and SC.” They then continue by explicitly forbidding re-voting: “However, the TC and SC will verify the validity of the voter-pseudonym signature and will not allow the same voter-pseudonym to

vote twice.” This approach — which ignores potential denial of service attacks on the network (independent of voter behaviour) and which forbids revoting — is very different to our proposal.

In *Verifiable Anonymous Vote Submission* [36] the *REVS* architecture is adapted to better deal with anonymity and verifiability. This work is based on two previous anonymization architectures — Mix Nets and Mix Rings — which were not originally intended for e-voting systems but which now form the central design feature of many proposals for remote electronic voting. In general, the design of such systems focuses on security aspects rather than on denial-of-service issues.

We note that relying on the internet provides opportunities for attack from foreign agents. Jefferson et al. write in *Analyzing Internet Security* [18]: “Because the internet is independent of national boundaries, an election held over the internet is vulnerable to attacks from anywhere in the world.”

In 2004, Selker and Goler report on *The SAVE system — secure architecture for voting electronically* [31]: “This voting architecture provides a means to vote over open networks in a way that is reliable, secure, and private.” Their proposal is based on demonstrating that — through n-version redundancy techniques — there is no single point of failure in their system. However, their proposed architecture is not robust against denial-of-service attacks.

Two years later, in 2006, another article — *E-voting in Estonia 2005. The first Practice of Country-wide binding Internet Voting in the World* [23] — reports on the co-ordination activities that are necessary when relying on the internet during e-voting: “System and network monitoring was performed by different parties on different levels during the e-voting period on a 24h basis. All major e-service providers (e.g. banks) and Internet operators were involved in the process with monitoring the overall “health” of Internet network traffic loads, analysis of possible Trojans/viruses etc.” They do not detail the contingency plans if their network fails during an election; but it is likely that the election would have to be aborted and re-run. Thus, one could say that their design is not dependable. The notion of “Design for dependability” appears in an article by Bryans et al. in 2006 [4], where they consider the importance of robustness and fault-tolerance. They conclude that: “. . . aborted elections are still failures.”

Qadah and Taha propose an alternative remote e-voting architecture and illustrate how mobile devices can be used as voting client machines [27]. However, they do note that their implementation — using public wireless networks — is not suitable for secure elections: “. . . for highly secure elections, such as political ones, voters need to access the e-voting system through secure channels including the use of secure client devices located at secure polling locations and connected to the e-voting system through secure Intranets/private networks”. It is interesting to note that they focus on the security of channels and networks without explicitly mentioning reliability.

3.2 Coercion and Anonymity

Coercion is a major issue in any voting system where voters are able to demonstrate how they have voted. In most traditional systems, specific procedures have

evolved in order to minimize the risk of coercion. Anonymous voting is the most widely applied technique for mitigating coercion — if all ballots are anonymous then there is no way for an elector to demonstrate (to a coercer) how they have voted. Thus, even if an elector is coerced there is no risk that the coercer can verify if the coercion has worked.

Remote e-voting would appear to increase the risk of coercion. Maaten, in *Towards Remote E-Voting: Estonian case* [22] provides evidence of coercion in remote e-voting: “During the last elections in Estonia some vote-buying incidents became public.”

The design of a secure (coercion-free) remote e-voting system is proposed in *Civitas: A Secure Remote Voting System* [10]. The paper addresses one of the major problems with remote voting: how can one ensure that voters cannot be coerced when the voting location is unsupervised? In particular they use the requirement that “voters cannot prove whether or how they voted, even if they can interact with the adversary while voting.” It should be noted that the architecture may be susceptible to denial-of-service attacks: “Civitas does not guarantee availability of either election authorities or the results of an election.

Our proposed system introduces no significant risks — over the paper system — with respect to anonymous voting. However, there is a coercion attack which could be used to force a voter to make a random vote: as a voter has a printed record of their vote against a random permutation of candidates it is possible that they would be obliged to vote randomly if an attacker forces them to record a particular sequence of preferences. This attack could not force a voter to record a particular vote because the attacker has no way of knowing how the preferences have been permuted but it does introduce an additional risk.

3.3 Other Related Issues

In 2002, Rubin analyses the *Security Considerations for Remote Electronic Voting over the Internet* [29] and concludes that: “... the technology does not yet exist to enable remote electronic voting in public elections.” We argue that, 8 years later, there have been no major technological advances that would require one to change this conclusion.

In *Swiss E-Voting Pilot Projects: Evaluation, Situation Analysis and How to Proceed* [3], the authors note that: “Parliament demanded of e-voting a similar level of security to that of postal voting.” As postal voting is the most problematic with respect to meeting requirements, it should not be a surprise that they conclude: “The required benchmark was exceeded in the pilot trials.” We argue that the benchmark for comparison must be set to a level equivalent to the best paper systems.

In *e-Voting Requirements and Implementation* [2], “the complexity of the deployment of e-voting systems and the inherent security issues that arise from the underlying distributed system” is considered. An architecture that focuses on “the security of the election servers and the channels between client machines and the servers” is proposed. Unfortunately, the authors identify a major weakness in their

architecture (and with remote voting, in general) — they cannot guarantee the security of the client machine from which a vote is cast.

4 Denial-of-Service: Our Specific Requirements

4.1 Our Specific Requirements

Elections that depend on distributed communicating (sub)systems are open to denial-of-service attacks on the underlying communication architecture. The consequences of such attacks are likely to be critical during the voting process — if electors are unable to vote for long periods of time then the election will almost certainly have to be re-run. Contrastingly, such attacks occurring before or after voting should not, if properly managed, have such serious consequences.

We propose that distributed voting systems must not depend on a reliable internet connection during the voting process in order to meet functional and non-functional requirements. In particular, no part of the voting process should depend on the sending or receiving of information on the internet (during the vote). This is the only way to guarantee that successful denial-of-service attacks cannot prevent electors from voting in a reasonable amount of time.

4.2 Simulation of Estelle Architecture Models

In previously reported research [13,14], through simulations of the formal models (written in Estelle [17]), we established that certain architectures could not provide an acceptable quality of service (to the voter) when the underlying communication network was open to denial-of-service attacks during voting. However, one particular architecture — using clocks for timestamping, but no other network communications during voting — appeared (through analysis of the simulation data) to be a possible solution to our problem.

In figure 1 we show the three alternative architectures that we modelled, in Estelle, for simulation. The first uses global lists for recording which electors (who are entitled to vote) have already voted and for the choice of candidates (vote options) offered to them. The second uses local lists to record information of/for electors who have gone to their local voting station; whilst using global lists for those who have chosen to vote elsewhere. The third has local copies of all electoral and candidate lists. The diagrams are generated from the semantics of the formal Estelle specifications. The key difference between the architectures is concerned with when the network is un use. Architecture one requires a network connection for every elector. Architecture two requires a network connection for electors who have chosen to vote at their non-default polling station. Architecture three never depends on a network connection during the voting process.

It is important to note that our simulation models — including the architecture of our chosen system — abstracted away from key aspects of e-voting systems that require the use of security protocols. As we evolved our design to incorporate these aspects, it was critical that we ensured the soundness of the abstraction: no new behaviour should introduce a need for communication across an unreliable network during the vote process.

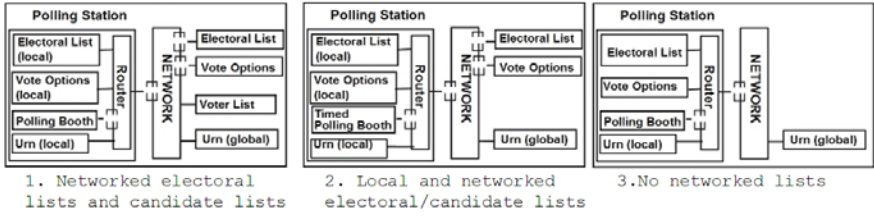


Fig. 1. Abstract Architecture Alternatives

4.3 Final Design: When Do We Need a Network?

Our design went through a number of stages. In figure 2 we show that consistency with our abstract architecture — with respect to network dependency — was maintained as further requirements for security, authentication, encryption and voter verifiability were added to the system. The key is that communication between these additional components (and the polling stations) is not necessary during the voting process. All new components are connected to local urns in each of the polling stations;

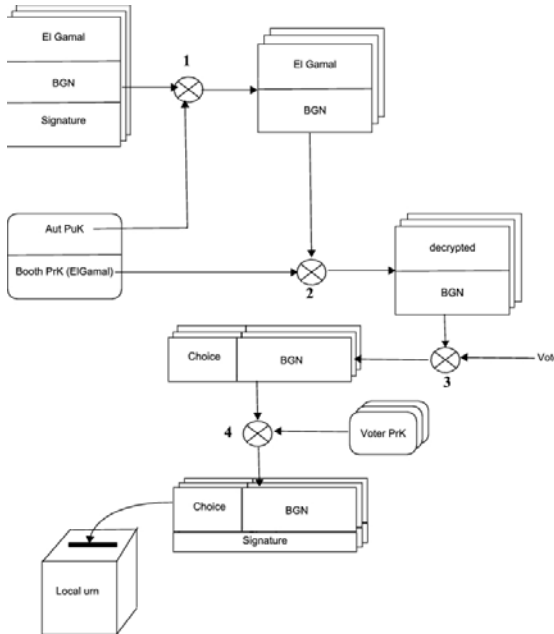


Fig. 2. Additional requirements do not require network during voting

In the top left of the figure we represent bulletin generation before the voting process starts:

1. The system uses generic bulletins which are permutations of the ordered list of candidates.
2. The bulletins are generated by mix-nets. The permutations are duplicated and each part is encrypted with a different algorithm:
 - (a) One in ElGamal with a public key for each booth (`booth PuK`).
 - (b) A second in BGN with the public key of the global urn.
3. After encryption, the bulletins are signed by a trusted authority.

During the voting process all communications between components are local:

1. The booth verifies the signature of the bulletin.
2. The booth decrypts the permutation encrypted in ElGamal.
3. The elector makes his/her choice and the decrypted permutation is destroyed.
4. The bulletin (the choice and the permutation encrypted in BGN) is then signed by the voter and put in the local urn.

Thus, the voting process does not depend on network communications.

We chose to use 2 different encryption schemes to meet 2 different requirements. BGN is required in order to provide a homomorphic mechanism for counting encrypted votes (as a whole) and decrypting the final result. This is a computationally complex algorithm and so we do not wish to use it to implement all our cryptographic functionality. Thus, we chose to use El Gamal where we optimize the computation by not requiring a homomorphic technique.

A final aspect that should be noted is that voter authentication, in our chosen system, is carried out (indirectly) after the voting process has terminated. No person is refused permission to record a vote in an urn — but during the counting process (in the global urn) all non-authentic votes are rejected in a first step. This approach can be complemented by additional checks at voting stations that permit only people entitled to vote (over-18s, for example) access the voting booths. However, our approach is robust to any failures in this initial filtering that would allow unauthorised voters access to the booth or urn. Our system would also be robust against someone authorised to vote being refused permission to vote at a particular station because this voter could try voting elsewhere.

5 Algebraic Specification

5.1 Specification and Validation of Count Rules

In previous work [12] algebraic techniques were used to model and validate the complex counting rules of the Irish parliamentary elections. A snippet of the algebraic specification of a `Vote` — a list of preferences for candidates — is given below:

```

--> *****
--> defining the Vote module
--> *****
mod! Vote {
  [Vote]
  protecting(NAT)
  protecting(ListNats)

--> some ops hidden

  op empty : Nat -> Vote
  op isempty : Vote -> Bool
  op addP : Vote Nat -> Vote
  op numCandidates : Vote -> Nat
  op invariant : Vote -> Bool
  op hasPref : Vote Nat -> Bool

--> some variables hidden
--> some equations hidden

  eq invariant(empty(numCs)) = false .
  ceq invariant(addP(v,n)) = true if (n <= numCandidates(v)) and
    (not(hasPref(v,n))) .
  ceq invariant(addP(v,n)) = false if (n <= numCandidates(v)) and
    (hasPref(v,n)) .
  ceq invariant(addP(v,n)) = false if n > numCandidates(v) .
}

```

A separate study [7] has shown the advantages of such formal models, over natural language descriptions, for the specification and validation of such algorithmic requirements. Motivated by their conclusions — and by the fact that requirements validation has been a major problem in voting systems — we chose to follow an algebraic approach to the specification of the data and data transformations in our e-voting system architecture. Further, in order to demonstrate that our approach is generally applicable, we chose not to develop a system that was appropriate only for the simplest type of counting algorithm (as with Presidential elections in France): our architecture has been designed to support the most complicated PRSTV voting schemes.

We note that these algebraic specifications in CafeObj were re-used because they provide a formal model that had already been validated to correctly represent the count algorithm. The transformation of the CafeObj models to an object oriented implementation language (like Java) follows well established methods [11]. The count implementation that results from the CafeObj specifications has an important role to play in later verification of the final prototype where the count operates on encrypted votes and involves a single decryption of the result.

This encrypted-count mechanism has not been formally verified and so we need some means of checking that it is correct. Our approach uses the

un-encrypted count (whose development was rigorously driven by the CafeObj models) as an oracle for testing the encrypted-count. In other words, we apply a form of regression testing to show that the results produced by the secure system are in agreement with the insecure system.

5.2 Verification of Data Transformations (Using Event-B Contexts)

A recurring reported problem with e-voting systems is the loss of votes arising from transport between system components; for example, from interface to urn, and from urn to count module. This problem is exacerbated by changing the way in which votes are represented as they move through the system. For example, with preferential voting, votes are typically recorded at the interface as an array of preferences.

In figure 3 we see how the way in which vote information is stored can change as votes move through the system.

In the interface, the voter conceptualises their vote as an array of candidates, some of whom are accorded preferences. However, in the ballot module the count algorithm “sees” each vote as an ordered sequence of preferences. As votes are transformed from one representation to another it is possible that a bug could transform a valid vote into an invalid vote [6]. Thus, we chose to specify such functions using Event-B contexts. In this way we formally verify (with the RODIN

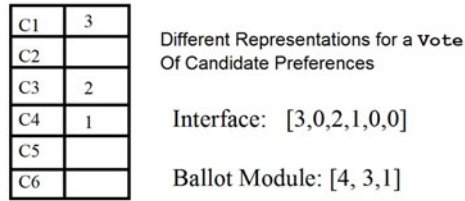


Fig. 3. Votes represented in different ways in the same system

```

CONTEXT
ISARCS10

SETS
set1

CONSTANTS
numCandidates
interface
module
move

AXIOMS
axm1 : numCandidates ∈ N1
axm2 : interface ∈ 1..numCandidates → 0.. numCandidates // a vote as stored at machine interface
axm3 : module ∈ 1.. (card(ran(interface))-1) → 1.. numCandidates // a vote as stored in the ballot module memory
axm4 : ∀i, j. i ∈ 1..numCandidates ∧ j ∈ 1..
axm5 : numCandidates ∧ {interface(i) = interface(j)} ⇒ // only 0 as repeated element in interface
      { (i = j) ∨ (interface(i) = 0) } // where 0 represents no preference for that candidate
axm6 : ∀i, j. i ≠ j ∈ interface ∧ j > 0 ⇒ j = i ∈ module // the transformation from interface to module
axm7 : ∃ m. m ∈ N1 ∧ 0..m ⊆ dom(interface) // cannot have missing elements in preference sequence
axm8 : move ∈ (1..numCandidates → 0..numCandidates) ⇒ // to prove that the move transformation is a bijection
      (1.. (card(ran(interface))-1) → 1.. numCandidates)

END
    
```

Fig. 4. Proving theorems about the data in the e-voting context

tool) that such transformations are correct. A simplified context specification, in figure 4 illustrates how the RODIN tool is used to prove theorems about the election data, as specified in an Event-B context.

The main property that we wish to prove is that the move transformation is a bijection. We note that the algebraic specification (in Event-B) of the module corresponds to that which is used in the CafeObj specification of the Vote. The advantage of re-formulating the model in Event-B is that the invariant properties can be proven when we specify the dynamic properties of the system as a machine which executes events.

6 Refinement for Formal Verification of Design Steps (Using Event-B)

As a first step towards verifying our system to be correct, we abstract away from multiple voting locations. Our goal is to prove certain properties about this simple architecture and then to refine the architecture in order to add further details/components. If we can prove this refinement to be correct then all properties that we have proven for the initial simple architecture will be guaranteed to be correct for the more detailed architecture.

Our first refinement step is to offer 2 voting locations. (This will then be refined to an arbitrary number of voting stations). A simplified part of the machine specification, in figure 5 illustrates how a new event — for adding a new voting station — can be added to the architecture in such a way that the RODIN tool verifies the correctness of the design step as a refinement. Here, the Event-B is used to model refinement between abstract machines, where behaviour is partially specified by the shared context in which the machines operate.

```

add_iso ≐
STATUS
  ordinary
ANY
  is
WHERE
  grd1 : is ∈ isoIairs
THEN
  act1 : isols = isols u {is}
END

```

Fig. 5. Modelling an architectural step as a refinement

7 The Prototype Implementation

As a proof of concept, we wished to implement this architecture as quickly as possible (following a rapid-prototyping development process). This implementation would replace the abstract network in our formal specifications with concrete communication protocols across the internet. Thus, we would be able to

demonstrate the feasibility of an implementation and validate the analysis from simulation of our formal models.

The underlying technology used to implement the distributed voting system prototypes — without any security mechanisms — was: Windows XP, Apache 2.2.9, MySQL 5.0.51, and PHP 5.2.6. On top of this, we built — using the `httpunit` tool that is popular in agile development of web sites [34] — an election simulation which simulated the behaviour of voters during the voting period (instantiating the same parameters as used in simulating our formal models).

We note that our final prototype — including the security mechanisms — is built on Java technology. The architecture of the final prototype is consistent with that used in our initial simulation. This increases confidence that the quality of service requirements will continue to be met; but we have not yet been able to execute the same simulations on the final prototype.

7.1 Simulation: Validation of Formal Requirements Model

Although not a primary goal of the prototype development, it was clear that we could build a generic implementation that could be instantiated to all of our main architectural options (including the only one which we felt was feasible). Thus, we were able to simulate `VoteAnywhere` elections — using the internet as our underlying communication network — for all the options. In figure 6 we see that — for the purposes of simulation — we instantiated only two different polling stations, each with three polling booths. This was sufficient for simulating all different scenarios of interest.

It was no surprise that the architectures that failed to meet quality-of-service requirements when we simulated the formal models also failed to meet the requirements when implemented using a real network: it is necessary but not sufficient that the abstract models meet the requirements in order for the concrete implementations to meet them. (For more details of the simulation results see [13].)

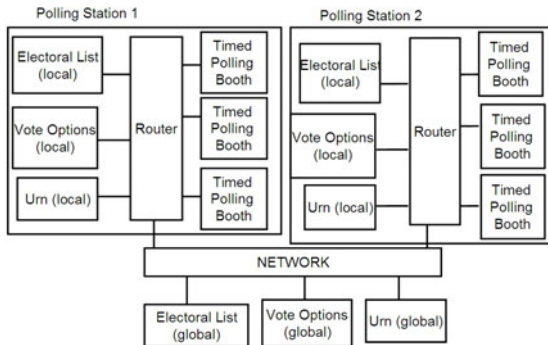


Fig. 6. Generic Architecture With 2 Polling Stations

We note that our formal models also abstracted away from communication time between system components, whether connected on a local or non-local network. We argued that such delays would be insignificant compared with the time taken for the elector to record a vote. Our election simulations validated the correctness of our abstraction — the speed of the internet connection (provided the service was available) had no effect, for all architectures tested, on the quality-of-service offered to the voter.

To conclude, our election simulation prototype demonstrated the feasibility of our architecture for meeting its requirements. The main outstanding concern was the implementation of the global clocks (see next section).

7.2 Trustworthy Global Clocks: Implementation Choices

In our simulations we made the assumption that the clocks on our different machines were synchronised; but we made no effort to guarantee that the assumption was met. We considered three implementation choices for ensuring that our clocks are synchronised in any further development of our distributed system:

1. **Network Time Protocol (NTP)** [26]: is a protocol for synchronizing the clocks of computer systems over packet-switched, variable-latency data networks. It would be the most appropriate solution to providing synchronised clocks between polling booths if we had a reliable network connection.
2. **Atomic Radio Clocks** [35]: the IEEE 1588 standard is designed for local systems requiring better accuracy than that provided by NTP. It is also designed for use where the cost of a GPS receiver in each communicating component is too high, or for where GPS signals are not reliable (or accessible)
3. **GPS Clocks** [32]: have already proven themselves in distributed real-time systems. We note that such a component could also facilitate automated verification of the location of voters (at particular polling stations). The potential impact (both positive and negative) of such information being available requires further analysis.

As none of these options is costly (with respect to the total cost of each voting booth and polling station) we propose that each booth have access to time generated by all three options (which may be controlled by a central machine in each polling station). Thus, the system would be robust against denial-of-service for any two of these three options during voting. Furthermore, the redundancy would introduce an extra level of security against some attacker attempting to manipulate the timestamp information on recorded votes (through manipulation of the local clocks).

7.3 Model Integration

The first prototype — without the security mechanisms — was developed by a software engineer with 4 years experience. The engineer was presented with all the different formal models that we had produced. The Estelle model was used

to construct the communication architecture. The algebraic specification (of a simplified count process) was used to develop the tabulation algorithm. The Event-B specifications of the design steps (refinements) played no role in the coding process — other than convincing us that the design was correct before it was implemented. The Event-B context specifications guided the implementation of Java code for specifying and verifying invariant properties. The engineer chose not to use extensions to the Java language that directly supported design by contract. Rather, they simply specified boolean invariant methods and threw runtime exceptions when such invariant properties were broken. These invariant properties helped identify coding errors (in the initial stages of implementation) but played little role in the verification of the design. Through documentation of the implementation code, we identified minor inconsistencies between the different models — these were mostly syntactic in nature.

The final prototype — with the security mechanisms — is in the process of being tested (against functional requirements). Through these tests (which were independently developed from the requirements models) we can verify that the count is correct, and that the three main features — *VoteAnywhere*, *Revote* and *Procuration* interact as required. We have no formal verification that the encryption algorithms central to the security mechanisms are correctly implemented — but the developers are experienced in using these same algorithms in a large number of security-critical systems.

It is clear from analysis of our development approach that the integration of our formal models is ad-hoc. We believe that are advantages from using different formal models at different stages of the development. However, establishing a re-usable method that coherently integrates such a mix of approaches is future research.

8 Conclusions

We have developed a prototype of an innovative voting system and addressed the major problem of denial-of-service attacks in a distributed architecture.

We have demonstrated that the development of an e-voting system can be done more rigorously through the use of formal methods, and that different modelling languages offer different advantages and disadvantages. The case study has not formally modelled all aspects and components of our voting system; a more complete model is work in progress. An important issue is a more formal integration of the different models that we have developed.

In current and future work we model our specific chosen system as a single member of a family of voting systems, where family members offer a unique subset of voting features [14]. We are also analysing the role of formality in maintaining these systems as requirements evolve (often due to changes in standards [15]).

Acknowledgements

Thanks to the anonymous reviewers for their comments and suggestions.

References

1. Abrial, J.-R., Butler, M.J., Hallerstede, S., Voisin, L.: An open extensible tool environment for event-b. In: Liu, Z., He, J. (eds.) *ICFEM 2006*. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)
2. Anane, R., Freeland, R., Theodoropoulos, G.: E-voting requirements and implementation. In: *The 9th IEEE International Conference on E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services*. CEC/EEE, Tokyo, Japan, July 2007, pp. 382–392 (2007)
3. Braun, N., Brändli, D.: Swiss e-voting pilot projects: Evaluation, situation analysis and how to proceed. In: Krimmer [20], pp. 27–36
4. Bryans, J.W., Littlewood, B., Ryan, P.Y.A., Strigini, L.: E-voting: Dependability requirements and design for dependability. In: *ARES 2006: Proceedings of the First International Conference on Availability, Reliability and Security*, Washington, DC, USA, pp. 988–995. IEEE Computer Society Press, Los Alamitos (2006)
5. Cansell, D., Gibson, J.P., Méry, D.: Formal verification of tamper-evident storage for e-voting. In: Hinchey, M., Margaria, T. (eds.) *Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007)*, London, England, UK, pp. 329–338. IEEE Computer Society Press, Los Alamitos (2007)
6. Cansell, D., Gibson, J.P., Méry, D.: Refinement: A constructive approach to formal software design for a secure e-voting interface. *Electronic Notes in Theoretical Computer Science* 183, 39–55 (2007)
7. Carew, D., Exton, C., Buckley, J., McGaley, M., Gibson, J.P.: Preliminary study to empirically investigate the comprehensibility of requirements specifications. In: Romero, P., Good, J., Acosta Chaparro, E., Bryant, S. (eds.) *Psychology of Programming Interest Group 17th annual workshop (PPIG 2005)*, pp. 182–202. University of Sussex, Brighton (2005)
8. Chaum, D., van der Graaf, J., Ryan, P.Y.A., Vora, P.: Secret ballot elections with unconditional integrity. Report CS-TR-1058, Department of Computing Science, University of Newcastle upon Tyne (2007)
9. Chen, Y.-Y., Jan, J.k., Chen, C.-L.: The design of a secure anonymous internet voting system. *Computers & Security* 23(4), 330–337 (2004)
10. Clarkson, M.E., Chong, S., Myers, A.C.: Civitas: A secure remote voting system. In: Chaum, D., Kutyłowski, M., Rivest, R.L., Ryan, P.Y.A. (eds.) *Frontiers of Electronic Voting*. Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), vol. 07311, Schloss Dagstuhl, Germany (2007)
11. Gibson, J.P.: Formal Object Oriented Development of Software Systems Using LOTOS. Thesis CSM-114, Stirling University (August 1993)
12. Gibson, J.P.: E-voting requirements modelling: An algebraic specification approach (with cafeobj). Report NUIM-CS-TR-2005-14, Department of Computer Science, National University of Ireland, Maynooth (2005)
13. Gibson, J.P., Lallet, E., Raffy, J.-L.: Analysis of a distributed e-voting system architecture against quality of service requirements. In: *The Third International Conference on Software Engineering Advances (ICSEA 2008)*, pp. 58–64. IEEE Computer Society Press, Los Alamitos (2008)
14. Gibson, J.P., Lallet, E., Raffy, J.-L.: Feature interactions in a software product line for e-voting. In: Nakamura, Reiff-Marganiec (eds.) *Feature Interactions in Software and Communication Systems X*, Lisbon, Portugal, June 2009, pp. 91–106. IOS Press, Amsterdam (2009)

15. Gibson, J.P., McGaley, M.: Verification and maintenance of e-voting systems and standards. In: Remenyi, D. (ed.) 8th European Conference on e-Government, Lausanne, Switzerland, July 2008, pp. 283–289. Academic Publishing International (2008)
16. Hoffman, L.J.: Internet voting: will it spur or corrupt democracy? In: CFP 2000: Proceedings of the tenth conference on Computers, freedom and privacy, pp. 219–223. ACM, New York (2000)
17. ISO/IEC. Estelle: A formal description technique based on an extended state transition model. Technical Report ISO 9074, Information technology - Open Systems Interconnection (1997)
18. Jefferson, D., Rubin, A.D., Simons, B., Wagner, D.: Analyzing internet voting security. *ACM Commun.* 47(10), 59–64 (2004)
19. Joaquim, R., Zuquete, A., Ferreira, P.: REVS — A Robust Electronic Voting System. In: Proceedings of the IADIS International Conference on e-Society, Lisbon, Portugal, June 2003, pp. 95–103 (2003)
20. Krimmer, R. (ed.): Electronic Voting 2006: 2nd International Workshop, Co-organized by Council of Europe, ESF TED, IFIP WG 8.6 and E-Voting.CC, Castle Hofen, Bregenz, Austria, August 2-4. LNI, vol. 86. GI (2006)
21. Krimmer, R., Triessnig, S., Volkamer, M.: The development of remote e-voting around the world: A review of roads and directions. In: Alkassar, A., Volkamer, M. (eds.) VOTE-ID 2007. LNCS, vol. 4896, pp. 1–15. Springer, Heidelberg (2007)
22. Maaten, E.: Towards remote e-voting: Estonian case. In: Prosser, A., Krimmer, R. (eds.) Electronic Voting in Europe. LNI, vol. 47, pp. 83–100. GI (2004)
23. Madise, Ü., Martens, T.: E-voting in estonia 2005. the first practice of country-wide binding internet voting in the world. In: Krimmer [20], pp. 15–26 (2005)
24. McGaley, M., Gibson, J.P.: E-voting: a safety critical system. Report NUIM-CS-TR-2003-2, Department of Computer Science, National University of Ireland, Maynooth (2003)
25. McGaley, M., Gibson, J.P.: A critical analysis of the council of europe recommendations on e-voting. In: EVT 2006: Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop 2006 on Electronic Voting Technology Workshop, pp. 9–22. USENIX Association (2006)
26. Mills, D.L.: Internet time synchronization: the network time protocol. *IEEE Transactions on Communications* 39(10), 1482–1493 (1991)
27. Qadah, G.Z., Taha, R.: Electronic voting systems: Requirements, design, and implementation. *Comput. Stand. Interfaces* 29(3), 376–386 (2007)
28. Roth, S.K.: Disenfranchised by design: voting systems and the election process. *Information Design Journal* 9(1), 1–8 (1998)
29. Rubin, A.D.: Security considerations for remote electronic voting. *ACM Commun.* 45(12), 39–44 (2002)
30. Sandler, D.R., Wallach, D.S.: The case for networked remote voting precincts. In: EVT 2008: Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop 2008 on Electronic Voting Technology Workshop, Berkeley, CA, USA, July 2008. USENIX Association (2008)
31. Selker, T., Goler, J.: The save system — secure architecture for voting electronically. *BT Technology Journal* 22(4), 89–95 (2004)
32. Sterzbach, B.: Gps-based clock synchronization in a mobile, distributed real-time system. *Real-Time Syst.* 12(1), 63–75 (1997)
33. Storer, T., Duncan, I.: Practical remote electronic elections for the uk. In: PST, pp. 41–45 (2004)

34. Tappenden, A., Beatty, P., Miller, J.: Agile security testing of web-based systems via httpunit. In: ADC 2005: Proceedings of the Agile Development Conference, Washington, DC, USA, pp. 29–38. IEEE Computer Society Press, Los Alamitos (2005)
35. Weibel, H., Béchaz, D.: IEEE1588 Implementation and Performance of Time Stamping Techniques. In: Conference on IEEE 1588, Gaithersburg (september 2004)
36. Zúquete, A., Almeida, F.: Verifiable anonymous vote submission. In: SAC 2008: Proceedings of the 2008 ACM symposium on Applied computing, pp. 2159–2166. ACM, New York (2008)

Testing Fault Robustness of Model Predictive Control Algorithms

Piotr Gawkowski¹, Konrad Grochowski¹, Maciej Ławryńczuk², Piotr Marusak²,
Janusz Sosnowski¹, and Piotr Tatjewski²

¹ Institute of Computer Science

{P.Gawkowski, J.Sosnowski}@ii.pw.edu.pl

² Institute of Control and Computation Engineering

{M.Lawrynczuk, P.Marusak, P.Tatjewski}@ia.pw.edu.pl

Warsaw University of Technology, ul. Nowowiejska 15/19, 00–665 Warsaw, Poland

Abstract. The paper deals with the problem of evaluating fault robustness of the software implemented Dynamic Matrix Control (DMC) Model Predictive Control (MPC) algorithms. Numerical and explicit implementations of the DMC algorithms are considered. It is shown that faults affecting the algorithms can provoke undesirable behaviour or even destabilize the process. Dependability was evaluated experimentally using two different software implemented fault injection approaches, the old one (FITS) and a new one (InBochs). FITS was not sufficient in case of the numerical DMC implementation. InBochs is based on the system emulator and delivers the same level of functionality as FITS while having capability to extend fault models.

Keywords: dependability evaluation, fault injection, process control, model predictive control.

1 Introduction

The testing of fault robustness of the software implementations of the Dynamic Matrix Control (DMC) Model Predictive Control (MPC) algorithms is discussed in the paper. The algorithms are designed to operate in a control system of a rectification column. It is the complex process with two inputs and two outputs, strong cross-couplings and significant time delays. High dependability and safety in particular are important features required in many control systems used in industry, automotive, medicine, civil engineering applications, etc. So, an important issue is to analyse system susceptibility to internal faults and identify unsafe situations. For this purpose various fault simulation techniques have been proposed and described in the literature, e.g. [1, 2]. Most of them were used to study calculation oriented applications. In case of real time systems with control feedback the fault effect analysis is more complex as the trace of the generated control signals and the reaction of the controlled process must be taken into account to qualify its behaviour. This is application dependent. In the literature such studies are rarely encountered. In most cases they relate to simple control algorithms (e.g. based on PID controller) and using dedicated simulation platforms, e.g. [5, 14, 19]. In the paper we present a more universal approach based on software implemented fault injectors (SWIFI).

Both numerical and explicit implementations of the DMC algorithms are researched. In case of the numerical DMC algorithm, it derives control action by solving a quadratic optimization problem at each iteration. It is an approach often used in practical applications. However, from the computational point of view, it is much more complicated than an explicit control algorithm. As a consequence, the numerical DMC control algorithm is hard to test using the conventional SWIFI approach. Thus, a new SWIFI system is proposed to address this problem.

The organization of the paper is as follows. Section 2 describes the numerical and the explicit versions of the DMC algorithm. Section 3 discusses the SWIFI systems. The new fault injection approach, capable of conducting more complex test scenarios than the old one, is also presented. Experiment set-up is described in Sect. 4. The experiments are discussed in Sect. 5. The last section concludes the paper.

2 DMC Algorithm

Model Predictive Control, thanks to performance it offers, is a widely used advanced control technique [12–18]. It has found acceptance in industry and is successfully applied practice. In the MPC algorithms the control action is calculated using a model of the process. Thus, during control signal calculation, the predicted behaviour of the control plant and constraints can be easily taken into consideration. If the model is accurate enough, MPC algorithms can offer better performance than classical control algorithms, especially for processes with difficult dynamics, e.g. with significant time delay. MPC algorithms are very useful to control Multi-Input Multi-Output (MIMO) processes with strong cross-couplings. It is so because, thanks to using the process model, the MPC algorithms contain the decoupling mechanism. Among different MPC techniques, Dynamic Matrix Control (DMC) is very popular because it uses a step-response model of the process which is easy to obtain [6, 18].

In the MPC algorithms the control signal is derived in such a way that predicted behaviour of the control system minimizes a performance function subject to the constraints of control and output signals. Thus, at each iteration of the algorithm the following optimization problem is solved:

$$\min_{\Delta u_{k+i|k}^j} \sum_{j=1}^{n_y} \sum_{i=1}^N \psi^j \cdot (\bar{y}_k^j - y_{k+i|k}^j)^2 + \sum_{j=1}^{n_u} \sum_{i=0}^{N_u-1} \lambda^j \cdot (\Delta u_{k+i|k}^j)^2, \quad (1)$$

subject to:

$$y_{\min}^j \leq y_{k+i|k}^j \leq y_{\max}^j,$$

$$u_{\min}^j \leq u_{k+i|k}^j \leq u_{\max}^j,$$

$$\Delta u_{\min}^j \leq \Delta u_{k+i|k}^j \leq \Delta u_{\max}^j,$$

where \bar{y}_k^j is a set-point value for the j^{th} output, $y_{k+i|k}^j$ is an output value for the $(k+i)^{\text{th}}$ sampling instant predicted at k^{th} sampling instant, $u_{k+i|k}^j$ are future values of the manipulated variables, $\Delta u_{k+i|k}^j$ are future changes of the manipulated variables (decision

variables of the optimization problem), $\psi^j \geq 0$ and $\lambda^j \geq 0$ are weighting coefficients for the predicted control errors of the j^{th} output and for the changes of the j^{th} manipulated variable, respectively, N and N_u denote prediction and control horizons, n_y , n_i denote the number of outputs and inputs, respectively.

The optimization problem (1) can be expressed in the matrix–vector form as:

$$J = (\bar{\mathbf{y}} - \mathbf{y})^T \cdot \boldsymbol{\Psi} \cdot (\bar{\mathbf{y}} - \mathbf{y}) + \Delta \mathbf{u}^T \cdot \mathbf{A} \cdot \Delta \mathbf{u}, \quad (1a)$$

subject to:

$$\mathbf{y}_{\min} \leq \mathbf{y} \leq \mathbf{y}_{\max},$$

$$\mathbf{u}_{\min} \leq \mathbf{u} \leq \mathbf{u}_{\max},$$

$$\Delta \mathbf{u}_{\min} \leq \Delta \mathbf{u} \leq \Delta \mathbf{u}_{\max},$$

where $\boldsymbol{\Psi}$ is a diagonal matrix of dimensionality $n_y N \times n_y N$, \mathbf{A} is a diagonal matrix of dimensionality $n_i N_u \times n_i N_u$, these matrices are composed of ψ^j and λ^j elements;

$$\Delta \mathbf{u} = [\Delta \mathbf{u}_k^1, \Delta \mathbf{u}_k^2, \dots, \Delta \mathbf{u}_k^{n_i}]^T, \quad \Delta \mathbf{u}_k^j = [\Delta u_{k|k}^j, \dots, \Delta u_{k+N_u-1|k}^j], \quad (2)$$

$$\mathbf{u} = [\mathbf{u}_k^1, \mathbf{u}_k^2, \dots, \mathbf{u}_k^{n_i}]^T, \quad \mathbf{u}_k^j = [u_{k|k}^j, \dots, u_{k+N_u-1|k}^j], \quad (3)$$

$$\mathbf{y} = [\mathbf{y}_k^1, \mathbf{y}_k^2, \dots, \mathbf{y}_k^{n_y}]^T, \quad \mathbf{y}_k^j = [y_{k+1|k}^j, \dots, y_{k+N|k}^j], \quad (4)$$

$$\bar{\mathbf{y}} = [\bar{\mathbf{y}}_k^1, \bar{\mathbf{y}}_k^2, \dots, \bar{\mathbf{y}}_k^{n_y}]^T, \quad \bar{\mathbf{y}}_k^j = \begin{bmatrix} \bar{y}_k^j, \dots, \bar{y}_k^j \\ \underbrace{\hspace{1.5cm}} \\ N \text{ elements} \end{bmatrix}, \quad (5)$$

$\Delta \mathbf{u}_{\min}$, $\Delta \mathbf{u}_{\max}$, \mathbf{u}_{\min} , \mathbf{u}_{\max} , \mathbf{y}_{\min} , \mathbf{y}_{\max} are vectors of lower and upper bounds of increments of the manipulated variables, values of the manipulated variables and of the values of output variables, respectively.

If the prediction is performed using a linear process model then the superposition principle can be used and the vector \mathbf{y} can be decomposed as:

$$\mathbf{y} = \tilde{\mathbf{y}} + \mathbf{A} \cdot \Delta \mathbf{u}, \quad (6)$$

where $\tilde{\mathbf{y}}$ is a vector of dimensionality $n_y N$ called a free response because it contains elements equal to the values of the process outputs in future in the situation if manipulated signals are frozen at the k^{th} sampling instant [13, 17, 18]:

$$\tilde{\mathbf{y}} = [\tilde{\mathbf{y}}_k^1, \tilde{\mathbf{y}}_k^2, \dots, \tilde{\mathbf{y}}_k^{n_y}]^T, \quad \tilde{\mathbf{y}}_k^j = [\tilde{y}_{k+1|k}^j, \dots, \tilde{y}_{k+N|k}^j]^T, \quad (7)$$

\mathbf{A} is a matrix of dimensionality $n_y N \times n_i N_u$ called a dynamic matrix composed of the elements of the process step response:

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n_1} \\ A_{21} & A_{22} & \cdots & A_{2n_2} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n_y,1} & A_{n_y,2} & \cdots & A_{n_y,n_i} \end{bmatrix}, A_{jm} = \begin{bmatrix} a_1^{j,m} & 0 & \cdots & 0 & 0 \\ a_2^{j,m} & a_1^{j,m} & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_N^{j,m} & a_{N-1}^{j,m} & \cdots & a_{N-N_u+2}^{j,m} & a_{N-N_u+1}^{j,m} \end{bmatrix}, \quad (8)$$

where $a_i^{j,m}$ ($i = 1, \dots, D$) are step response coefficients of the control plant describing influence of m^{th} input on j^{th} output as the step response model has the following form:

$$y_k^j = \sum_{m=1}^{n_i} \sum_{i=1}^{D-1} a_i^{j,m} \cdot \Delta u_{k-i}^m + a_D^{j,m} \cdot u_{k-D}^m, \quad (9)$$

where Δu_k^m is a change in the m^{th} manipulated variable at the k^{th} sampling instant, D is equal to the number of time instants after which the coefficients of the step responses can be assumed as settled, u_{k-D}^m is a value of the m^{th} manipulated variable at the $(k-D)^{\text{th}}$ sampling instant.

The optimization problem (1) is solved by the numerical DMC algorithm at each iteration. As the result of optimization the vector Δu of future control increments is obtained. Then, the elements Δu_{klk}^j of the vector Δu are applied to the process and the optimization is repeated in the next sampling instant.

If constraints are not important in the particular control problem or a fast controller must be obtained, then the performance index from the problem (1) is minimized without constraints. As a result the following control law is obtained:

$$\Delta u = (A^T \cdot \Psi \cdot A + A)^{-1} \cdot A^T \cdot \Psi \cdot (\bar{y} - \tilde{y}). \quad (10)$$

This DMC control law can be formulated as:

$$\begin{bmatrix} \Delta u_{klk}^1 \\ \Delta u_{klk}^2 \\ \vdots \\ \Delta u_{klk}^{n_i} \end{bmatrix} = K^e \begin{bmatrix} \bar{y}_k^1 - y_k^1 \\ \bar{y}_k^2 - y_k^2 \\ \vdots \\ \bar{y}_k^{n_y} - y_k^{n_y} \end{bmatrix} - \sum_{j=1}^{D-1} K_j^u \begin{bmatrix} \Delta u_{k-j}^1 \\ \Delta u_{k-j}^2 \\ \vdots \\ \Delta u_{k-j}^{n_i} \end{bmatrix}, \quad (11)$$

where K^e is a matrix of dimensionality $n_i \times n_y$ and K_j^u , $j=1, \dots, D-1$ are matrices of dimensionality $n_i \times n_i$ composed of coefficients of the controller; these matrices and the resulting control law are calculated off-line. The detailed description of the DMC algorithm can be found, e.g. in [18].

3 Software Implemented Fault Injectors

Software Implemented Fault Injectors (SWIFI) have a numerous advantages over other fault injection techniques [3]. One of the most important is the very high level of controllability over injected disturbances and observability of fault propagation and effects. In the research two fault injectors are used: FITS and InBochs.

3.1 FITS Fault Injector

FITS is a specialized debugger as it uses Windows Debugging API to control the execution of the application under tests (AUT) [8, 19]. It can suspend/resume the AUT's threads, read/write AUT's memory (data, stack or code), CPU/FPU registers states, etc. The whole concept is depicted in Fig. 1. To examine the control algorithm, the reactive application is implemented that contains implementations of the control algorithm, the controlled process model (i.e. rectification column), the environmental disturbances, and the layer for data exchange between the controller and the controlled process. FITS disturbs only during the execution of selected parts of AUT – called *testing areas*. Separated code and data of the controller from the rest of the emulated environment allow FITS to selectively disturb only the controller.

FITS can emulate the existence of faults of different types (stuck-at, bit-flips, etc.) or even can emulate complex fault or error models. However, the most commonly used in the community is the single bit-flip fault as it well mimics the occurrence of Single Event Upsets (SEU [1, 3]).

To examine the fault sensitivity of the given fault location, the set of tests is performed (i.e. *experiment*). Each test is single execution of the AUT with the disturbance of a given type injected. During the test FITS pauses the AUT at the precisely defined time instant of its execution, injects the fault into the target fault location (e.g.

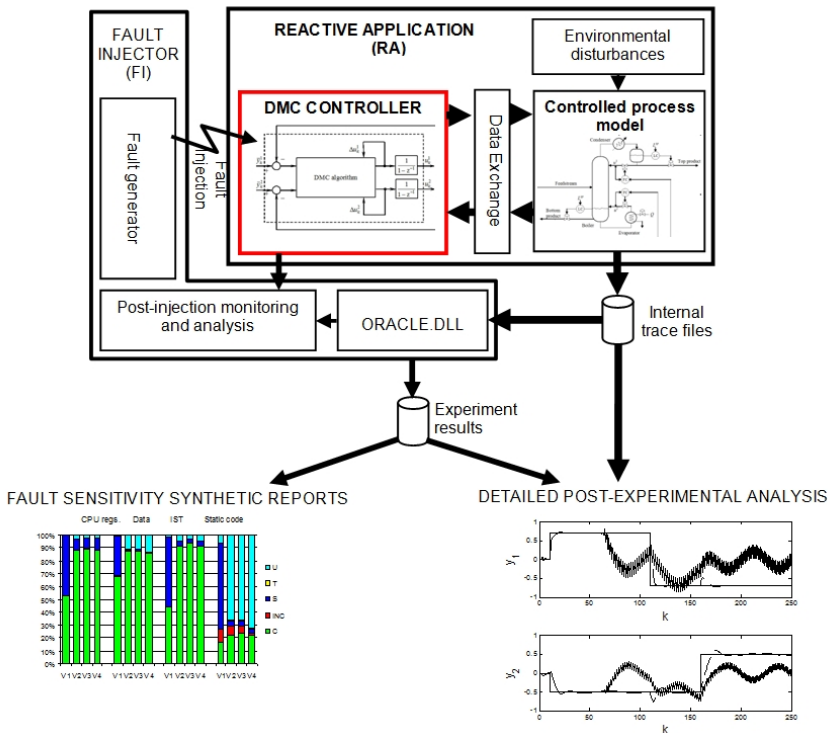


Fig. 1. FITS – detailed concept of fault injection approach

CPU register, memory location, instruction code), and (after AUT resumes the execution) collects (with the post-injection monitoring module) all events occurring in AUT (like exceptions triggered by AUT, messages, result files, etc.). As the AUT finishes its execution the results are judged for correctness by FITS oracle module on the basis of the data collected during the referential (i.e. fault-free) AUT execution (called *golden run*). The whole experiment is conducted by FITS automatically. At the end of the experiment summarized results are given. In general, four classes of test results are distinguished: *C* (correct results produced), *INC* (incorrect/unacceptable results), *S* (test terminated by the system due to un-handled exception, e.g. memory access violations, invalid opcode), and *T* (timed-out test). Additionally, if the application signals the user that error is detected before the termination, the *U* category may also be present (a user being warned).

The examined implementation of the explicit control algorithm takes 124 machine instructions (405 bytes of the static code). The algorithm needs execution of 1020000 instructions for the whole simulation horizon (300 discrete sampling instants). The numerical DMC controller introduced significant increase of the code and run time in terms of the number of executed instructions – the reference execution (100 discrete sampling instants) required 6811 machine instructions in code and in execution of 1 522 115 055 instructions at runtime. Unfortunately, such big number of instructions is not possible to be handled by the FITS fault injector in reasonable time. FITS efficiency of the golden run execution is at the level of 20÷30 thousands of instructions per second depending on the target machine performance. So, approximately 14 hours would be required only for the golden run execution.

In case of fault injection tests FITS offers high level of controllability over the place and time instants of injections (each execution instant of particular machine instruction is distinguished). That is implemented as trapping the execution at the fault triggering instruction and the bypassing action (if the required instant is not reached): the breakpoint is disabled, the triggering instruction is executed in single-step mode and the breakpoint is enabled once again. Described bypassing introduces four context switches. The efficiency of this process is around 16 000 of bypassed iterations per second. The performance problems described here are resolved with the new fault injection system – InBochs.

3.2 InBochs Fault Injector

The InBochs is based on the open-source x86 system emulator Bochs project [4]. The concept is presented in Fig. 2. The whole system is purely emulated – no virtualization features are used. However, Bochs executed at the single core of double core AMD Opteron 280 (running at 2.4 GHz) assures the performance at the level of 30 millions of machine instructions per second enabling quite smooth work with the guest operating systems. As the InBochs extends the original Bochs emulator with fault injection capabilities, the InBochs can also be executed on a variety of hosting operating systems (e.g. Windows, Linux, Solaris, MacOS) and it supports any available x86 operating systems as guest OS. The impact of the injector module within the InBochs (see Fig. 2) is practically imperceptible as it uses the Bochs internal structures and functions to pause the system execution at the precisely defined time instant of the AUT execution and to disturb the AUT context.

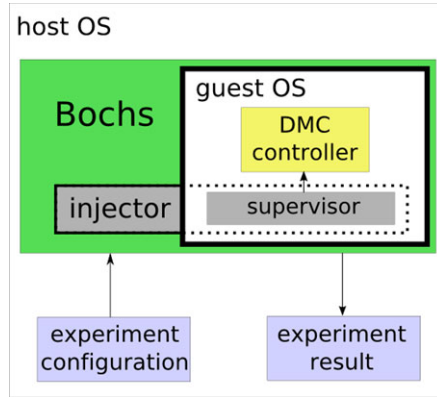


Fig. 2. InBochs fault injector concept

InBochs has the whole functionality of FITS. However, the fault injection is realized by the interpretation of the script in a script language dedicated for injectors. Experiment showed that such interpreter is not introducing much time delays while it significantly increases the flexibility of the fault injection system (e.g. new fault models can be easily implemented as scripts).

To conduct the experiments the *supervising* application has to be executed at the guest OS. It is responsible for the communication with the InBochs's injector module – it initiates the experiment starting the AUT within the guest OS, provides the AUT's process identifier to the injector module, and (at the end of the test) sends the results to the InBochs.

Contrary to FITS, the AUT execution is not debugged and execution performance is the same during the experiments as during the normal system emulation. So, after crossing a certain threshold the test duration can hopefully be shorter in InBochs. Figure 3 presents the comparison of test duration in both fault simulators in the function of the triggering instruction iteration instance. The tested application executes 1000 machine instructions in a sequential block within the loop. As the fault is injected into further loop iterations, the time needed by FITS increases significantly (it is dependent to the injection moment) while in the case of InBochs it only depends on the total number of executed instructions within the whole test. However, in case of Windows systems guests, the supervising application requires additional time for the initialization of Windows Debugging Tools (from 8 to 20 seconds – that time is constant for each test). In case of Linux guest the initialization of the supervisor is only 0.3 seconds. Please note that this time delays are not considered in Fig. 3.

In practice, if the fault triggering iteration is bigger than 1 million, the test will probably be faster on InBochs in case of the Windows target. On Linux that threshold is much lower (30 thousands). Nevertheless, despite the target operating system, the InBochs outperforms FITS in golden run execution (see Sect. 3.1) – for the considered numerical DMC implementation golden run takes 230 seconds (approx. 14 hours in FITS). Additionally, InBochs has access to some very internals of the system while FITS can access only user application. To conclude, it is worth to stress that despite some drawbacks, the InBochs solution has much bigger potential of use.

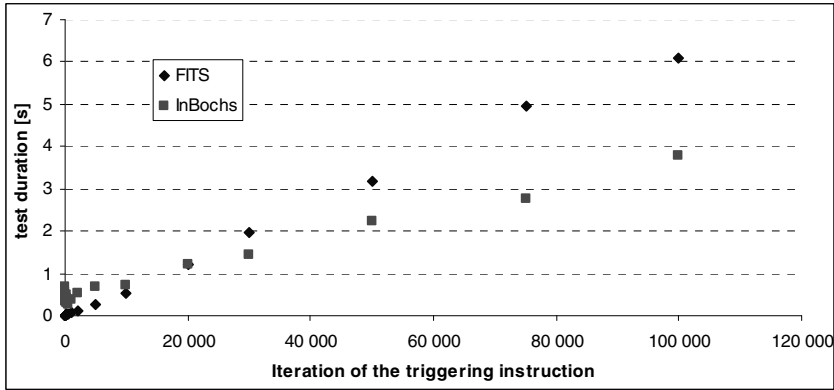


Fig. 3. The comparison of the test duration for FITS and InBochs

4 MIMO Process Description and Experiment Set-up

The considered process is a rectification column with two manipulated (inputs) and two controlled (outputs) variables (shown in Fig. 4). The process has strong cross-couplings and significant delays. It is described by the continuous-time transfer function model [20] (time constants are given in minutes):

$$\begin{bmatrix} Y^1(s) \\ Y^2(s) \end{bmatrix} = \begin{bmatrix} \frac{12,8}{16,7s+1} & \frac{-18,9e^{-4s}}{21,0s+1} \\ \frac{6,6e^{-8s}}{10,9s+1} & \frac{-19,4e^{-4s}}{14,4s+1} \end{bmatrix} \cdot \begin{bmatrix} U^1(s) \\ U^2(s) \end{bmatrix} + \begin{bmatrix} \frac{3,8e^{-4s}}{14,9s+1} \\ \frac{4,9}{13,2s+1} \end{bmatrix} \cdot U^3(s). \quad (12)$$

where the controlled variables are: y^1 – methanol concentration in the distillate (the top product), y^2 – methanol concentration in the effluent (the bottom product), the manipulated variables are: u^1 – flow rate of the reflux, u^2 – flow rate of the steam into a boiler, u^3 is feed flow rate (a disturbance). All process variables are scaled.

For the considered rectification process the numerical and explicit DMC algorithms were designed. In the case of the numerical implementation the sampling period $T_p=2$ min was assumed. The dynamics and prediction horizons were assumed equal to $D=N=50$. Other values of the parameters of the controller are: the control horizon $N_c=25$ and the values of coefficients: $\psi^1=\psi^2=1$, $\lambda^1=\lambda^2=10$. The simulation horizon is 100 sampling instants. In the case of the explicit version the sampling period $T_p=1$ min is assumed, the dynamics horizon is equal to the prediction horizon $D=N=100$, the control horizon $N_c=50$, the values of coefficients are: $\psi^1=\psi^2=1$, $\lambda^1=\lambda^2=10$, and the simulation horizon is 300 discrete sampling instants.

The simulation scenario is as follows. At the beginning, the process is driven to a given set-point. Then, at sampling instant 30 the change in the feed stream flow rate (u^3) is introduced (from 0 to 0.1). Another change in u^3 is made at the instant 140 (from 0.1 to -0.05). However, in the case of the numerical implementation only the first change of the feed stream flow rate manifests as we limited the overall simulation horizon for that version.

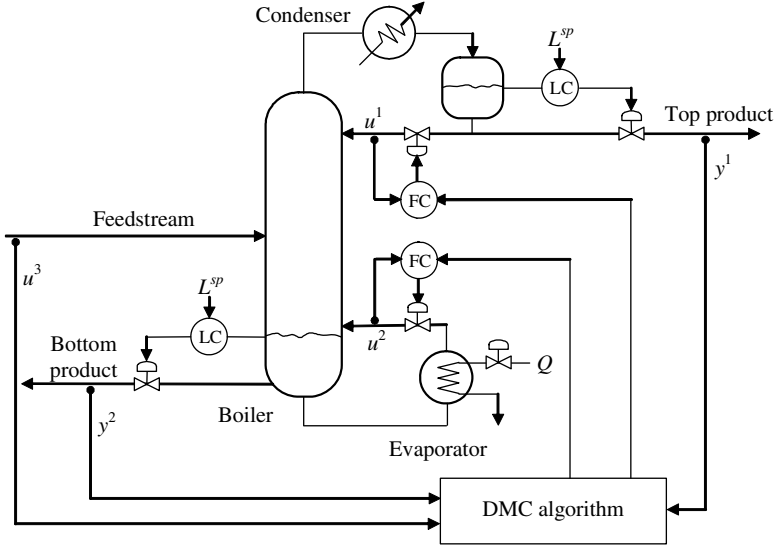


Fig. 4. Structure of the control system of the rectification column

Fault simulators used in this research (Sect. 3) can limit the scope of disturbances only to the selected parts of the application. Here, only the code of the control algorithm and its data are disturbed. The tested applications are also instrumented to send some measures (e.g. related to values of internal variables, output signal deviations, and signalisation of failures detected by the controller itself) to the fault injector using user-defined messages (collected by fault simulators – [8, 19]).

In this study the single bit-flip faults within CPU registers, application's data and machine instruction code (latching and non-latching [19]) are considered. Faults are injected pseudorandomly in time of the program execution and in space (bit position within the disturbed resource, distribution over application's memory) to mimic Single Event Upset (SEU) effects [1, 2, 3]. To get better insight into the numerical DMC implementation fault sensitivity, some specific fault injection policies are also considered (described later). One fault is injected per single run of simulation. At the end of each test run the qualification of control performance is assessed (Sect. 3). Here, the *SSE* factor (Sum of Squared Errors – calculated over y_1 and y_2) is used (13)

$$SSE = \sum_{k=1}^n \left((\bar{y}_k^1 - y_k^1)^2 + (\bar{y}_k^2 - y_k^2)^2 \right) \quad (13)$$

where n denotes the simulation horizon (Sect. 2). The reference *SSE* value for considered output trajectories from fault-free execution is 2.53 (\bar{y}_k^1 , \bar{y}_k^2 are corresponding set-point values). Because of a dynamic nature of the process, the *SSE* value is different from 0, as it takes some time to reach desired reference output values. Experiments shown that responses with $SSE < 5$ can be qualified as correct ones. However, the threshold *SSE* must be chosen arbitrarily by an expert (Sect. 5). By changing *SSE*

threshold we can admit various levels of the control quality. Moreover, it is possible to detect temporary violations of safety conditions.

Analysis of fault effects requires detailed information on the faults injected and the application behaviour. FITS provides details about every test (simulated fault injection). Hence, manual replay of the whole test execution can be done. Moreover, all the events and user messages occurring during the test are recorded. The tested application is instrumented to save its outputs (here simulation results, i.e. a set of process signals (u^1, u^2, y^1, y^2) in subsequent sampling instants) into separate files for each test (file names managed by fault injectors). This gives a possibility for post-experiment analysis of fault effects in the correlation with the injected fault and observed behaviour for each test (see Sect. 5).

5 Fault Sensitivity Experiments

The explicit DMC version is written in C language and compiled using Microsoft Visual C++ 2005. The fault sensitivity analysis is conducted with the described FITS fault injection system (Sect. 3.1) [9]. It is worth noting that floating point instructions constitute 38% of the code in the static code while dynamically they are executed in 54,8% of time. Moreover, instructions organizing computational loops in the DMC implementation (*sub, test, add, jnz, jg*) take another 38%. Hence, the application is strongly computational with high degree of FPU utilization and rather limited instruction set. On the other hand, the activity ratio for CPU resources is high (98, 94, 80, 98, 97, 81 % for EAX, EBX, ECX, EDX, ESI and EDI, respectively) [8].

Faults in the CPU and FPU registers, data area of the application, executed instruction stream, and static code image are considered. For each fault location approximately 1000 disturbed executions are investigated (single fault injected in each application execution). The summary of results (according to categories described in Sect. 3) is presented in Fig. 5 (explicit DMC [9]). Figure 6 presents results of the numerical DMC implementation for comparison (discussed later on).

As the explicit algorithm uses many parameters (400) it is very robust to fault located in the data area – there are only few data memory locations critical for the algorithm – most of the data correspond to the algorithm's parameters. Also the high degree of FPU robustness could be astonishing. Past experience shows that the FPU is rarely used hard [7] (e.g. only few FPU stack locations used simultaneously). This results in overall low fault sensitivity of the FPU. Nevertheless, there are some very sensitive locations within the FPU (e.g. control registers). Moreover, the analysis of used floating point value ranges showed that the considered single bit inversions are unlikely to provoke much difference in the value of disturbed variable. Similar effect was reported in [19]. The most fault sensitive resource of the DMC controller is its code. Fortunately, there are software techniques (e.g. exception handling, duplication of critical data and code) that can be applied at the source code level to provide fault robustness ([8, 11] and references therein). Despite that, the rare errors on the controlled process inputs are not critical for its behaviour.

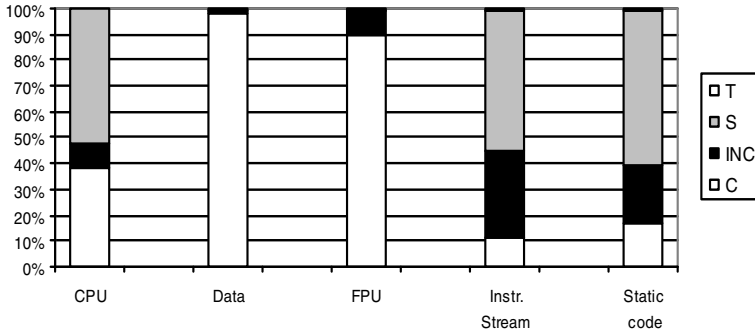


Fig. 5. Experimental results of the explicit DMC implementation; FITS fault injector; categories of results: *C* - correct results produced, *INC* - incorrect/unacceptable results, *S* - test terminated by the system, and *T* - timed-out tests

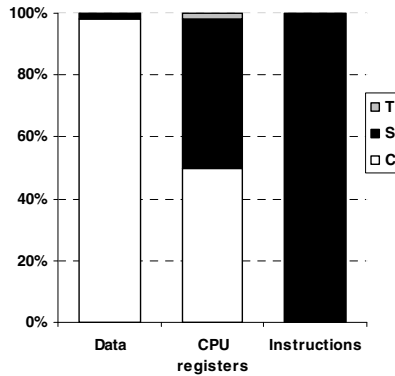


Fig. 6. Experimental results of the numerical DMC implementation; InBochs fault injector

Analyzing fault susceptibility it is worth correlating the observed effects (the simulated process behaviour) with the injected fault details in accordance to the source and machine code of the disturbed DMC. Figures 7 and 8 present plots of the application outputs (y^1, y^2 – left plots and u^1, u^2 – right plots) over the explicit DMC iteration number in case of sample fault disturbed executions [9]. For reference the undisturbed simulation results are shown, i.e. the golden run (solid lines).

In the case considered in Fig. 7a, the fault is injected at the sampling instant 28. It results in the change of the *faddp* instruction into the *fmulp* (operands remained unchanged). The instruction disturbed is used to calculate the *du1* variable of the DMC algorithm (corresponding to the Δu vector element – see Sect. 2). As a result the source code statement `du1+=r1[i]*vektup[i]` is changed to `du1*=r1[i]*vektup[i]`. The result of this disturbance varies on the control signal and process states. For instance, the considered fault injected at the 28th sampling instant results in $SSE=3.39$ (Fig. 7a), at the 101st sampling instant in $SSE=4.25$ (Fig. 7b), at the 224th – $SSE=2.53$ (Fig. 7c) (the same as the reference SSE value). Hence, it disturbs the top product composition.

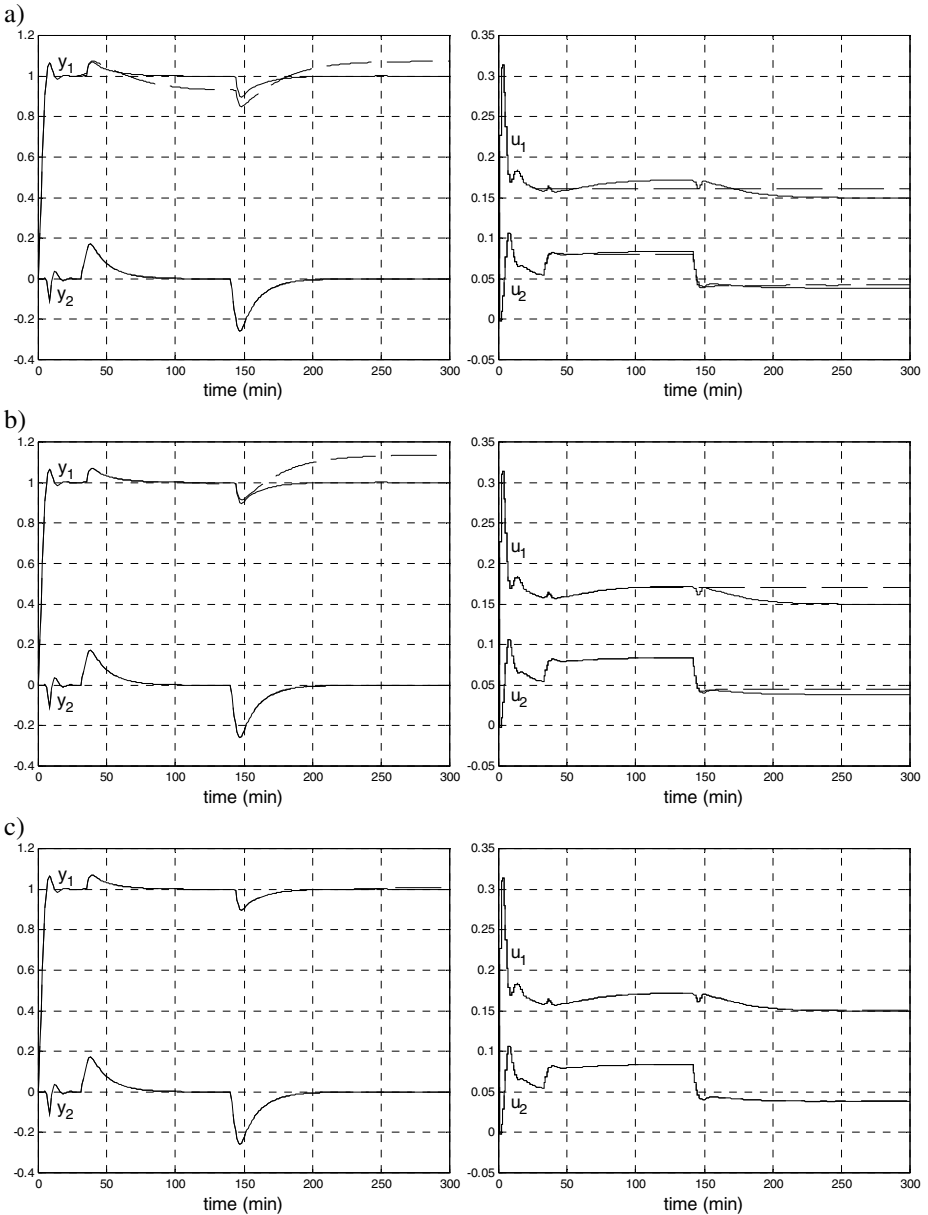


Fig. 7. Single bit inversion within the $faddp$ disturbs the top product composition. Faults injected at sampling instants: a) 28th, $SSE=3.39$, b) 101st, $SSE=4.25$, c) 224th, $SSE=2.53$; golden run responses – solid line, fault injected responses – dashed line.

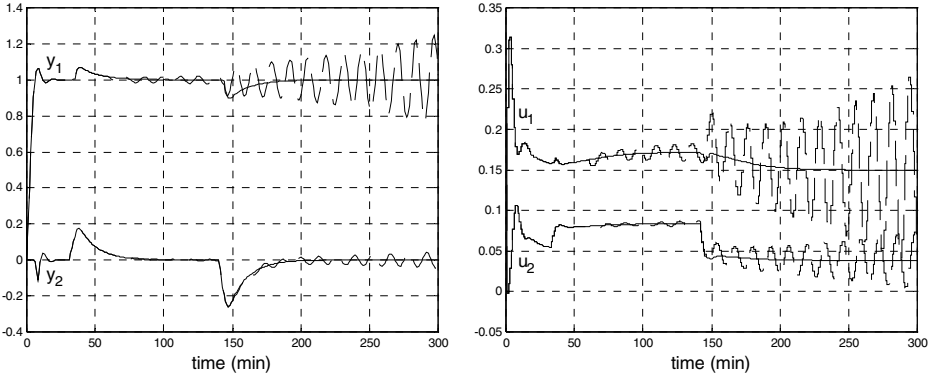


Fig. 8. Single bit inversion within the *faddp* destabilizes the process. Fault injected at the 61st sampling instant, $SSE=4.40$; golden run responses – solid line, fault injected responses – dashed line.

It is worth noting that the overwhelming majority of incorrect behaviour relates to very high SSE values (higher than 1000). It means that the values of control errors cumulate. On the other hand it is possibly easier to detect such big deviations using additional diagnostic subroutines (e.g. assertions on DMC variables) like in [11].

However, some rare critical situations with relatively low SSE deviation were observed (as in Fig. 8). Single bit inversion (at 61st sampling instant) within the same instruction as described above destabilises the process. In this case the instruction mnemonic remains unchanged while its first operand changed from *st(2)* to *st(0)*. Observed SSE is 4.40. Similar problems were identified in the results of the numerical DMC implementation (discussed later on).

A series of tests of fault robustness of the numerical DMC algorithms were performed with the InBochs system (Sect. 3.2). The overview of the results is presented already in Fig. 6. As one can see, the numerical implementation is even more fault robust than the explicit one. Unfortunately, due to long test durations, the number of tests for each presented fault location is limited to 200. In these experiments no incorrect result was noticed. In fact, this can be explained by the application specificity. The numerical DMC implementation uses a lot of very large matrices of parameters. Moreover, they are dynamically allocated resulting in significant ratio of memory addressing instructions. Previous research showed that faults affecting such applications usually finish their execution correctly or applications are terminated by the operating system due to unhandled exceptions (typically the memory access violations) [7].

To get better insight into the fault sensitivity, two more experiments were conducted. The target fault locations were the model parameters of the process in two matrices. For each matrix 20 tests were performed. To provoke more destructive effects, the injected faults inversed the whole byte (the second after the most significant one) of the double format parameter (randomly chosen) within the given matrix. Analysis showed that such faults result in the most significant changes of the target parameters. As already mentioned, this is unique property of the floating point values – bit inversions are very unlikely to provoke big value deviations. Also in these

experiments the most frequent situations were the differences of the *SSE* at the level of 0.001 and less – not noticeable in practice.

Despite that, in the mentioned additional experiments some interesting situations were reported. Example responses are shown in Figs 9 and 10. The undisturbed simulation results (the golden run) are shown in each figure for comparison (solid lines). At the beginning, the process is driven to a given set–point. Then, at 30th minute of simulation the change in the feed stream flow rate (u^3) is introduced (from 0 to 0.1). The *SSE* index calculated for the golden run is equal to $SSE=2.53$.

In the case considered in Fig. 9, the fault is injected at the beginning of simulation and $SSE=2.79$. As a result the bottom product response almost does not change. However, the top product composition is disturbed – it is shifted towards bigger values. Moreover, the steady–state control error is nonzero. Such a situation should be detected but it is not critical. It is possible to continue process operation till the controller is fixed.

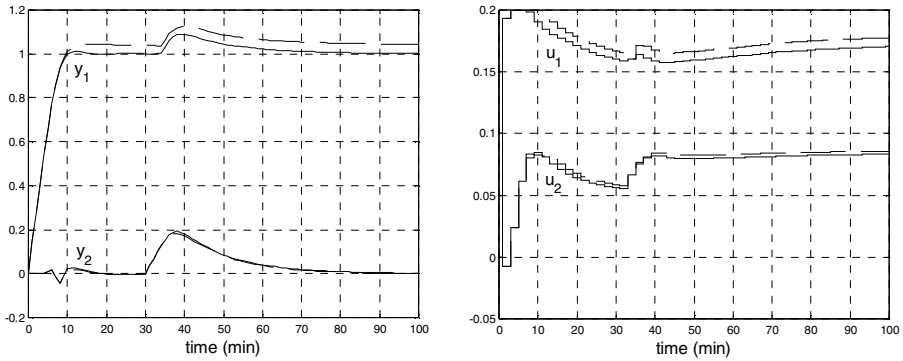


Fig. 9. Responses obtained after fault injection at the beginning of simulation; $SSE=2.79$; golden run responses – solid line, fault injected responses – dashed line

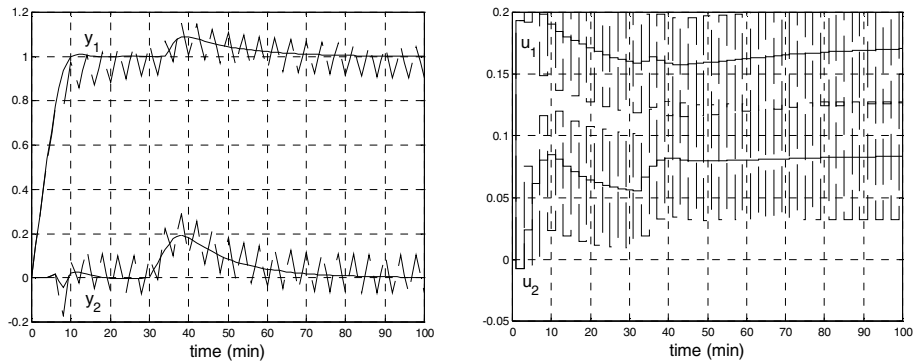


Fig. 10. Oscillatory responses obtained after fault injection at the beginning of simulation; $SSE=3.38$; golden run responses – solid line, fault injected responses – dashed line

More critical situation is illustrated in Fig. 10. The $SSE=3.38$ is bigger than in the previous case. All signals are oscillating. However, fortunately, they are oscillating near the golden run responses and are not growing. The above results showed that the SSE value threshold for the correctness judgment has to be revised after detailed analysis and it should probably involve more sophisticated oracle procedure (e.g. detection of oscillations, violations of control constrains).

6 Conclusion

The paper deals with fault-sensitivity evaluation of software implementations of the numerical and explicit DMC algorithms. The fault sensitivity was evaluated by fault simulation. The new fault injection system based on system emulator can handle long lasting experiments which was not possible with the old FITS fault injector used previously [8, 10, 11, 19]. Results proved that the InBochs fault injector is capable of such tasks. Moreover, the presented InBochs system delivers the same functionality and the same level of fault injection controllability and effects observability as the old one. Additionally, it introduces very flexible scripting language (e.g. providing the capability to extend fault models) and promises wider range of fault targets (i.e. new operating systems, fault locations in different system components, etc.). Further research will extend the InBochs capabilities in the above mentioned directions.

An interesting observation is that a large number of faults do not result in control errors. A detailed analysis of the controller behaviour revealed various kinds of natural (intrinsic) redundancy. We have found that even in the case of sophisticated control algorithms involving quite long history of state variables the natural fault tolerance capabilities are quite high. Moreover, they can be improved significantly with simple software mechanisms. On the other hand, we have identified unstable behaviour of the system for some faults, which needs special treatment. The explicit DMC implementation was hardened and described in [9, 10]. Analogous software-based fault hardening is planned to be designed and evaluated for the numerical DMC implementation.

Acknowledgements. P. Gawkowski and J. Sosnowski realised this research within grant 4297/B/T02/2007/33 from Polish Ministry of Science and Higher Education.

References

1. Anghel, L., Leveugle, R., Vanhauwaert, P.: Evaluation of SET and SEU effects at multiple abstraction levels. In: 11th IEEE IOLTS Symposium, pp. 309–314. IEEE Press, Los Alamitos (2005)
2. Arlat, J., Crouzet, Y., Karlsson, J., Folkesson, P., Fuchs, E., Leber, G.H.: Comparison of physical and software implemented fault injection techniques. *IEEE Transactions on Computers* 52(9), 1115–1133 (2003)
3. Benso, A., Prinetto, P.: *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Kluwer Academic Publishers, Dordrecht (2003)
4. The Bochs project homepage, <http://bochs.sourceforge.net/>

5. Corno, F., Esposito, E., Reorda, M., Tosato, S.: Evaluating the effects of transient faults on vehicle dynamic performance in automotive systems. In: ITC 2004, pp. 1332–1339. IEEE Press, Los Alamitos (2004)
6. Cutler, R., Ramaker, B.: Dynamic matrix control – a computer control algorithm. AIChE National Meeting, Houston (1979)
7. Gawkowski, P., Sosnowski, J.: Dependability evaluation with fault injection experiments. IEICE Transactions on Information & System E86-D, 2642–2649 (2003)
8. Gawkowski, P., Sosnowski, J.: Experiences with software implemented fault injection. In: International Conference on Architecture of Computing Systems, pp. 73–80. VDE Verlag GMBH (2007)
9. Gawkowski, P., Ławryńczuk, M., Marusak, P., Sosnowski, J., Tatjewski, P.: Dependability of the software implementation of the explicit DMC algorithm. IADIS International Journal on Computer Science and Information System 8(1), 44–58 (2008)
10. Gawkowski P., Ławryńczuk M., Marusak P., Sosnowski J., Tatjewski P.: Software Implementation of Explicit DMC Algorithm with Improved Dependability. In: T. Sobh, et al. (eds) Novel Algorithms and Techniques In Telecommunications, Automation and Industrial Electronics, pp. 214–219, Springer (2008)
11. Gawkowski, P., Ławryńczuk, M., Marusak, P.M., Tatjewski, P., Sosnowski, J.: On improving dependability of the numerical GPC algorithm. In: Proc. European Control Conference 2009, Hungary, pp. 1377–1382 (2009)
12. Ławryńczuk, M., Marusak, M., Tatjewski, P.: Cooperation of model predictive control with steady-state economic optimisation. Control and Cybernetics 37, 133–158 (2008)
13. Maciejowski, J.M.: Predictive control with constraints. Prentice-Hall, Harlow (2002)
14. Mariani, R., Fuhrmann, P., Vittorelli, B.: Fault Robust Microcontrollers for Automotive Applications. In: Proc. IEEE On-line Test Symposium, pp. 213–218. IEEE Press, Los Alamitos (2006)
15. Marusak, P., Tatjewski, P.: Actuator Fault Tolerance in Control Systems with Predictive Constrained Set-Point Optimizers. International Journal of Applied Mathematics & Computer Science 4, 539–551 (2008)
16. Morari, M., Lee, J.H.: Model predictive control: past, present and future. Computers and Chemical Engineering 23, 667–682 (1999)
17. Rossiter, J.A.: Model-based predictive control. CRC Press, Boca Raton (2003)
18. Tatjewski, P.: Advanced control of industrial processes. In: Structures and algorithms. Springer, London (2007)
19. Trawczynski, D., Sosnowski, J., Gawkowski, P.: Analyzing Fault Susceptibility of ABS Microcontroller. In: Harrison, M.D., Sujan, M.-A. (eds.) SAFECOMP 2008. LNCS, vol. 5219, pp. 360–372. Springer, Heidelberg (2008)
20. Wood, R.K., Berry, M.W.: Terminal Composition Control of a Binary Distillation Column. Chemical Engineering Science 28, 1707–1717 (1973)

Towards Net-Centric Cyber Survivability for Ballistic Missile Defense*

Michael N. Gagnon, John Truelove, Apu Kapadia**,
Joshua Haines, and Orton Huang

Massachusetts Institute of Technology, Lincoln Laboratory
244 Wood Street, Lexington MA, 02420, USA
{michael.gagnon,jtruelove,jhaines,orton}@ll.mit.edu
Indiana University, School of Informatics and Computing
901 E 10th Street, Bloomington IN, 47408, USA
kapadia@indiana.edu

Abstract. The United States Department of Defense (DoD) is engaged in a mission to unify its software systems towards a “net-centric” vision—where commanders gain advantage by rapidly producing, consuming, and sharing information using service oriented architectures (SOAs). In this paper, we study the cyber survivability of mission-critical net-centric systems, focusing on Ballistic-Missile-Defense (BMD) systems. We propose a net-centric architecture for augmenting the survivability of critical DoD net-centric systems. Our architecture draws inspiration from several theories of warfare, focusing on the goal of giving cyber commanders “decision superiority.” Our architecture prescribes a net-centric decision-support system that implements the *Cyber OODA loop* (the cycle of observing, orienting, deciding, and acting within the cyber domain). We present an illustration-of-concept prototype implementation, and describe its role in a ballistic-missile exercise. We relate our experiences from this exercise and suggest future directions towards achieving net-centric cyber survivability.

1 Introduction

Faced with a multitude of special-purpose systems suited to individual missions such as Ballistic Missile Defense (BMD) and Space Situational Awareness (SSA), the United States Department of Defense (DoD) has recognized the need to integrate these systems into a *service oriented architecture (SOA)*. SOAs allow various entities to maintain control of their critical systems, and yet offer their *services* over the network. For example, a radar will be able to track both satellites and missiles in support of either space or BMD missions—previously

* This work is sponsored by the Department of Defense under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

** This work was performed while Apu Kapadia was at MIT Lincoln Laboratory.

an unavailable capability. The DoD is thus engaged in a mission to unify its computer networks towards a “net-centric” vision, where various systems are available via the Global Information Grid (GIG).

We expect that adversaries may attempt to employ cyber attacks to interfere with the DoD’s SOA systems. For example, an adversary may accompany a ballistic missile attack with a multitude of cyber attacks designed to inhibit defensive missile counter-measures. It is therefore imperative that mission-critical net-centric systems operate dependably, even when under cyber attack.

We propose a net-centric architecture for augmenting the survivability of critical DoD network services. Our architecture draws inspiration from several theories of warfare, focusing on the goal of giving commanders “decision superiority” in the cyber domain. Commanders have decision superiority when they are able to make better and quicker decisions than their adversaries [5].

We evaluate our architecture in the context of a BMD demonstration. In November 2009, MIT Lincoln Laboratory demonstrated a proof-of-concept net-centric BMD decision-support system during the simulation of an intercontinental ballistic missile (ICBM) attack [10]. For this exercise, we developed and deployed an illustration-of-concept implementation of our survivability architecture. This exercise demonstrated the principles of our survivability architecture in a realistic simulation of a coordinated ICBM and cyber attack. Our experience developing this survivability architecture taught us several important lessons learned, which we share in this paper.

We make the following contributions:

- We propose that critical SOA systems support cyber survivability by employing our “Net-centric Cyber Decision-Support” (NCDS) architecture. The NCDS architecture represents an interesting point in the space of techniques for achieving cyber survivability. The NCDS architecture complements existing approaches to survivability by improving the user’s ability to (1) correct runtime faults and (2) collaborate with artificial-intelligence systems to produce higher-quality decisions.
- We instantiate the NCDS architecture as an illustration-of-concept implementation. We evaluate our implementation in the context of a simulated ICBM and cyber-attack exercise.
- We relate our experiences from developing our survivability architecture, and suggest future directions towards achieving cyber survivability for mission-critical SOA services.

The rest of our paper is organized as follows. Section 2 presents background material on the theory of warfare, which provides the inspiration for our survivability architecture. Section 3 presents our proposed survivability architecture. Section 4 presents our demonstration, including the scenario, implementation details, and results. In Section 5 we relate the lessons we learned while designing, implementing, and testing our survivability architecture. We conclude the paper in Section 6.

2 Background

The design of our survivability architecture draws from several theories of warfare. In particular our design uses concepts from information warfare, net-centric warfare, and decision superiority. In this section we present background material on these concepts and relate them to decision-support systems and cyber survivability.

2.1 Information Warfare

Information warfare is concerned with protecting, improving, and leveraging one's own information while simultaneously corrupting the adversary's information [15]. Two central concepts of information warfare are the related concepts of *situational awareness* and *decision-making processes*. Situational awareness is one's perception and comprehension of their environment [8]. A decision making process is simply a method by which an entity makes decisions. It could be a formally specified routine or an unspecified ad-hoc approach. Regardless of the level of formality, military strategists often conceptualize the decision-making process as a four-part cycle of observing, orienting, deciding, and acting—the *OODA loop* [3]. Figure 1 graphically presents a simplified graphical view of the OODA loop.

Actors attempt to achieve *decision superiority* by making better decisions more quickly. Adversaries attempt to prevent decision superiority by conducting *offensive information operations* that attempt to corrupt their opponent's decision-making processes [15,19]. Actors cannot enact good decisions if their observations are corrupted or if they cannot synthesize observations into satisfactory intelligence. While there are many ways to disrupt the OODA loop's feedback cycle, (including such things as physical attack and psychological operations), in this research we are principally concerned with cyber activities.

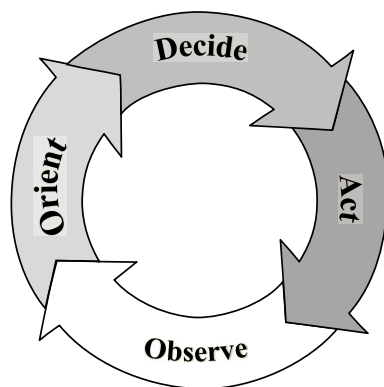


Fig. 1. The OODA loop

There are two specific OODA loops we discuss in this paper.

1. The *BMD OODA Loop* is the decision-making process enabling ballistic missile defense. This OODA loop includes all the systems that allow decision-makers to observe and analyze potential missile threats as well as to enact defensive counter-measures. A proof-of-concept service-oriented-architecture (SOA) system has been developed that strengthens the BMD OODA loop by giving decision-makers better access to observations and intelligence. We use this SOA system as the platform to evaluate the implementation of our survivability architecture; our architecture protects the BMD OODA loop from cyber attack.
2. The *Cyber OODA Loop* is the decision-making process enabling cyber-attack defense. Similar to the BMD OODA Loop, the Cyber OODA loop facilitates the observation and analysis of cyberspace and enables decision-makers to enact defensive counter-measures. Our survivability architecture augments cyber-survivability by strengthening the Cyber OODA loop.

While cyberspace generally refers to the global domain of interconnected computing systems, in this work we focus on the specific cyber domain that supports ballistic missile defense in the United States. Thus when we refer to the Cyber OODA loop we are referring to the specific decision-making process used to defend the BMD mission from cyber attack.

2.2 Decision-Support Systems

A decision-support system (DSS) is a system that improves a particular decision-making process by making it more effective, less risky, and faster. A DSS essentially aims to strengthen a particular OODA loop. For example, the SOA system for BMD is a decision-support system. Likewise, our cyber-survivability architecture is a DSS intended to augment the survivability of mission-critical software systems.

DSS-based survivability architectures are not new. Schwaegerl et al. propose to use a DSS to increase the survivability of complex power systems [17]. Lee et al. developed a DSS system that improves the survivability of damaged submarines [11]. Relating to cyber survivability, a wide variety of existing software systems can be viewed as decision-support systems that employ automated decision making to improve the survivability of software systems [9,18,16].

Our cyber-survivability architecture is a decision-support system. Our approach diverges from existing approaches by using *net-centric services* to facilitate human and automated decision making.

2.3 Net-Centric Warfare

The DoD has adopted the theory of network-centric warfare in its campaign to adapt to warfare in the information age [1]. Net-centricity promises to enhance mission effectiveness based on the following premises:

1. “A robustly networked force improves information sharing.
2. Information sharing enhances the quality of information and shared situational awareness.
3. Shared situational awareness enables collaboration and self-synchronization, and enhances sustainability and speed of command.
4. These, in turn, dramatically increase mission effectiveness.” [13]

Net-centric forces leverage high-quality information networks to strengthen their OODA loops. By rapidly sharing observations and intelligence, decision makers can make better decisions more quickly—achieving decision superiority.

Service-oriented architectures (SOAs) facilitate information sharing and collaboration, enabling net-centricity. Accordingly, the DoD is increasingly interested in developing SOA systems to facilitate mission-critical decision-support systems.

In the next section we outline our net-centric survivability architecture, which we use to augment the survivability of the net-centric BMD system.

2.4 Cyber Survivability

We consider the concept of *cyber survivability* as the ability of a system to continue to provide a specified level of mission functionality while being subjected to system faults [12].

We believe that engineers should follow a three-prong approach to building systems with cyber survivability. First, it is imperative that engineers follow engineering best-practices to maximize the robustness of net-centric SOA systems. Second, we believe that engineers should augment the survivability of net-centric SOA systems at run time by using cyber decision-support systems. Third, the architecture should be able to evolve to meet as yet unforeseen threats by using secure architectural principles.

3 The Net-Centric Cyber Decision-Support Architecture

As the DoD intends to use net-centric SOA systems during warfare, we expect these systems to be subject to attack. More importantly than usual, such mission-critical systems must be robust to system faults that result from either innocent error or malicious attacks.

As defined in Section 2, we believe that engineers should augment the cyber survivability of their systems by using cyber decision-support systems. In this section we describe such systems as they apply to net-centric architecture: *net-centric cyber decision-support* (NCDS) systems.

An NCDS system utilizes net-centric SOA services (*cyber services*) to strengthen the Cyber OODA loop. By strengthening the Cyber OODA loop, NCDS systems allow decision makers to intelligently respond to system faults, such as those caused by cyber attacks. We prescribe a particular architecture for building NCDS systems. Our NCDS architecture prescribes six roles that cyber services and human operators fulfill.

1. *Cyber sensors* monitor events in cyberspace.
2. *Cyber analyzers* coalesce sensor data, provide situational awareness, and synthesize actionable intelligence.
3. *Decision-mediators* facilitate decision-making by presenting situational awareness and potential actions to decision makers in an intelligent manner.
4. *Automated decision-makers* produce “reflex” decisions.
5. *Human decision-makers* produce “cognizant” decisions.
6. *Actuators* enact decisions.

These roles facilitate both localized and distributed decision making. For example, a particular server could contain several sensors, analyzers, and actuators. In addition, it could contain its own decision mediator and automated decision maker. This way the server is capable of self-sufficiently performing reflexive decision making. In addition, the server publishes its own observations and analysis, contributing to global situational awareness.

In the following subsections we describe the components of our NCDS architecture in greater detail.

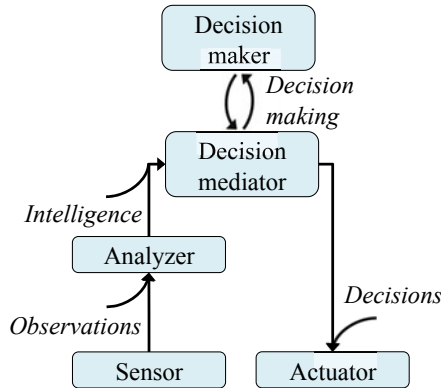


Fig. 2. Our NCDS architecture component implements the Cyber OODA Loop

3.1 Cyber Sensor Services

Cyber sensors directly observe the cyber environment. Cyber sensors should be *indicative*. An indicative sensor reveals important information about system state, such as the presence or cause of faults.

To maximize situational awareness, a variety of indicative sensors should be placed throughout the network such as on network devices as well as on desktop and server systems. These sensors will provide direct observations that—once transformed into intelligence—will give decision makers critical information necessary to enact decisions to remedy faults.

In order for observations to be useful, they must be communicated to decision makers (via intermediary services). Each cyber sensor is subsequently a net-centric service. Individual cyber analyzers subscribe to the specific observations that they require for their analysis. As a sensor observes its domain, it continually publishes observations to its subscribers.

3.2 Cyber Analyzer Services

Sensors merely observe and report events. Without analysis, raw sensor data is often meaningless to decision makers. *Cyber analyzers* synthesize actionable intelligence from raw sensor data. For example, an application sensor might monitor program control flow, while a complementary analyzer would determine if the observations were suspicious. If an analysis reveals suspicious behavior, the analyzer would summarize the observations in a way that would be meaningful to a decision maker.

Because an NCDS system will likely contain many sensors and analyzers, we expect that it would be overwhelming to allow analyzers to directly report intelligence to decision makers. To manage information overload, analyzers report intelligence to decision makers through *decision mediators*.

3.3 Decision Mediator Services

When a system fault has occurred, one or more decision makers must study the intelligence and decide to enact one or more responses to remedy the fault. To facilitate decision making and manage information overload, each decision maker interacts with a single *decision mediator*. A decision mediator is an automated process that intelligently chooses the most relevant information to present to the decision maker. Additionally, the mediator culls the universe of possible responses, selecting the most promising actions to present to the decision maker.

Consider, for example, a scenario where an adversary launches a denial-of-service attack against a defended computer system. The attack produces deviations in application control flow, network traffic, and CPU utilization. Automated analyzers study observations (gained from the suite of cyber sensors) and notice that the deviations are indicative of an attack. The analyzers subsequently publish this intelligence to the relevant decision mediators. The decision mediators survey the analyses and guesses which information will be most useful to the decision maker. In this example, the decision mediator forms a belief that the analysis of CPU utilization will be most useful to its decision maker. The decision mediator then presents an abridged form of the analyses to the decision maker, emphasizing the importance of the CPU-utilization analysis. In addition, the mediator also presents a set of suggested responses to the decision maker.

This example illustrates the heuristic nature of the decision mediator. Due to the limitations of artificial intelligence, the mediator may fail to present the most relevant information and responses to the decision maker in its first attempt. It is therefore important that decision mediation be an interactive process. When the decision maker is unsatisfied with its information or response palette, it

redirects the mediator to re-analyze and re-present the intelligence and palette of responses. This interaction continues until the decision maker chooses to enact one or more responses to remedy the fault.

3.4 Decision Makers

Our NCDS architecture prescribes employing both human decision makers and automated decision makers to achieve decision superiority. Automated decision makers provide *reflex* decisions—i.e. quick and safe decisions. We say a decision is safe if it is unlikely to cause harm or when its harm can easily be undone. Human decision makers provide *cognizant* decisions—potentially dangerous decisions that take into account the myriad information that cannot easily be modeled in the NCDS domain.

Our prescription constrains automated decision-makers to making safe decisions since we believe that state-of-the-art artificial intelligence (AI) is not mature enough to enact potentially dangerous decisions. This reflects current DOD doctrine that AI is too immature to allow unmanned systems to automatically make the decision to fire lethal weapons [14]; in the cyber domain, dangerous decisions may also have lethal consequences. With regards to ballistic missile defense, for example, a poorly chosen firewall rule could accidentally block radar communication—breaking the BMD OODA loop and preventing missile defense.

Automated decision makers are important because they maintain a fast OODA tempo—contributing to decision superiority. However, automated decision makers are constrained by the decisions they make (via policy) as well as their cognitive ability to choose good decisions (via the limits of AI). To achieve decision superiority, it is therefore important to also employ human decision makers.

Like automated decision-makers, human decision-makers review intelligence and potential actions and direct actions when appropriate. However, we expect human operators (1) to have access to more situational awareness than can be encoded in an NCDS system and (2) to be better able to consider the consequences when choosing to enact potentially dangerous decisions.

3.5 Actuator Services

Cyber actuators enact decisions. Similar to cyber sensors, actuators are deployed throughout the network on servers, client machines, network devices, etc. For example, a firewall actuator could allow a decision maker to change the firewall rules for a particular firewall. In our illustration-of-concept demonstration, we use a hypervisor actuator to dynamically switch to an alternate guest operating system.

4 Demonstration

We implemented an illustration-of-concept NCDS system and demonstrated its operation during a simulated ballistic missile attack. During the demonstration

a human operator used the NCDS system to observe, analyze, and respond to a cyber attack conducted in coordination with the missile attack. We describe the demonstration here detailing the scenario, the components of our NCDS implementation, and the cyber attack.

4.1 Scenario

A proof-of-concept BMD net-centric decision support system was demonstrated during the simulation of an intercontinental ballistic missile (ICBM) launch in November 2009. During the demonstration, operators used the BMD net-centric system to receive situational awareness about the ICBM. While the projectile was in flight, the simulated adversary launched a denial-of-service cyber attack targeting a critical radar-sensor service. Our illustration-of-concept implementation of the NCDS architecture observed symptoms of the attack and sent an alert to a “cyber operator.” The intelligence indicated that the radar-sensor service’s operating-system kernel was consuming a disproportionate amount of CPU time. In response, the cyber operator directed the radar-sensor service’s hypervisor to switch to an alternate operating system and restart the service. This successfully mitigated the attack, since it was targeting a vulnerability that was specific to the previous operating system.

In the following subsections we describe the components of this demonstration in greater detail. We begin by briefly describing the net-centric BMD system, then follow with a description of the cyber attack. We conclude this section by describing the illustration-of-concept NCDS implementation that we used to mitigate the cyber attack.

4.2 Cyber Victim: Radar-Sensor Service

Our NCDS architecture was deployed during the demonstration of a proof-of-concept net-centric BMD decision-support system [10]. Similar to our NCDS architecture, this system uses SOA services to strengthen the Ballistic-Missile-Defense OODA loop.

One principal component of the net-centric BMD system is the Radar-Sensor Service, which publishes radar data that authorized users may subscribe to through a web interface. (These services are analogous to NCDS Sensor Services.) This radar data is consumed by subsequent analytical components, which ultimately present the data as an executive summary to an end user who acts as a decision-maker. Without the data feed from the radar-sensor service, situational awareness for the entire BMD mission is lost.

The radar-sensor and subsequent components are implemented as SOA services in Java. For the demonstration, each service executed on its own computer system. To facilitate integration with the cyber sensor and actuator, the Radar-Sensor Service ran within VMware Workstation (the host operating system was Linux CentOS 5, while the guest operating system alternated between Linux CentOS 5 and FreeBSD). The other services executed on non-virtualized Linux Red Hat hosts, though in principle they could just as easily have been virtualized.

4.3 Cyber Attack

We expect that a capable adversary will use all available means to circumvent ballistic missile defense, including using information operations to weaken the defender’s ability to enact missile countermeasures. We implemented such a cyber attack that targets the Radar-Sensor Service. By disabling the Radar-Sensor Service, the cyber attack disrupts the BMD OODA loop—preventing BMD commanders from potentially enacting effective missile counter-measures.

We model an adversary that has gained execute access on a single computer within the mission network. From their perch inside the network, the adversary launches an *algorithmic-complexity* attack by sending a continuous stream of specially crafted packets to the computer hosting the Radar-Sensor Service.

A single computer cannot usually generate enough traffic to cause a denial-of-service with a simple packet flood. However, algorithmic-complexity attacks effect worst-case performance in vulnerable algorithms, allowing attackers to consume a disproportionate amount of victim resources [6]. Our attack causes disproportionate CPU consumption of the operating system running the Radar-Sensor Service. Our attack exploits a specific Linux-kernel vulnerability that was patched in 2003 [7,20]. We re-introduced the vulnerability on our system by “un-patching” the CentOS 5 kernel. With this vulnerability, a 34-byte exploit packet, transmitted at a sustained rate of 800 Hz (for an average bandwidth consumption of 26 kbps), is sufficient to cause a complete denial of service.

4.4 NCDS Implementation Overview

To illustrate how our NCDS system can augment the survivability of mission-critical software systems, we developed an illustration-of-concept implementation of the NCDS architecture. We tailored our implementation to mitigate the demonstration’s particular cyber attack. Though the defense’s success was predetermined, the implementation is valuable for presenting the concept of operations (CONOPS) for our NCDS architecture. Further, we also believe that the implementation represents an effective approach for augmenting survivability through application-specific fault tolerance implemented using virtualization. This implementation also led to several lessons learned, which we describe in Section 5. Our implementation employs each component of the NCDS architecture described in section 3:

- **VTop** implements a **cyber-sensor** service that monitors CPU utilization. We also implemented a trivial **cyber-analyzer** service that publishes an alert when CPU utilization is suspicious (when kernel-space utilization is above a preset threshold).
- **VM-Switch** implements a **cyber-actuator** service that can dynamically switch the operating system of Radar-Sensor-Service instantiations.
- Our implementation only supports a **human decision maker**, the “cyber operator.” We provided a trivial implementation of the **decision-mediator** service. The decision-mediator provides the cyber operator with the output

of VTop, any alerts generated by the cyber-analyzer, as well as an interface to VM-Switch via a Java graphical user interface.

4.5 Cyber Sensor: VTop

Overview. The VTop Cyber Sensor monitors and publishes a specific statistic pertaining to CPU utilization. At a rate on the order of 1000 Hz, the sensor samples the processor's state, recording whether the processor is idle or is executing in user or kernel space. Every second, the sensor aggregates the tally and publishes the proportion of time spent idle or in user or kernel space. This sensor is indicative of system faults that manifest themselves as anomalous utilization patterns. For example, our cyber attack manifests itself by causing the CPU to spend approximately 100% of its time in kernel space.

Implementation. VTop uses virtualization technology to make its observations. The observed system (the Radar-Sensor Service computer) runs as a virtual machine within VMware Workstation. VTop uses VMware's VProbes facility to trace the execution of the observed system [2]. VTop is implemented as a 10-line VProbe script piped to a Ruby script that parses the VProbe output. A Java application publishes the latest window of observations every second.

VTop relies on performance isolation provided by VMware Workstation. VProbes (and hence VTop) is not adversely affected by the cyber attack since the performance of the guest operating system does not adversely affect the performance of the hypervisor. If VTop were implemented as a component of the observed system, then the attack would likely interfere with the sensor. Here we rely on the assumption that an adversary cannot affect the operation of the hypervisor; we essentially assume that the hypervisor is a trusted computing base (TCB). While this assumption is not warranted in our setup [4] we believe that it should be possible to implement a trustworthy hypervisor that is dependably immune to attack.

Results. VTop and its accompanying analyzer successfully detected the cyber attack. Our main concern was that VTop would impose undue computational and network overhead. Due to the difficulty of benchmarking hypervisors, it is difficult to ascertain precise quantitative measurements of VTop's computational overhead. For the demonstration though, we did observe that VTop's performance was satisfactory; it did not degrade the throughput of the Radar-Sensor Service.

From running informal tests offline, we measured VTop's network overhead. On average, each VTop sensor observation requires ~ 9.5 KB uncompressed (~ 1.5

¹ In our setup, it might be possible for an adversary to exploit a vulnerability in the host operating system. In addition, VMware Workstation itself might be vulnerable to attack. For example, it is possible that VMware workstation contains an algorithmic-complexity vulnerability that would allow an adversary to violate Workstation's performance-isolation property.

KB compressed). A large portion of this communication overhead is due to the message format imposed by our net-centric messaging protocol. In future work it will be important to significantly reduce the network overhead of sensors and analyzers since we expect them to continually publish data throughout the network.

4.6 Cyber Actuator: VM-Switch

Overview. The VM-Switch actuator uses the hypervisor to dynamically switch the Radar-Sensor Service’s operating system from Linux CentOS 5 to FreeBSD. Since the Radar-Sensor Service is implemented as a Java application, we are able to run the service on both Linux and FreeBSD. After the actuator switches to FreeBSD it restarts the Radar-Sensor Service.

This actuator is capable of remedying faults within the operating system. Since the demonstration’s cyber attack is specific to the Linux operating system, switching to FreeBSD makes the Radar-Sensor Services immune to the attack.

Fault Tolerance. Though this actuator is tailored to our demonstration, it represents the general technique of achieving fault tolerance through design diversity. Design diversity is a standard approach to achieving fault tolerance. In software systems, identical duplicates do not provide useful redundancy since software faults are usually caused by design errors [4]. Usefully redundant software is obtained by redundant copies that provide desired semantics with independent designs. The hope is that independent designs will experience faults under different circumstances. When one design experiences a fault, switch to an alternate design in the hope that the circumstances will not cause a fault in the alternate design. There are many open problems with using design diversity to achieving fault tolerance, but we are optimistic that future work will provide useful mechanisms for NCDS actuators.

Implementation. Like VTop, the VM-Switch actuator relies on virtualization to achieve its effect. Before the demonstration we prepared each VM (Linux and FreeBSD) by installing a copy of the Radar-Sensor Service. We used VMware’s `vmrun` utility to “pause” the FreeBSD VM and activate the Linux version. The guest VM communicates via the host’s network interface using network-address translation (NAT). When VM-Switch activates, it (1) pauses Linux and unpauses FreeBSD, (2) also updates the NAT to maintain the same IP address, and (3) restarts the Radar-Sensor Service in FreeBSD.

Luckily, the Radar-Sensor Service is semantically stateless since the service simply publishes data that it directly receives from the radar (i.e. the application does not need to maintain any state to operate). Being stateless simplifies the design and implementation since application state does not need to be transferred from the Linux VM to the FreeBSD VM during activation [2]. Our approach is still applicable to stateful applications. For such applications greater care must

² Although the service is *semantically* stateless, the service was not implemented to be stateless. As a consequence we were forced to transfer a small amount of “initialization state” between VMs during switches.

be taken to not only transfer state, but to also ensure that an attack cannot propagate through the transferred application state.

Results. As expected, VM-Switch successfully mitigated the attack. However, the response latency is longer than we anticipated. On average, the response took 33 seconds. This latency is largely due to the latencies imposed by our messaging framework. The VMware pause and un-pause operations contributed little latency. We expect that future work will show that significantly shorter actuation latencies are possible.

4.7 Cyber Analyzer and Decision Mediator

As mentioned before, the components developed for the cyber analyzer and decision mediator are trivial and included solely to provide an end-to-end illustration-of-concept of our NCDS architecture.

The cyber analyzer receives the CPU consumption of the target system, provided as a SOA service by VTop, and generates an alert message if the CPU consumption of the kernel exceeds a threshold for a given time (in this case, the CPU consumption only needs to be at 100% for over 1 second).

The decision mediator simply provides an interface to VTop and VM-Switch to a human operator. Figure 3 presents a graphical view of VTop output during an offline test of the NCDS implementation. During the demonstration the cyber operator reviewed VTop intelligence like this graph to detect and respond to the cyber attack. Figure 3 shows the progression from normal operation under Linux, through the attack, and finally remedied operation under FreeBSD once the cyber operator initiated the response.

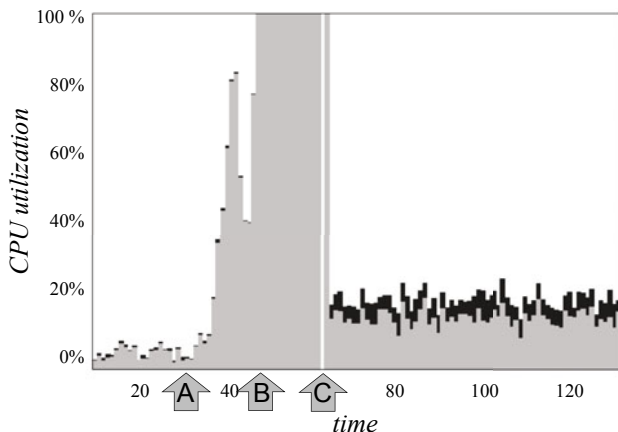


Fig. 3. Graphical view of VTop output over a two minute window (120 seconds). Legend: idle (white), kernel-space (gray), user-space (black). At time 0, the Radar-Sensor Service is operating normally under Linux. (A) Attack commences, (B) Attack becomes fully effective, (C) The VM-Switch actuator activates the FreeBSD alternate virtual machine.

5 Lessons Learned and Future Directions

5.1 Human and Automated Decision Maker

In our demonstration, we relied on a human decision maker to activate the VM-Switch actuator. On one hand, humans have a greater ability to recognize patterns and have access to more situational awareness than can be easily encoded in a computer system. On the other hand, humans are slow relative to the speed that automated decision makers can react. It is important that future work clarifies the roles of automated and human decision makers in such a way that leverages the strengths of each approach.

5.2 False Positives and False Negatives

Though the VTop sensor is indicative, it is subject to false positives (the kernel may consume 100% of the CPU but the Radar-Sensor Service may actually work fine if its throughput is maintained) and false negatives (many attacks will not manifest symptoms observable via VTop).

In general, false negatives are tolerable because we expect other sensors and analyzers to compensate for a particular sensor's blind spots. We expect that false positives will not be problematic for VTop since a cyber operator can use the decision mediator to rapidly investigate a potential fault.

VTop's accuracy is only acceptable because we rely on the assumption that other, more indicative sensors exist. For example, we could employ a cyber sensor that subscribes to the Radar-Sensor Service; when it observes the absence of expected radar data it is strongly indicative that a fault has occurred. Such sensors are also desirable because their accuracy seems more amenable to analysis. Future NCDS must balance the benefits and costs of generic versus specialized sensors and analyzers. Perhaps generic frameworks for developing specialized components can provide the solution.

5.3 Adversary Model

In our demonstration we use a limited adversary model where an adversary used a single denial-of-service attack. During warfare, we expect real adversaries to launch a multitude of attacks targeting all components of the BMD OODA loop. Further, we should also expect non-DOS attacks, such as disinformation attacks and attacks against the Cyber OODA loop.

To combat the wide range of attacks we expect to encounter during warfare it will be important to develop a diverse suite of sensors, analyzers, and actuators that are capable of mitigating all of these types of attacks.

5.4 Statefulness

The VM-Switch actuator is easiest to implement when it is applied to a stateless application. When developing this actuator we assumed that since the Radar-Sensor Service is semantically stateless that its implementation would also be

stateless. This turned out to be a wrong assumption, as the Radar-Sensor Service maintained a small amount of state, which we needed to transfer during a VM-Switch activation. Since it is easier to protect stateless implementations, we recommend that semantically stateless applications should also use stateless implementations.

However, even if a distributed application is completely stateless with respect to application-layer semantics, it does not necessarily mean that it will be stateless on all layers of the network stack. In traversing a modern, secure, enterprise network, a typical data packet will need to pass through multiple network devices, such as firewalls and proxy servers, which will track some sort of state. Therefore, in order to ensure application layer survivability, security restrictions at lower levels of the network stack will need to be relaxed, creating additional attack surface. We seek to further investigate this trade off, in order to optimize cyber survivability.

5.5 Mission-Level Awareness

Based on user feedback obtained from BMD subject-matter experts, we recognize the importance of generating “mission-level awareness.” It is of little value to a commander to know that a service has failed; it is of great value to know that the BMD mission is currently compromised during a missile attack because a critical service has failed. Mission-level awareness can be embedded in the NCDS architecture by implementing analyzers that correlate system faults with mission impact. It remains an open problem though how such analyzers should be implemented.

6 Conclusion

As the DoD increases its reliance on net-centric systems it becomes increasingly important to protect those systems from cyber attack. Towards this end, we have developed the Net-centric Cyber-Decision-Support (NCDS) architecture. Our architecture aims to improve the user’s ability to quickly enact good decisions that will remedy faults at run time. Our approach to survivability stands to advance the state of the practice by leveraging existing dependability approaches in two ways.

First, our approach adds another “layer” of dependability on top of existing approaches. We acknowledge that it is not currently possible to develop flawless complex-system software. However, there are numerous techniques for improving the “quality” of such software. We expect that software developers will use current engineering best practices to develop the highest quality software possible. Much research in computer dependability is dedicated to establishing and improving methods for developing dependable software. Our architecture complements this research by providing a scheme to improve the user’s ability to detect and remedy faults at runtime.

Similarly, our approach adds a layer of collaborative decision-making on top of existing intrusion-detection-and-response (IDR) approaches. Much IDR research

is dedicated to establishing and improving methods for (1) observing computing systems, (2) analyzing observations, and (3) formulating and enacting responses. Rather than developing such a technique, in this research we propose an architecture that integrates existing IDR approaches into a complete system that improves decision making. We also acknowledge that current AI technology is insufficient to completely supplant human decision making. Thus we propose to facilitate collaboration between AI systems and human decision makers to produce higher-quality decisions.

Thus our design of the NCDS architecture represents an interesting point in the space of techniques for achieving cyber survivability; it complements existing approaches for developing dependable software and for detecting and responding to cyber attacks.

Though we believe that the NCDS architecture provides a promising approach to improving cyber survivability, it is not yet mature for production. Several open questions remain before we can endeavor to engineer a production implementation of the NCDS architecture. For example, how do we balance the competing needs to share information and to conserve bandwidth? What are the limits of automated decision making? How should automated and human decision makers interact? Is it possible to implement a trustworthy hypervisor? Is survivability assurable? Though such questions abound, we believe that future work can answer these questions.

Our experience developing the NCDS architecture and illustration-of-concept implementation has proven valuable for the lessons we have learned. We expect that these insights will be valuable to ourselves and to others as we continue to work towards achieving survivability in cyber space.

Acknowledgments

We would like to thank Michael Locasto for contributing to early discussions in the development of our architecture.

Succeeding in such a large-scale operational-like event required a team of 25 people, all of whom we would like to thank. In particular John Urbano was key to the development and integration of the cyber-survivability components within the net-centric framework.

References

1. Alberts, D.S., Garstka, J.J., Stein, F.P.: Network centric warfare: Developing and leveraging information superiority (1998)
2. VMware Technology Network Blog. Introducing VProbes: a stethoscope for your VM (June 2008), <http://blogs.vmware.com/vmtn/2008/06/introducing-vpr.html>
3. Boyd, J.R.: A discourse on winning and losing. Maxwell Air Force Base, AL: Air University. Library Document No. M-U 43947, Briefing slides (1987)

4. Chen, L., Avizienis, A.: N-version programming: A fault-tolerance approach to reliability of software operation. In: International Symposium on Fault-Tolerant Computing, FTCS (1978)
5. Coakley, T.: Decision superiority. *Air & Space Power Journal* (May 2001)
6. Crosby, S.A., Wallach, D.S.: Denial of service via algorithmic complexity attacks. In: USENIX Security Symposium (2003)
7. National Vulnerability Database. Vulnerability summary for CVE-2003-0244, <http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2003-0244>
8. Endsley, M.R.: Toward a theory of situation awareness in dynamic systems. *Human Factors* 37(1), 32–64 (1995)
9. Keromytis, A.D.: Characterizing self-healing software systems. In: Proceedings of the 4th International Conference on Mathematical Methods, Models and Architectures for Computer Networks Security (MMM-ACNS) (2007)
10. MIT Lincoln Laboratory. Air and missile defense technology principal accomplishments (2008), <http://www.ll.mit.edu/mission/airmissile/airmissileaccomplishments.html>
11. Lee, D., Lee, J., Lee, K.H.: A decision-support system to improve damage survivability of submarine. In: Hendtlass, T., Ali, M. (eds.) IEA/AIE 2002. LNCS (LNAI), vol. 2358, pp. 61–78. Springer, Heidelberg (2002)
12. Lipson, H.F., Fisher, D.A.: Survivability—a new technical and business perspective on security. In: NSPW 1999: Proceedings of the 1999 workshop on New security paradigms, pp. 33–39. ACM, New York (2000)
13. United States Department of Defense. The implementation of network-centric warfare. United States Government Printing Office (January 2005)
14. United States Department of Defense. FY2009–2034 Unmanned systems integrated roadmap (2009), <http://www.acq.osd.mil/uas/>
15. United States Joint Chiefs of Staff. Joint publication 3-13: Information operations (February 2006), <http://www.dtic.mil/doctrine>
16. United States National Institute of Standards (NIST). Guide to intrusion detection and prevention systems (IDPS) (Special publication 800-94) (February 2007)
17. Schwaegerl, C., Seifert, O., Buschmann, R., Dellwing, H., Geretshuber, S., Leick, C.: Increase of power system survivability with the decision support tool CRIPS based on network planning and simulation program PSS@SINCAL. In: Setola, R., Geretshuber, S. (eds.) CRITIS 2008. LNCS, vol. 5508, pp. 119–130. Springer, Heidelberg (2009)
18. Sidiroglou, S., Locasto, M.E., Boyd, S.W., Keromytis, A.D.: Building a reactive immune system for software services. In: In Proceedings of the USENIX Annual Technical Conference, pp. 149–161 (2004)
19. United States. Information operations [electronic resource]. U.S. Air Force, Washington, D.C (2005)
20. Weimer, F.: Algorithmic complexity attacks and the linux networking code (May 2003), <http://www.enyo.de/fw/security/notes/linux-dst-cache-dos.html>

A Safety Case Approach to Assuring Configurable Architectures of Safety-Critical Product Lines

Ibrahim Habli and Tim Kelly

Department of Computer Science, University of York, United Kingdom
{Ibrahim.Habli, Tim.Kelly}@cs.york.ac.uk

Abstract. Companies are increasingly adopting a product-line approach to the development of safety-critical systems. A product line offers large-scale reuse by exploiting common features and assets shared by systems within a specific domain. In this paper, we discuss the challenges of justifying the safety of architectural configurations and variation when developing product-line safety cases. We then address these challenges by defining an approach to developing product-line safety cases using the patterns and modular extensions of the Goal Structuring Notation (GSN). In this approach, we use the GSN patterns extension for explicitly capturing safety case variations and tracing these variations to their extrinsic source in the architectural model. Further, we use the GSN modular extension to organise the safety case into core and variable argument modules which are loosely coupled by means of argument contracts. We demonstrate this approach in a case study based on a product line of aero-engine control systems.

Keywords: Safety Cases, Architectures, Product Lines, Variation Management.

1 Introduction

To reduce the engineering costs of safety-critical systems, companies are increasingly adopting a product-line approach which offers large-scale reuse by exploiting common features and assets shared by systems within a specific domain. In particular, the safety case in a safety-critical product line is a valuable asset which should be systematically documented, reused and maintained. Otherwise, the value of a safety-critical product line can be easily undermined if the safety case is developed from scratch, or in an ad-hoc manner, for each product within the product line. Given that products in a product line share most of their functional features, components, failure modes and risk mitigation measures, it is reasonable to expect that these products also share strategies which can be used to argue why they are acceptably safe to operate within certain environments. For example, if products derived from a product-line share a set of functional configurations, which pose similar risks managed using common risk-mitigation measures, it would be sensible to expect that the safety case for these products share a set of core (i.e. common) argumentation strategies too.

However, like most reusable product-line assets, the challenges do not lie simply in exploiting and managing commonalities. Rather, the key challenges often lie in the

management of the way in which assets may vary, according to predefined architectural constraints. In other words, it is important to identify and manage both the *common* and the *variable* structures of the product-line safety case and the ways in which these structures may be reused and composed in order to derive a compelling, comprehensible and traceable safety case for each individual product.

In this paper, we describe and evaluate a safety case approach to justifying the architectures of safety-critical product lines. We start by introducing product-lines. We then discuss the challenges of managing variations within product-line safety cases and how they can correspond to architectural configurations. We then present an approach to developing product-line safety cases using the patterns and modular extensions of the Goal Structuring Notation (GSN) [3]. Finally, this approach is evaluated in an aerospace case study.

2 Product-Line Engineering

A product line comprises a configurable architecture and a set of reusable core assets. Products can be derived from the product-line architecture and core assets based on a predefined process that manages and controls permitted configurations and *variations*. These variations in a product line are not an indication that the development is unstable. Instead, they indicate that differences between products are identified, analysed and controlled. Product-line development is an integrated approach in that any development or assessment artefacts, particularly early-lifecycle artefacts such as requirements and analysis models, can be reused as long as they adhere to the context, architectural constraints and variation rules defined in the product line. To reap the benefits of product lines, the development should be carried out according to defined processes and within certain technical and business constraints. The following definition by the Software Engineering Institute (SEI) provides a comprehensive description of product lines [1]: “A *software product line* is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way”. What distinguishes the SEI’s definition from many other accounts of product lines is that it explicitly addresses the need to develop the product-line core assets and derive new products “*in a prescribed way*”, i.e. according to defined processes. There are two essential processes in any product line, namely:

- Product-line establishment process (domain engineering)
- Product derivation process (application engineering)

In the domain engineering process, the product-line scope is defined, permitted variations are identified and the reference architecture and core assets are developed [2]. Subsequently, in the product derivation process, products within the scope of the product line are developed from the reusable assets, according to permitted variations.

3 Variation in Product-Line Safety Cases

The safety case of a product line is not immune from the impact of environmental and system variations. Ignoring or underestimating the impact of these variations may

seriously weaken confidence in the product-line safety case. For example, any mismatches between the product line’s permitted architectural options and the assumed architectural options considered in the safety case can invalidate certain safety case claims and evidence. Fig. 2 shows an example safety argument, represented in GSN, for a safety-critical system. GSN represents safety arguments in terms of basic elements such as goals, solutions, and strategies (Fig. 1). Arguments are created in GSN by linking these elements using two main relationships, ‘supported by’ and ‘in context of’ to form a goal structure. The principal purpose of any goal structure is to show how goals (claims about the system) are successively broken down into sub-goals until a point is reached where claims can be supported by direct reference to available evidence (solutions). As part of this decomposition, using the GSN it is also possible to make clear the argument strategies adopted (e.g. adopting a quantitative or qualitative approach), the rationale for the approach and the context in which goals are stated (e.g. the system scope or operational role). GSN has been adopted by a growing number of companies within safety-critical industries (such as aerospace, railways and defence) for the presentation of safety arguments within safety cases. The key benefit experienced by those companies adopting GSN is that it improves the comprehension of the safety argument amongst all of the key project stakeholders (i.e. system developers, safety engineers, independent assessors and certification authorities). In turn, this has improved the quality of the debate and discussion amongst the stakeholders and has reduced the time taken to reach agreement on the argument approaches being adopted.

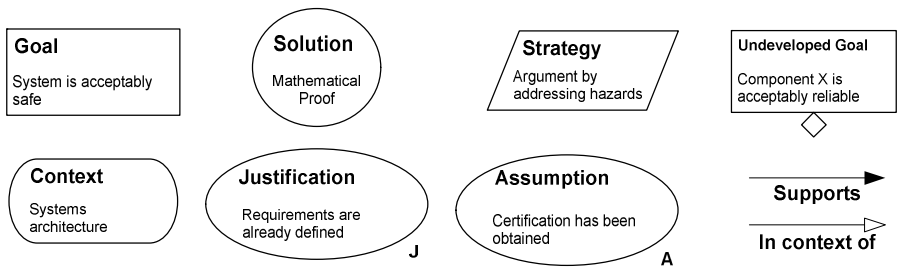


Fig. 1. Core Elements of GSN

The style of the reasoning in the argument in Fig. 2 is hazard/risk directed. Briefly, the top-level claim, that the system is safe to operate within the specified environment (*SysSafe*), is substantiated by arguing that the risk posed by the identified hazards is acceptable. The top-level claim is made in the context of some definition of the system and its environment (*CSys* and *CEnvironment*). Next, the argument considers the acceptability of the risk posed by each identified hazard in the context of specific risk tolerability criteria (*CTolerabilityCri*). Finally, the risk acceptability of each hazard is supported by appealing to the deployment of sufficient risk-mitigation measures (*CRiskMitig*) which are implemented by a number of components and interactions (*CompSafe1*, *CompSafe2*, *CompSafe3* and *ComInter*).

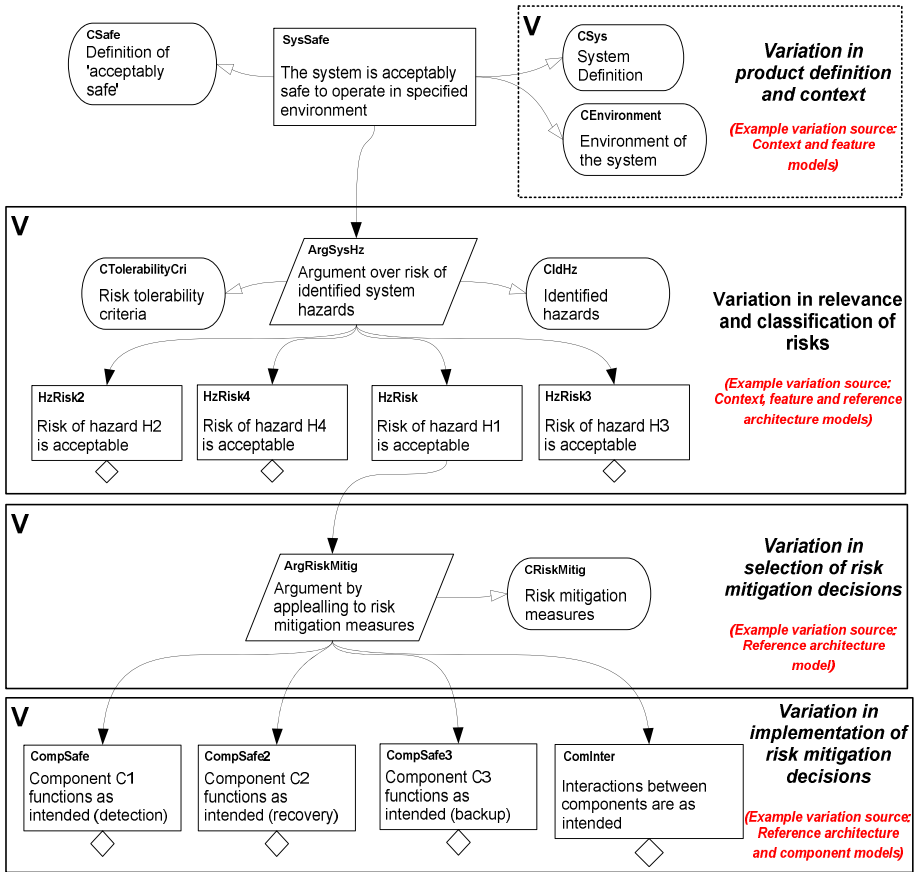


Fig. 2. Impact of Product-Line Variation on the Safety Argument

Product-line variation can potentially affect the argument in Fig. 2 in any of the following ways:

- Context and system definition:** In a product line, the context and feature models define how products, derived from the product line, vary from one another. To this end, the safety argument context elements (*CSys* and *CEnvironment*) have to vary to correspond accurately to the way in which each product is configured, based on the context and feature models.
- Hazard identification and risk assessment:** Depending on the features selected and the environmental conditions assumed for each derived product, the way in which the argument considers the risk posed by hazards could vary (*ArgSysHz*). Not all hazards may be relevant to all product configurations. Also, the risk assessment results for each applicable hazard may vary due to some variable external or system features. Further, the risk tolerability criteria may vary across products if these products are deployed in different environments with different certification requirements, e.g. civil vs. military applications.

- **Risk mitigation measures:** The product-line reference architecture may offer a number of options for mitigating certain risks (*CRiskMitig*). Not all these options may be selected for each product. Consequently, the product-line safety argument needs to accommodate variation concerning how each derived product may mitigate its associated risks.
- **Implementation of risk mitigation measures:** Two derived products may share the same risk-mitigation strategy, but may vary in how they implement such a strategy. For example, two products may adopt a strategy for fault-detection by means of monitoring. However, the product-line reference architecture may offer two or more alternatives for implementing a monitor, e.g. either in software or in hardware. The product-line safety case should therefore explicitly address possible variation in how risk mitigation measures may be implemented.

The above example safety case variations show the importance of developing the product-line safety case in such a way that permitted safety case variations are explicitly defined and traced to other variations in the environment and architectural configuration of a derived product.

4 Managing Product-Line Safety Cases Using GSN

In this paper, we address the challenges presented in the previous section by defining an approach to developing product-line safety cases using the patterns and modular extensions of GSN. The patterns and modular extensions of GSN are introduced in the next two sections, followed by a detailed description and analysis of how they can be used to create and manage product-line safety cases.

4.1 GSN Patterns Extension

Based in part on the principles of the work on design patterns by Christopher Alexander [9] and the ‘Gang of Four’ (Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides) [10], the concept of a safety case pattern was developed as “*a means of documenting and reusing successful safety argument structures*” [3]. A safety case pattern captures argument structures observed to be common in forming the backbone of certain safety cases in a particular domain or across different domains. To create safety case patterns, the core of GSN was extended by adding the following types of abstraction (Fig. 3):

- **Structural Abstraction** – supporting generalised n-ary, optional and alternative relationships between GSN elements
- **Entity Abstraction** – supporting generalisation/specialisation of GSN elements

Safety case patterns are created in GSN by abstracting the details of commonly-used safety argument structures. These patterns describe a successful and a proven style of argumentation rather than a concrete argument for a particular system.

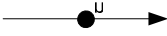
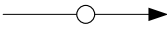




Multiplicity Extensions	
	A solid ball is the symbol for many (meaning zero or more). The label next to the ball indicates the cardinality of the relationship
	A hollow ball indicates "optional" (meaning zero or one)
	A line without multiplicity symbols indicates a one to one relationship (as in conventional GSN)
Optionality Extension	
	This symbol is defined for use over the existing relation types. Choice can be used to denote possible alternatives in satisfying a relationship. It can represent a 1-of-n and m-of-n selection.
Entity Abstraction Extensions	
 Uninstantiated Entity	Entity remains to be instantiated i.e. at some later stage the 'abstract' entity needs to be replaced (instantiated) with a more concrete instance.
 Undeveloped Entity	This placeholder denotes that the attached entity requires further development, i.e. at some later stage the entity needs to be (hierarchically) decomposed and further support by sub-entities.

Fig. 3. GSN Pattern Notation – Entity Abstraction Extensions [3]

4.2 GSN Modular Extension

To support the certification of highly-integrated, modular and reconfigurable systems such as Integrated Modular Avionics (IMA), the core of GSN was extended with modular features [4]. These features support the development of *modular and compositional safety cases*. One fundamental objective of modular and compositional safety cases is to reduce the amount of effort needed for the reassessment of a safety case after system changes or reconfiguration. Rather than considering the safety case as a *single monolithic* structure, it can be viewed as a set of well-defined and scoped modules, the composition of which defines the system safety case. Each argument module is specified by an interface, comprising [4]:

1. Goals addressed by the module
2. Evidence presented within the module
3. Context defined within the module
4. Arguments requiring support from other modules

Inter-module dependencies:

5. Reliance on goals addressed in other modules
6. Reliance on evidence presented within other modules
7. Reliance on context defined within other modules

Elements 5, 6 and 7 are called *Away Goals*, *Away Context* and *Away Solutions* (Fig. 4). In particular, an *Away Goal* is a goal reference which is used to support, or provide contextual backing for, an argument presented in one module. However, the argument supporting that goal is presented in another module (hence creating interdependencies between the safety case modules).

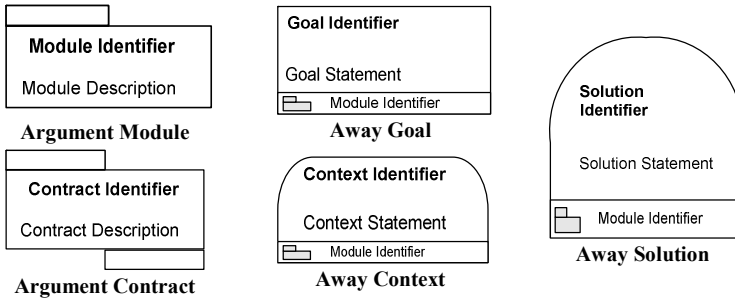


Fig. 4. GSN Modular Notation [4] [5]

Further, in order to promote loose coupling and therefore minimise the impact of change between interrelated safety case modules, the concept of an argument *contract* was created in [4], and later refined in [5] [7] (Fig. 4). Argument contracts preserve the overall integrity of the modular safety case when the internal details of one or more argument modules are modified. This is mainly because these contracts are specified using the interfaces of the interrelated argument modules rather than using the internal details of these modules (hence protecting interdependent modules from changes to the internal details of the arguments contained in these modules). Essentially, a safety case contract captures a ‘rely-guarantee’ relationship between two argument modules. In a contract, items 4 to 7 in a module interface define the ‘rely’ conditions whereas item 1 defines the guarantee conditions (items 2 and 3 should hold during the composition of the two or more argument modules).

5 Capturing Safety Case Variations Using the GSN Patterns Extension

In this section, we show how the GSN patterns extension can be used to capture variation in the product-line safety case. A product-line safety case comprises reusable argument structures and evidence used as the basis for the definition of the safety case for each product derived from the product-line assets. Some of these argument structures and evidence are shared between all product safety cases, and therefore are *core*, while others differ from one product safety case to another, and therefore are *variable*. Generally, there are two types of product-line safety case variation: *intrinsic* and *extrinsic variation*.

Intrinsic variation exists whenever there is more than one argumentation style to support the safety claims of a *particular* product-line instance. *Extrinsic* variation, in contrast, is more peculiar to product-line safety cases. The source of this type of variation is not the product-line safety case itself but rather the reusable assets referenced in the safety case from product-line models such as the feature and reference architectural models. This is because many of these assets are expected to vary in how they are developed, configured and composed. This variation may change the contribution of these assets to safety and therefore may change the way in which the safety of the system is justified in the safety case. To this end, it is important that extrinsic

variation in a product-line safety case be explicitly linked to, and constrained by, its source, be it a variation in the context model, the feature model or the reference architecture model.

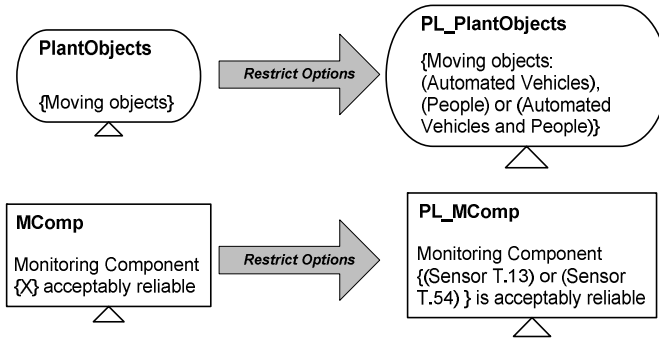


Fig. 5. Defining Safety Case Variations using Entity Abstraction Extensions

As discussed in Section 4.1, there are two types of abstraction supported in the GSN patterns extension: *Entity Abstraction* (supporting generalisation/specialisation) and *Structural Abstraction* (supporting optionality and multiplicity). Here, we use *Entity Abstraction* to capture and restrict variation in GSN elements (goals, strategies, context, assumptions, justifications and evidence). On the other hand, we use *Structural Abstraction* to capture and restrict variation in the relationships between the GSN elements (*‘supported by’* and *‘in the context of’* relationships). The left hand side of Fig. 5 shows example un-instantiated GSN elements (types of *Entity Abstraction*). The items in the curly brackets represent types of un-instantiated information (*‘Moving objects’*, *‘X’* and *‘Coverage Analysis Data’*). These items need to be instantiated before they can be used as part of a concrete safety argument. When representing extrinsic variation in the product-line safety case, the un-instantiated GSN elements should be explicitly associated with their source of variation. As shown on right hand side of Fig. 5, *‘Moving objects’* are limited to *‘Automated Vehicles’*, *‘People’* or *‘Automated Vehicles and People’*, which are objects defined in the product-line context model. In other words, these are the moving objects which are assumed to be within the scope of the product-line and, as such, the safety argument is only valid in the context of one of these sets of moving objects. Similarly for the *‘Monitoring Component {X}’* in the *‘MComp’* Goal, two variants of this component exist in the *‘PL_MComp’* Goal: *‘Sensor T.13’* and *‘Sensor T.54’*. These variants are alternative design components offered by the reference architecture. Variations in the safety case do not only affect the GSN elements, but they also affect the way in which these elements are connected. To capture and restrict variation in the relationships between the GSN elements of a product-line safety case, we use the *Multiplicity* and *Optionality* symbols which are part of the GSN patterns extension (*Structural Abstraction*).

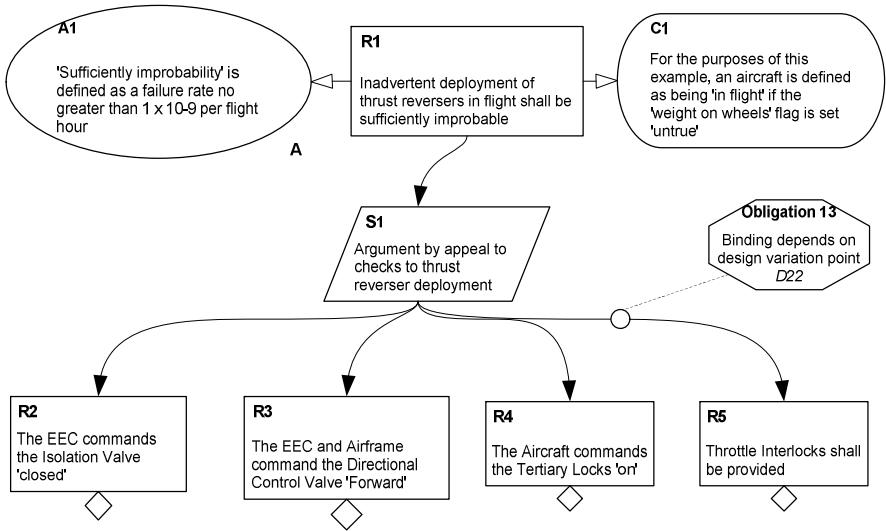


Fig. 6. Thrust Reverse Deployment Protection Argument (adapted from [6])

Fig. 6 illustrates how variation in the way in which GSN elements may be connected can be captured and traced to the product-line models. For each multiplicity and optionality, we attach a GSN element called ‘Obligation’ (octagon symbol), which describes the basis on which options and alternatives may be selected. More specifically, each obligation element that is based on extrinsic variations should be traced to one or more product-line variations in the context, feature or reference architectural models. The argument in Fig. 6 is an adaptation of the thrust reverse deployment defined in [6]. In order to prevent an inadvertent deployment of thrust reversers in flight, three core checks should be performed: ‘The EEC commands the Isolation Valve closed’ (R2), ‘The EEC and Airframe command the Directional Control Valve Forward’ (R3) and ‘The Aircraft commands the Tertiary Locks on’ (R4). The use of throttle interlocks on certain engines is optional (labelled in Fig. 6 as design variation point ‘D22’). In other words, the use of throttle interlocks is only required by some airframe manufacturers. As such, ‘Obligation 13’ is used in Fig. 6 to indicate that ‘R5’ is optional and its selection depends on the instantiation of the design variation point ‘D22’ (the design assumption here is that, regardless of whether ‘D22’ is instantiated or not, as a minimum requirement, the overall failure rate is not greater than 10^{-9} per flight hour).

6 Composing Product-Line Safety Cases Using the GSN Modular Extension

The product-line safety case should be highly reconfigurable in order to support the derivation of a safety case for each product developed from the product-line assets. In the previous section, we described how to embed variations into the product-line safety case using the GSN patterns extension. These variations need to be instantiated

for each derived *product* safety case. However, a key constraint is to instantiate these variations in a way that minimises the effort needed to assess the suitability of a derived safety case used for the assurance of a particular product. In this section, we propose that the effort needed for the assessment of each product safety case could be reduced by the adoption of the concept of modular and compositional safety cases. Modularity is a proven design technique for improving the flexibility of software systems, particularly through promoting loose coupling and high cohesion between modules interacting through a stable set of interfaces. These interfaces hide the details of each module, exposing only information required for module integration, hence reducing unnecessary dependencies between interacting modules.

In this section, we use the GSN modular extension, reviewed in Section 4.2, to structure the safety case into core and variable *argument modules*. These argument modules are defined and composed in a way that protects core argument modules from the permitted variation in the optional argument modules by means of predefined argument *contracts*. In order to improve the process of deriving a safety case instance from the product-line safety case, we establish a clear and traceable mapping between the structures of the product-line safety argument and the structures of the product-line feature and reference architecture models. Further, it is important to define a product-line safety case in such a way that variations, e.g. options and choices, can be added or removed with little impact on the overall structure of the product-line safety case and its instances. This can be achieved by defining the safety case in terms of loosely-coupled argument modules. One mechanism for defining loosely-coupled argument modules is to reduce '*hardwired*' dependencies between the argument modules by means of predefined argument *contracts* [5]. These argument *contracts* can serve as a mediator or a proxy between interrelated argument modules, isolating the way in which one argument module is '*supported*', or '*contextually-backed*', by another argument module. More specifically, argument contracts can be used to contain the impact of variation in one argument module and prevent it from propagating to other argument modules. The way in which argument contracts can be used to contain the impact of product-line variations within a safety case is illustrated in Fig. 7. The top-left hand side of Fig. 7 shows an argument structure in which the argument module '*Function X*' is supported by either the argument module '*Redundancy*' or the argument module '*Monitoring*', i.e. variation in the form of alternativity. Here, because these two argument modules are *directly* connected to the argument module '*Function X*', a change to these modules may propagate to the argument module '*Function X*' and vice versa. The top right hand side of Fig. 7 shows how the insertion of an argument contract between the argument module '*Function X*' and the argument modules '*Redundancy*' and '*Monitoring*' creates a 'buffer zone' that masks the way in which the argument module '*Function X*' is supported. For example, if a new optional argument module is later added to the product-line safety case to support the argument module '*Function X*', any required change should be contained within the argument contract and should not propagate to the argument module '*Function X*'. Fig. 7 also shows internal details of the argument contract, showing how the public goal '*SysDesign*', which is part of the public interface of the argument module '*Function X*', is supported by a strategy which is based on either a claim of adequate redundancy, provided by the argument module '*Redundancy*', or a claim of

adequate monitoring, provided by the argument module ‘*Monitoring*’. In other words, within the scope of the argument module ‘*Function X*’, the actual way in which the goal ‘*SysDesign*’ is supported is unknown but entrusted to the argument contract, i.e. breaking any direct coupling between the argument module ‘*Function X*’ and the argument modules ‘*Redundancy*’ and ‘*Monitoring*’.

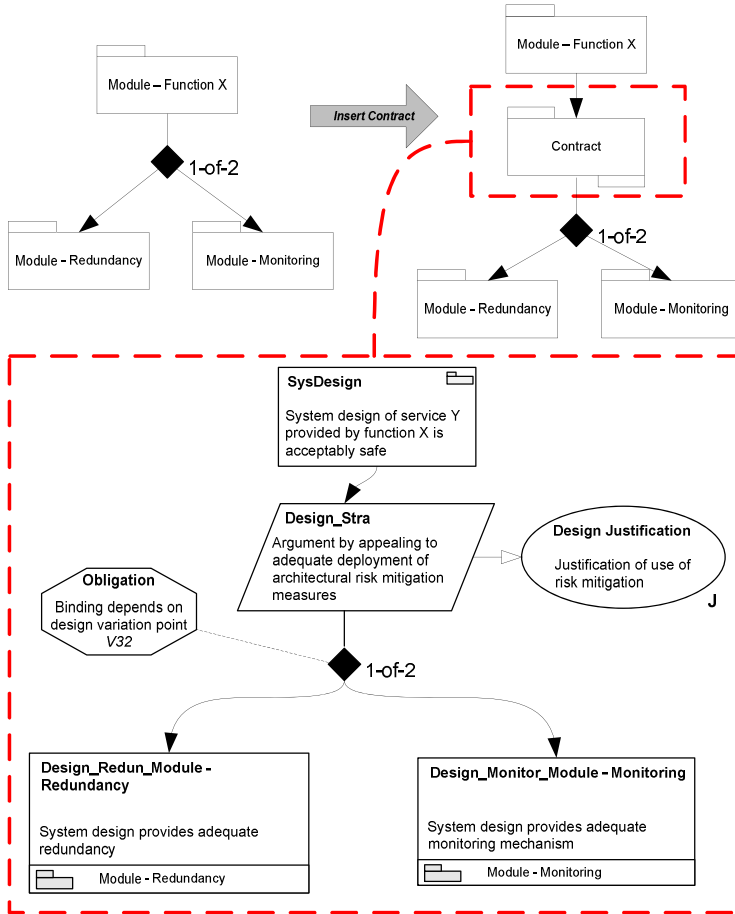


Fig. 7. Variations within an Argument Contract

In short, modularity in the product-line safety case can support the encapsulation of the arguments behind product-line functions and components in reusable and self-contained modules. Equally importantly, dependencies between these modules should be minimised in order to contain potential ripple-effects resulting from changes in the options and choices provided by the product line.

7 Case Study – Assuring Safety of Aero-Engine Sensors

We have demonstrated how a product-line safety case can be constructed using the patterns and modular extensions of GSN by means of case studies from the aerospace and automotive domains. The case study presented in this section is partly based on an aero-engine control system described in [8].

7.1 System Overview

Modern aircraft gas turbine engines are controlled by a Full Authority Digital Electronics Control (FADEC) system. This control system is a high-integrity computer system that controls and monitors the operation of the engine. Within aero-engine product lines, not only may these product lines vary at the engine level (e.g. optional/alternative physical and interface characteristics), but also they can vary at the control system level (e.g. optional/alternative functional and technological characteristics). The control system variants can be considered as part of a product line embedded within the larger engine product line. In this case study, we focus on the commonalities and variabilities within an engine control system product line.

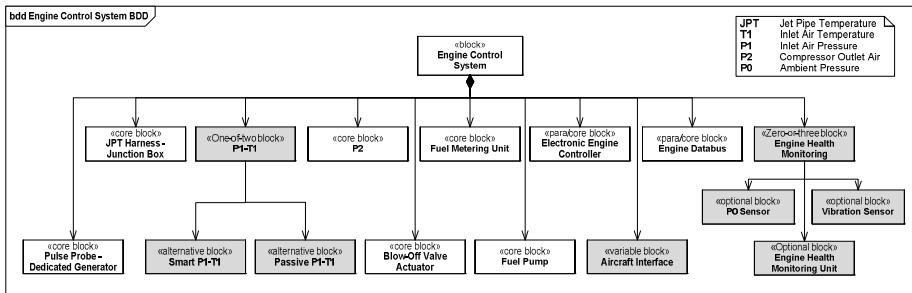


Fig. 8. Block Definition Diagram

A depiction of the reference architecture of the control system product line is shown in Fig. 8 and Fig. 9 (specified in the Systems Modelling Language (SysML) [11]). Here we focus on three different types of variations within this architecture. Firstly, the reference architecture provides two alternative means for measuring the engine’s inlet air temperature (T1) and inlet air pressure (P1) using either passive or smart sensors. The key differences between these two types of sensor are as follows. The reliability of T1 and P1 measurements is higher when smart sensor are used. Also, signal conditioning, signal selection, fault detection and fault accommodation can be carried out locally by the smart sensors T1 and P1 (as opposed to being fully performed by the Electronic Engine Controller when passive sensors are used). Further, smart sensors can communicate with the Electronic Engine Controller via the Engine Databus, and not through a dedicated pipe, and therefore can simplify “*wiring and connections, piping and reduce weight*” [8]. The second key architectural variation lies in the Aircraft Interface block. This variation supports the realisation of various interface requirements requested by different customers (e.g. special interface

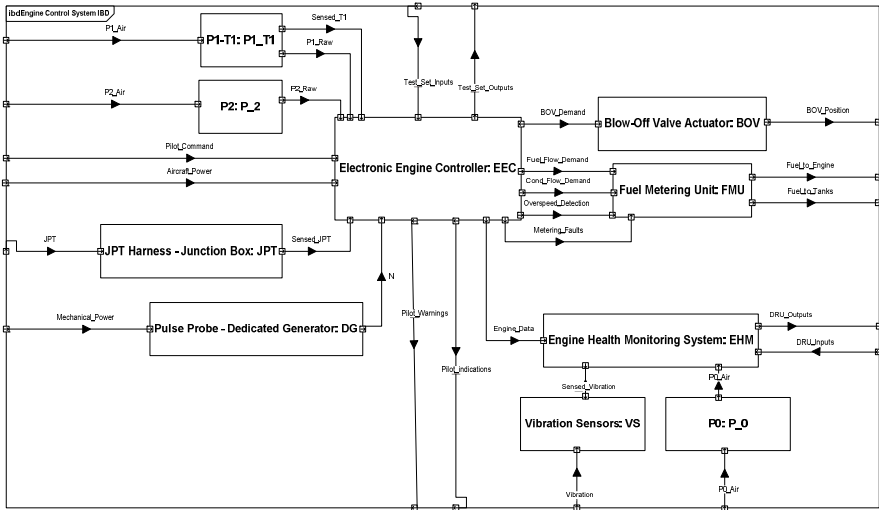


Fig. 9. Internal Block Diagram

requirements by aircraft manufacturers). Finally, the reference architecture offers engine health monitoring as an optional block. This block comprises an Engine Health Monitoring Unit and P0 (ambient pressure) and vibration sensors.

7.2 Safety Case

In this case study, we developed a safety case which considered six different hazards. In this paper, due to page constraint, we only consider the argument over the risk of hazardous overspeed. Fig. 10 shows the argument over the mitigation of the faults contributing to hazardous overspeed, e.g. faults in fuel flow, speed measurements and T1/P1 measurements. In particular, the ‘T1/P1’ argument module, referenced in the above argument concerning the contribution of the T1/P1 measurements, requires further instantiation as it embodies variations which are associated with the ability to choose between the deployment of either smart or passive sensors.

The argument within the ‘T1/P1’ module is shown in Fig. 11. In this argument, the top-level claim is made in the context of either passive or smart sensors. This choice is captured using the GSN choice pattern symbol. This choice is associated with ‘*Obligation 2*’ which references design variation concerning the selection of either passive or smart sensors in the reference architecture. Here, regardless of the selected design choice, the reasoning strategy is common – arguing over the T1/P1 reading, conditioning and transmission. The claims concerning the T1/P1 reading, conditioning and transmission are supported by three argument contracts, each of which requires instantiation based on the T1/P1 design choice. The importance of these argument contracts is that they defer the instantiation of variations associated with the T1/P1 design choice and therefore preserve the overall integrity of the T1/P1 argument, i.e. localising the impact of the T1/P1 design choice by hiding this impact behind the contracts.

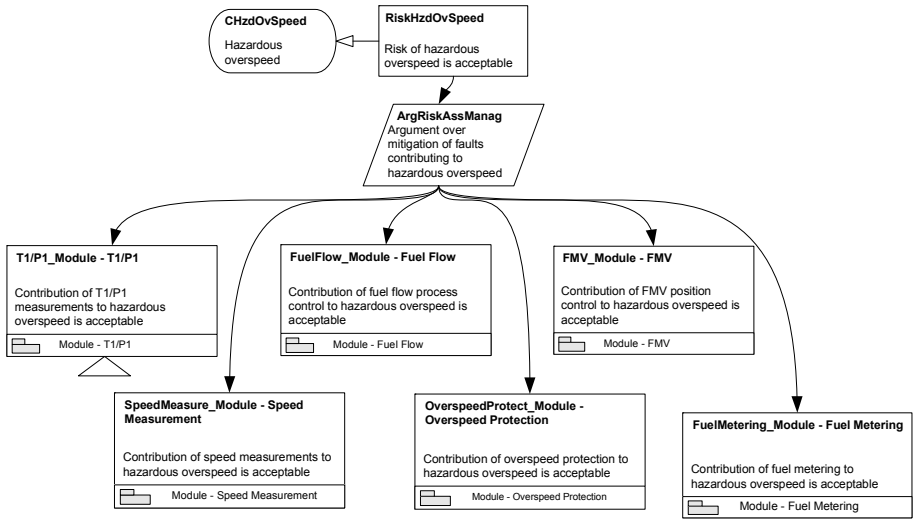


Fig. 10. Argument Module – Hzd Overspeed

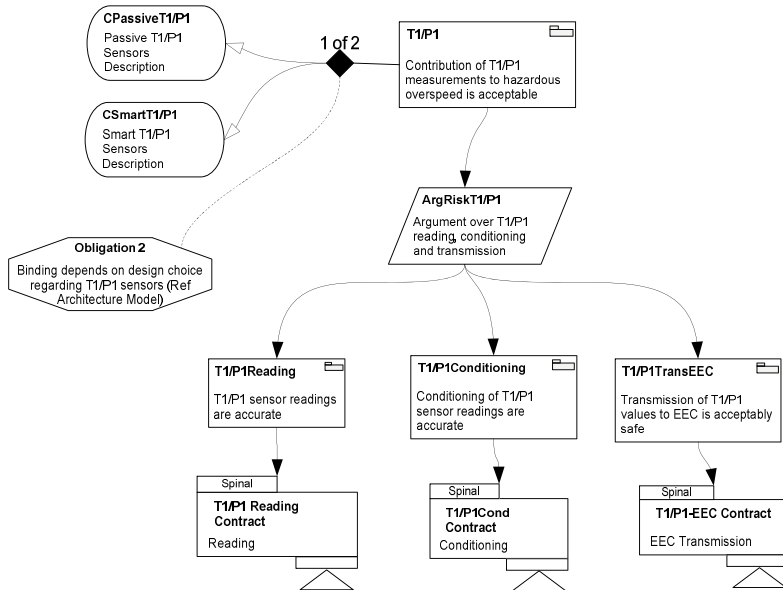


Fig. 11. Argument Module - T1/P1

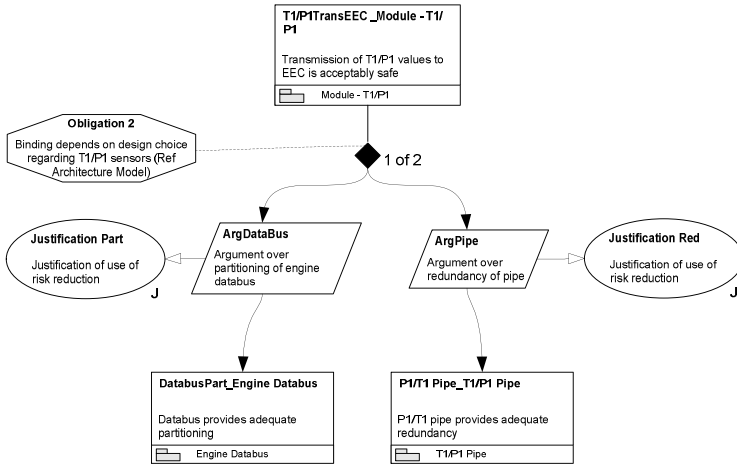


Fig. 12. T1/P1-EEC Contract

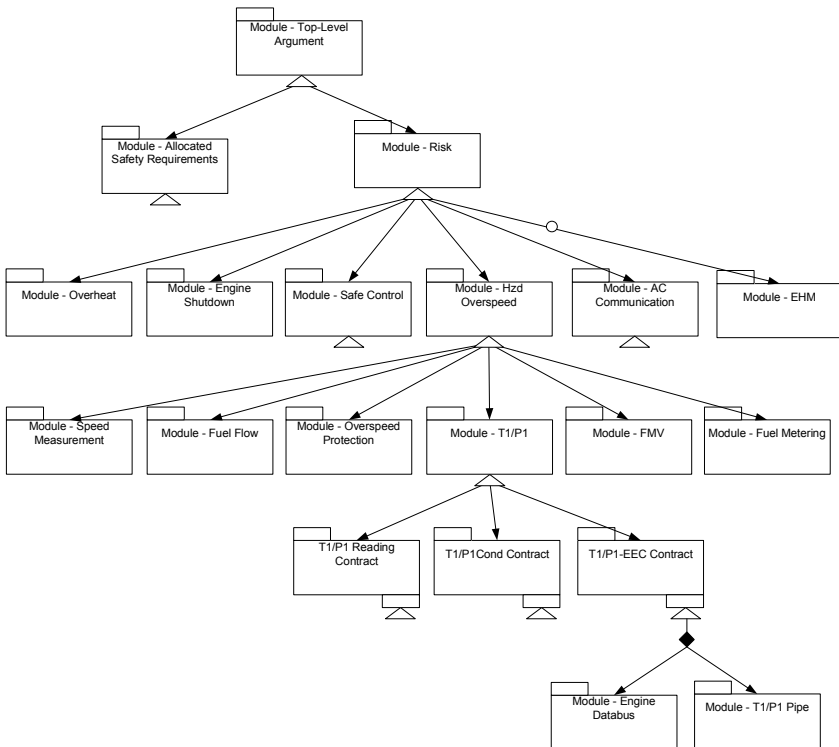


Fig. 13. Safety Case - Module Viewpoint

Fig. 12 shows the argument presented within the ‘*T1/P1-EEC*’ contract. The claim that the transmission of T1/P1 values to EEC is acceptably safe is either supported by arguing over Engine Databus partitioning if smart sensors are chosen or arguing over pipe redundancy if passive sensors are chosen. The choice is made based on the instantiated design variation in the reference architecture regarding the T1/P1 sensor alternatives. Subsequently, depending on the T1/P1 sensor choice, the instantiated contract relies either on the ‘*Engine Databus*’ argument module or the ‘*T1/P1 Pipe*’ argument module to fulfill the guarantee in supporting the claim that the transmission of T1/P1 values to EEC is acceptably safe (‘*T1/P1TransEEC*’).

Finally, in order to show a modular viewpoint of the safety case developed in this case study, we have generated the safety case model shown in Fig. 13. This viewpoint represents a high-level depiction of the top-level safety case in terms of the safety case’s argument modules and contracts. This viewpoint also shows the distinction between *core* argument modules such as the ‘*Engine Shutdown*’ and ‘*Fuel Metering*’ modules and *variable* argument modules such as the ‘*AC Communication*’ and ‘*T1/P1*’ modules (variability indicated using the un-instantiation symbol). Further, this viewpoint reveals some ‘structural stability’ in the way in which the safety case is organised where the higher-level modules address individual hazards and lower-level modules address how the architectural blocks and properties contribute to these hazards. Also, at these two levels, the impact of product-line variation is contained within the argument modules and contracts.

8 Related Work

Safety standards acknowledge the economic need to employ previously developed systems, functions and components [12] [13] [14] [15]. In civil aerospace for example, systems may be reused across different aircraft types, without the need for additional assessment, provided that evidence of similar design, installation, application and operation can be produced [15]. Otherwise, the safety assessment process should be performed to examine the impact of the reusable systems on the aircraft functions. Also in civil aerospace, particularly for airborne software, the American Federal Aviation Administration (FAA) created an Advisory Circular (AC), offering means to satisfy the requirements of the aerospace software guidance DO-178B regarding the use of reusable software components [16].

Generally, most standards place additional constraints on the way in which reusable components are developed, verified, integrated, and maintained. The goal of such constraints is to ensure that the safety of the overall system is not compromised as a result of incorporating reusable components. For example, the flexibility developed into reusable software components may result in unreachable code, and therefore, some standards require that system developers demonstrate that the risk of leaving unreachable code is less than the risk of removing it [13]. In extreme cases where a reusable component was not developed to the safety integrity level of a new system, reverse engineering and the application of more rigorous techniques may be required (in addition to the generation of a new safety argument). In short, although most safety standards do not address product-lines explicitly, they take into account the need to employ previously developed artefacts. They often require additional activities to assess the impact of a reusable component or function on the safety of the overall system.

Research in the field of product-line safety has focused on adapting traditional safety analysis techniques, such as Fault Tree Analysis (FTA) and Failure Modes and Effects Analysis (FMEA), to suit product-line processes. The majority of this work has been produced within the Laboratory for Software Safety at Iowa State University. Most noticeable is the extension of Software FTA (SFTA) to address the impact of product-line variation on safety analysis [17] [18] [19] [20]. This approach is based on a technique for the development of a product-line SFTA in the domain engineering phase and a pruning (or trimming) technique for reusing this SFTA for the analysis of new product-line instances. The approach offers a systematic approach to treating SFTA results as a reusable product-line asset. In particular, the ability to partially automate the pruning of a product-line SFTA, supported by domain expert reviews, should improve confidence in the analysis outcomes. The approach was later extended to demonstrate how to integrate product-line SFTA with the product-line requirements [21]. Further, it was used to integrate the results of the product-line SFTA with state-based models [21]. Based on this integration, reusable test scenarios can be generated for examining the validity of design definitions.

Finally, in a case study examining the impact of product-line variation on the safety assessment data for a Full Authority Digital Engine Control (FADEC), obtained from Rolls-Royce Controls, Stephenson et al created a dependency matrix that traces “*areas of vulnerability*” in the safety assessment data as a consequence of changes “*when moving from one product to another*”. A key conclusion of this case study was that a dependency matrix has the potential to improve traceability between product-line design and safety assessment, particularly in “*enhancing the completeness and robustness of a product line’s safety-related requirements*” [23].

9 Summary and Conclusions

In this paper, we showed how a product-line safety case can be developed in such a way that it can be reused for the assurance of individual product-line instances. We showed that, in addition to the reuse of typical design assets, safety case structures can form a key part of a safety-critical product line’s core assets. We adopted a ‘neat’ way for developing product-line safety cases using the GSN patterns and modular extensions. In particular, we demonstrated how optionalities and multiplicities in GSN patterns can be restricted to address the extrinsic variation in a product-line safety case in a traceable way. This traceability was realised using explicit obligations linking extrinsic safety case variation to their source in the context, feature and architectural models. Further, this paper did not introduce any new symbols which would unnecessarily complicate the development or comprehension of GSN-based safety arguments. This paper only introduced guidance on how the existing GSN (core, patterns and modular GSN) can be used to develop product-line safety cases.

Finally, it is important to note that the approach presented in this paper is suitable for safety-critical product-lines where the impact of variation can be traceably identified, examined and justified. For example, aerospace and automotive applications often satisfy this criterion by being driven by the need to reduce unnecessary complexity in order to simplify the design, and therefore the assessment, of safety-critical functions. To this end, this approach is not suitable for novel and complex applications in poorly understood

domains, as variation would aggravate an already complex problem, an aspect that should be avoided for safety applications. It is also noteworthy that, like in any intellectual design activity, the usage of a particular notation does not mechanically guarantee the production of the intended artefact. Therefore, although the modular and patterns extensions of GSN can help in defining a clear, structured and configurable product-line safety case, engineers need first and foremost to demonstrate a good understanding of design concepts related to abstraction, information hiding and separation of concerns. The product-line safety case approach presented in this paper can only be of value to competent safety case developers with adequate understanding of the product-line scope and domain.

References

1. Clements, P., Northrop, L.: *Software Product Lines: Practices and Patterns*. Addison-Wesley, Reading (2001)
2. Weiss, D.M., Robert, C.T.: *Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley Professional, Reading (1999)
3. Kelly, T.P.: *Arguing Safety – A Systematic Approach to Safety Case Management*. DPhil Thesis, Department of Computer Science, University of York, UK (1998)
4. Bate, I.J., Kelly, T.P.: *Architectural Considerations in the Certification of Modular Systems*. In: Anderson, S., Bologna, S., Felici, M. (eds.) *SAFECOMP 2002*. LNCS, vol. 2434, p. 321. Springer, Heidelberg (2002)
5. Industrial Avionics Working Group (IAWG): *Modular Software Safety Case Process – Part A: Process Definition (October 2007)*, <http://www.assconline.co.uk/>
6. Attwood, K., Kelly, T.P., McDermid, J.A.: *The Use of Satisfaction Arguments for Traceability in Requirements Reuse for System Families*. In: *International Workshop on Requirements Reuse in System Family Engineering (2004)*
7. Fenn, J., Hawkins, R., Kelly, T.P., Williams, P.: *Safety Case Composition Using Contracts – Refinements Based on Feedback from an Industrial Case Study*. In: *15th Safety Critical Systems Symposium (2007)*
8. Dowding, M.: *Maintenance of the Certification Basis for a Distributed Control System – Developing a Safety Case Architecture*. MSc Report, Department of Computer Science, University of York, UK (2002)
9. Alexander, C.: *A Pattern Language: Towns, Buildings, Construction*. OUP, USA (1978)
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading (1995)
11. Object Management Group (OMG): *Systems Modelling Language*. vol. 1.1, OMG (2008)
12. EUROCAE/RTCA: *ED-12B/DO-178B: Software Considerations in Airborne Systems and Equipment Certification*. EUROCAE/RTCA (1994)
13. UK Ministry of Defence (MoD): *00-55 Requirements of Safety Related Software in Defence Equipment. Part 2: Guidance, Issue 2, Defence Standard*, UK Ministry of Defence (1997)
14. International Organization for Standardization (ISO): *ISO26262 Road vehicles – Functional safety. Draft, Baseline 15 (2009)*
15. Society of Automotive Engineers (SAE): *Aerospace Recommended Practice 4754: Certification Considerations for Highly-Integrated or Complex Aircraft Systems*. SAE (November 1996)

16. Federal Aviation Administration (FAA): AC 20-148: Reusable Software Components (December 2004)
17. Dehlinger, J., Lutz, R.: PLFaultCAT: A Product-Line Software Fault Tree Analysis Tool. *Automated Software Engineering* 13(1), 169–193 (2006)
18. Feng, Q., Lutz, R.: Bi-Directional Safety Analysis of Product Lines. *Journal of Systems and Software* 78(2), 111–127 (2005)
19. Dehlinger, J., Lutz, R.: Software Fault Tree Analysis for Product Lines. In: 8th IEEE International Symposium on High Assurance Systems Engineering (HASE 2004), Florida, USA (2004)
20. Dehlinger, J., Lutz, R.: Fault Contribution Trees for Product Families. In: 13th International Symposium on Software Reliability Engineering (2002)
21. Dehlinger, J., Humphrey, M., Suvorov, L., Padmanabahn, P., Lutz, R.: Decimal and PLFaultCAT: From Product-Line Requirements to Product-Line Member Software Fault Trees, Research Demonstration. In: 29th International Conference on Software Engineering (ICSE 2007), Minneapolis (2007)
22. Liu, J., Dehlinger, J., Lutz, R.: Safety Analysis of Software Product Lines Using State-Based Modeling. *Journal of Systems and Software* 80(11), 1879–1892 (2007)
23. Stephenson, Z.R., de Souza, S., McDermid, J.A.: Product Line Analysis and the System Safety Process. In: 22nd International System Safety Conference (2004)

Increasing the Resilience of Critical SCADA Systems Using Peer-to-Peer Overlays

Daniel Germanus, Abdelmajid Khelil, and Neeraj Suri*

DEEDS Group, Computer Science Department, TU Darmstadt, Germany
{germanus,khelil,suri}@cs.tu-darmstadt.de
<http://www.deeds.informatik.tu-darmstadt.de>

Abstract. Supervisory Control and Data Acquisition (SCADA) systems are migrating from isolated to highly-interconnected large scale architectures. In addition, these systems are increasingly composed of standard Internet technologies and use public networks. Hence, while the SCADA functionality has increased, its vulnerability to cyber threats has also risen. These threats often lead to reduced system availability or compromised data integrity, eventually resulting in risks to public safety. Therefore, enhancing the reliability and security of system operation is an urgent need. Peer-to-Peer (P2P) techniques allow the design of self-organizing Internet-scale communication overlay networks. Two inherent resilience mechanisms of P2P networks are path redundancy and data replication. This paper shows how SCADA system's resilience can be improved by using P2P technologies. In particular, the two previously mentioned resilience mechanisms allow circumventing crashed nodes and detecting manipulated control data.

1 Introduction and Contributions

Supervisory Control and Data Acquisition (SCADA) systems form key components for Critical Infrastructure (CI) trustworthy monitoring and control. The ubiquitous communication developments are also leading to highly interconnected CIs, resulting in large scale and heterogeneous SCADA networks [23]. While the Internet scale ranges to 10^9 nodes, the US powergrid is currently comprised of 10^5 nodes and steadily increasing [10]. The transition from local area to wide area SCADA systems also corresponds to an increasing replacement of proprietary and vendor-specific communication protocols by open standards [17] and Commercial-Off-The-Shelf (COTS) protocols mainly based on Internet hardware and the Internet Protocol (IP) suite. Overall, the wide area SCADA systems entail immense heterogeneity in terms of network technologies and node properties. Heterogeneity mainly manifests in the interconnection of legacy and state-of-the-art devices, both varying in terms of their computational capacities.

The growing usage of low cost COTS components comes at the cost of potentially increasing the vulnerability of SCADA systems to node and communication failures and cyber attacks. The crash of SCADA network nodes usually

* Research supported in part by EU INSPIRE, CASED (www.cased.de), and EU CoMiFin.

implies the disturbance of SCADA message flows and consequently may perturb the required trustworthy monitoring and control of the corresponding CI and physical processes. More and more SCADA systems are interconnected through public networks and mainly the Internet. Public networks expose them to cyber threats [7]. Being exposed to cyber threats through the Internet eventually results in data integrity attacks, i.e., the deliberate injection of incorrect data to the SCADA system which may have fatal consequences on the proper operation of CIs. Such SCADA perturbations may result in severe CI failures at the basic level of service disruption to critical impact on public/CI safety. Consequently, SCADA resilience against failures and attacks is essentially needed.

Paper Contributions. To achieve increased SCADA system resilience against cyber threats in large-scale systems, our paper proposes a minimally intrusive and low cost communication overlay onto legacy SCADA systems using Peer-to-Peer (P2P) technologies [3]. In particular, we show that our approach efficiently (i) prevents data loss due to node crashes, and (ii) detects and remedies data integrity attacks. Path redundancy and data replication are two P2P mechanisms that we rely on for this purpose. Path redundancy refers to multiple paths between pairs of peers; data replication implies distributed and redundant data storage across the network.

We propose a middleware-based approach that requires only minimal changes to the existing SCADA software system. Furthermore, our solution is scalable to accommodate ongoing developments of interconnected SCADA systems. In addition, our P2P-based data integrity approach enables avoiding the usage of public key infrastructures. This enables the integration of SCADA nodes with low computational capacities, as the timeliness overhead introduced by cryptographic operations may violate SCADA timeliness requirements. Our simulation results show that the SCADA system remains stable in the presence of the considered perturbations. Our approach is currently evaluated in the INSPIRE EU research project [24] by using both, simulation and testbed experiments.

The paper is structured as follows. In Section 2, the system, data, fault, and attack models as well as the requirements on SCADA protection are introduced. Section 3 presents our solution. The simulation environment and evaluation results are described in Section 4. The related work is presented in Section 5 and the conclusion as well as future work are provided in Section 6.

2 Preliminaries

We now present our system model which describes a large-scale SCADA system along with the data model and a fault/attack model. The system model gives an overview of the node types in the system and how they interact with each other. The data model specifies the different data flows and control loops within the SCADA system. Furthermore, the attack and fault model describes faults and attacks that are addressed by our solution to enhance the overall SCADA system resilience. Finally, requirements for SCADA systems are presented to provide mitigation for the considered faults and attack classes.

2.1 System Model

The SCADA system is a network connecting a large number of sensor/actuator clusters with a small set of central control rooms. A generalized SCADA system topology is illustrated in Figure 1. Following the trend to interconnect CIs, the corresponding autonomous SCADA systems are increasingly interconnected in order to facilitate sharing among the different involved authorities, operators etc. Here, autonomy describes a self-contained operator domain, which is not dependent on data exchange with other operator domains to guarantee proper operation. Figure 1 highlights the interconnection of SCADA systems while showing one representative autonomous SCADA system in detail. In the following paragraphs we describe the common components of SCADA systems.

Sensors and Actuators are the monitoring/response components of the SCADA system. Sensors report measurements such as pressure or temperature. Actuators are the controlling elements and conduct operations such as opening or closing valves. Sensors and actuators offer very limited computational and storage capacities.

Remote Terminal Units (RTU) are located “in the field”, i.e., they are scattered along the CI. Both, sensors and actuators are attached to a designated remote terminal unit (RTU), either wired (e.g., through serial interface) or wireless (e.g., through ZigBee [30]). RTUs communicate with superordinate stations using IP encapsulated SCADA protocols.

Master Terminal Units (MTU) collect sensor data from several RTUs and provides the data to other high-level stations (e.g., human machine interfaces (HMI) to give human users a system overview). MTUs also send actuator commands to the appropriate RTU which executes them.

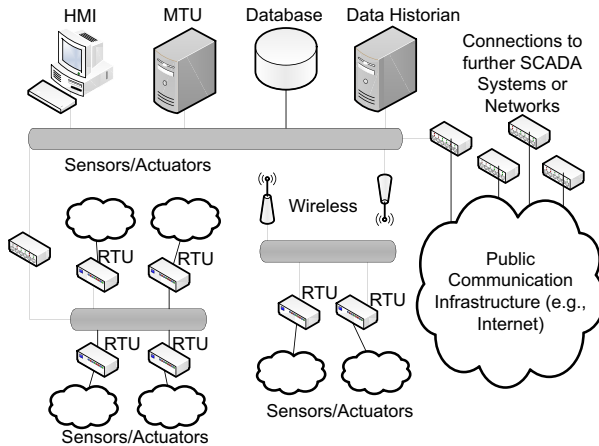


Fig. 1. Basic Components of Interconnected SCADA Systems

Other stations may reside in a SCADA system as well. The two most prominent ones being a data historian, which preserves relevant information over long durations. The second involves Human Machine Interfaces (HMI) and represents stations that provide human SCADA operators insights and controlling capabilities of the SCADA system.

2.2 Data Model

The communication between SCADA nodes is usually message-based. In each autonomous SCADA system, two message flow directions exist: (i) Upward messages are sent on behalf of RTUs and contain raw or aggregated sensor values, and (ii) downward messages are sent by high-level stations through an MTU to specific RTUs. Examples for downward messages are status requests (e.g., to retrieve a specific sensor value) or actuator commands. Also, different autonomous SCADA systems exchange messages. The message types to be exchanged are individually defined, depending on the participants of an interconnection. Especially, data confidentiality and privacy are considered to address legal obligations, e.g., due to business or political reasons. Therefore, we consider only a subset of data is exchanged that is necessary to support the operation of a distant CI.

Two different control loop classes exist: (i) Safety-critical and (ii) operation-critical. Safety-critical control loops are found between each RTU and its sensors or actuators. Immediate reaction in millisecond ranges is required here to keep the surveilled CI processes stable. On the other hand, operation-critical control loops exist between all SCADA nodes and also across autonomous SCADA system boundaries. Operation-critical control loops have weaker timeliness requirements than safety-critical control loops.

2.3 Fault and Attack Model

In this paper we focus on two fundamental fault and attack classes: Node crashes and data integrity attacks.

Data flow interruptions are usually consequences of node crashes and may conceal crucial monitoring information to decision making SCADA stations. Therefore, node crashes endanger each CIs correct operation. We consider node crashes between source and destination nodes, i.e., the source node is still able to send data, but the transmission path to the destination is disturbed by the crashed node and therefore data gets lost.

Data corruption may be a result of illegitimate data modification on behalf of an attacker. Furthermore, we assume that attackers are not omnipresent and take over a small fraction of peers. The target of data corruption attacks may be any RTU or router in the SCADA system which we assume to be IP based. We do not consider attacks directed against sensors, actuators, or high-level stations. Consequent on a data integrity attack is the provision of incorrect data to the SCADA system which results in an inconsistent system state. The introduced fault and attack classes endanger both safety-critical and operational-critical control loops.

2.4 Design Requirements

In the following, we present the main design requirements demanded to provide protection of large-scale SCADA systems against the discussed faults and attacks. Our objective is to design an add-on protection layer for legacy and evolvable SCADA systems. The protection strategy should besides supervising the SCADA system also provide for immediate and active reaction to reach graceful degradation and a timely recovery of the system in case of faults/attacks.

Flexibility. The solution should be capable to withstand temporary network disconnection or churn effects like frequent entering and leaving nodes. These effects can happen in large interconnected topologies due to node or link failures.

Interoperability. For the multitude of different node types present in a large-scale SCADA system, it is beneficial to mask the nodes heterogeneity. This lowers the customization efforts and eases the solution's deployment.

Minimal Intrusiveness. The desired solution should be minimally intrusive, so existing SCADA applications can be integrated with minimal efforts.

The requirements above show the need for a software layer that mediates between the SCADA applications and the underlying network. This layer should ensure the continuous operation of the applications across the heterogeneous SCADA platforms with their varied network perturbations. This protection software layer is commonly realized by a middleware approach. A middleware should intercept, process and forward SCADA messages, e.g., between the application layer and the network transport layer. In addition, this middleware should fulfil the following crucial requirements.

Protection Enhancement. The new middleware should not introduce new threats to the system, but support it to increase the system's overall resilience against the presented threats.

Resource Frugality. SCADA systems exist for several decades now. Some of these systems employ devices which cannot be regarded as state-of-the-art in terms of computational or storage capacities. Therefore, a solution should not be too resource-intensive in order to meet SCADA's timeliness criteria.

Scalability. Due to many interconnected SCADA networks, a large number of nodes shall cooperate efficiently. Thus, the solution needs to be scalable.

3 P2P-Based Middleware for SCADA Protection

In the following subsection, several middleware approaches will be discussed regarding their suitability in a SCADA context. Subsequently, our approach and its concordance to the requirements will be presented.

3.1 Middleware Approach Selection

There are a variety of approaches to build a middleware.

Publish/Subscribe (pub/sub) [13,4] is an approach that mediates between event/data producers (publishers) and its consumers (subscribers). Pub/sub systems are usually maintained by so called brokers who decide on the dissemination paths. Broker systems represent a weak point of the overall system, since in case they become unavailable for some reason (e.g., an attack), the pub/sub system is rendered ineffective.

Transactional middlewares [9,2] provide strong data consistency and address reliable data dissemination and persistent storage. Yet, this class of systems requires a notable amount of system resources and thereby contradicts the requirement of small overhead consumption.

Web Services [19,15] provide a state-of-the-art communication and data dissemination paradigm. While web services offer high interoperability, the scalability and protection enhancement requirements are violated. Like for the pub/sub approach, web services require central authorities that provide discovery services.

Mobile Agent Systems [21,27,15] are autonomous and decentralized systems used for self-organizational tasks. Their primary target is neither data dissemination nor replication, but they may be employed for network or node reorganization during perturbations. Mobile agent systems meet the scalability and flexibility requirements. However, the less provable dependability and security of mobile agent systems made these systems less accepted.

We propose the usage of P2P technology as a solution, because P2P meets all previously introduced requirements. A detailed comparison of our approach with the given requirements is presented in Section TODO [3,3]. P2P architectures outperform pub/sub systems in terms of the scalability and protection enhancement requirements. Since the broker system functionality is not distributed among all participating nodes. P2P-based middleware outperforms the transactional and web services middleware approaches as these are "heavy" systems with high computational requirements to reach high data consistency. Mobile agents require complex management and fault-tolerance mechanisms which are easily overcome by P2P.

3.2 PeSCADA Architecture

The network architecture of PeSCADA involves RTUs, MTUs, and high level stations (cf. Figure 1). To increase system resilience, PeSCADA is integrated like a middleware into each of the previously mentioned nodes (cf. Figure 2). It resides between the SCADA application and the IP layer, listens to and extracts SCADA messages of the original SCADA application and finally stores them in the P2P network (cf. Figure 3). The listener component is hooked into the SCADA application communication. A SCADA model describes message formats and relevant payload is extracted from the original messages. Consequently, the extracted payload is forwarded to the local P2P client which stores the data in the P2P overlay.

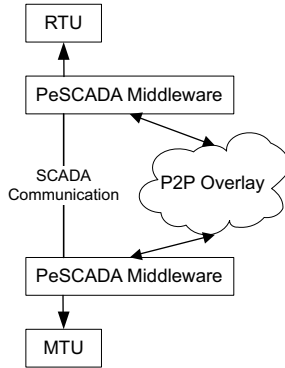


Fig. 2. Integration of PeSCADA into existing SCADA Systems

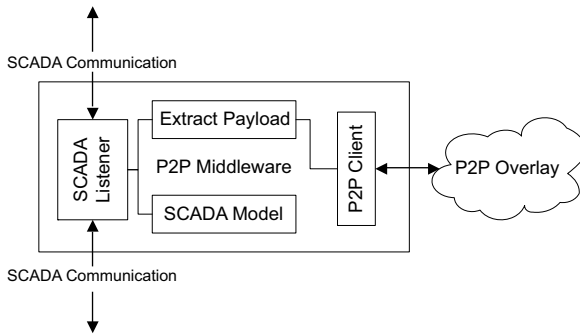


Fig. 3. PeSCADA Middleware Building Blocks

Data storage within PeSCADA is realized with a DHT, which is a distributed associative array that stores key/value pairs and assigns, according to the key’s value, maintenance responsibilities to overlay nodes. The architecture of a DHT is flat, while interconnected SCADA systems are hierarchically organized. To address this architectural difference, each autonomous SCADA system introduces its own *local* overlay network. Local overlays promote legal and performance aspects in interconnected large scale topologies, e.g., data that may reflect corporate secrets is stored locally in the domain of its originating operator. Also, lookup and data retrieval latencies are improved in case overlays are limited to the network of each autonomous SCADA system. Besides local overlay networks all autonomous SCADA systems are part of a *global* overlay network. Data to be shared among different autonomous SCADA systems is specified in filter lists and stored in the global overlay. The notion of local and global overlays is also depicted in Figure 4.

Thus, the functionality of the PeSCADA middleware layer is threefold: (i) Overlay management, (ii) data management, and (iii) SCADA communication

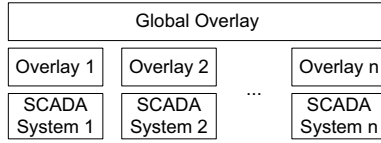


Fig. 4. Arrangement of Local and Global Overlay Networks

interception. The overlay management includes the functionality that is required to maintain a P2P network, in the PeSCADA case either a Chord [26] or Kademlia [18] protocol implementation.

3.3 PeSCADA’s Concordance with Requirements

PeSCADA is based upon structured P2P networks [18,26], since they provide good scalability properties and require $O(\log n)$ routing steps in a network with n peers, whereas unstructured networks require up to $O(n)$ steps.

Structured P2P networks are also capable to handle churn, i.e., frequent entering and departing peers in notable amounts. Therefore, these overlays provide flexibility according to our requirements.

Furthermore, P2P technology meets our interoperability requirement, by providing an identical interface for all peers and thereby masking heterogeneity.

The P2P functionality is inserted into existing SCADA applications as a middleware. This forms a logical layer between the operating system and the SCADA application to intercept, process, and forward SCADA messages. The existing SCADA application needs minimal changes, eventually only a reconfiguration and no source code changes is needed.

Integrating new technology into an existing system bears the risk of introducing new threats. Different P2P-related attacks, like for instance the Sybil attack [12], require an adversary to newly introduce a notable amount of peers to infiltrate the network. Since SCADA systems are no open networks like for example P2P filesharing networks, attackers are not able to arbitrarily introduce new peers. Furthermore, a secure admission protocol [25] could be applied to disallow entering of unsolicited peers. The proposed P2P-based approach does not replace the SCADA system functionality, but represents a supplement to increase SCADA system resilience.

While the previously mentioned attack class targets the P2P routing mechanism, other attacks against structured P2P networks exist [28,8], although they also require malicious peers colluding on behalf of an adversary. It is part of the future work to address these attacks in detail. Due to the natural physical distribution, it is unlikely that an attacker is able to hijack all peers that hold replicas of a specific datum at the same time; this property of P2P networks provides protocol inherent resilience.

The overhead requirement is met as well, since no computationally challenging operations are executed. This requirement was the main driver to decide against

the application of cryptographic methods, since legacy devices cannot achieve both, cryptographic computations and SCADA timeliness requirements.

3.4 P2P-Inherent Resilience Mechanisms

Path redundancy presents a simple robustness concept in P2P networks: Requests can be sent along different paths to both speed up data retrieval and offer increased resilience against node or link failures. The number of redundant paths to be chosen can be configured. Clearly, using more redundant paths implies lower latencies for data retrieval but generates more network traffic in the system.

We evaluated [16] the choice of suitable P2P technology for SCADA systems. Regarding the requirements of interconnected large-scale SCADA systems, structured P2P networks with Distributed Hash Tables (DHT) are appropriate for three reasons, i.e., (i) low routing latencies, (ii) good scalability, and (iii) data discovery guarantees in DHTs [16].

Data replication in DHTs increases the availability of data throughout the network. DHT entries are stored at k different peers, usually $k = 3$. Large values of k result in increased availability and fault tolerance of the system. The downside is reduced system performance due to increasing network traffic. If a peer p that provides a datum d leaves the network, d is still available at $k - 1$ other peers. Subsequent to peer p 's departure, P2P self-organization mechanisms adapt the P2P network's routing tables and choose another peer to store the replicas formerly stored on p .

3.5 PeSCADA Strategies to Increase SCADA System Resilience

In this subsection, we present PeSCADA's anticipation approach for the considered perturbations described in Section 2.3, namely node crashes and data integrity attacks.

Protecting SCADA from Node Crashes. The PeSCADA middleware tracks the reception of sensor messages in MTUs by using expectancy timers to suspect message loss. In case a message becomes overdue, PeSCADA requests the specific sensor message from the P2P overlay network. This mechanism bridges the time between a node crash and its recovery (e.g., by reboot) or until the routing tables are updated through the distributed routing algorithm (e.g., OSPF [22]). This helps to deliver data to the SCADA application during perturbations, i.e., PeSCADA acts as a surrogate data delivery mechanism.

Protecting SCADA from Data Integrity Attacks. PeSCADA is able to discover data corruption attacks, if the location of corruption is between source and destination. We consider corruptions that occur after initial message replication in the overlay, i.e., the corruption occurs on a compromised router. PeSCADA operates as follows: Whenever a SCADA message arrives at an MTU through the conventional SCADA communication channel, the MTU requests the same message via the P2P overlay from q different replica locations and

compares it to the initially received message which is accepted if they are identical. The parameter q needs to be less or equal to k which is the system wide replication degree. Choosing large q means that attackers are required to hijack more nodes to successfully spoof the system. On the other hand, choosing small q results in better performance as less messages will be sent through the network.

The DHT data storage does not take the duties of a distributed SCADA data historian. Since autonomous SCADA systems contain up to 10^6 datapoints that send data in second to deca-second intervals, long term storage in devices with varying and potentially very limited resource capacities cannot be realized. Regarding the two previously introduced strategies, the DHT serves for effects like node crashes or to detect ongoing data integrity attacks. These effects require countermeasures near in time, therefore, short term data storage is in favor. Short term storage also decreases the P2P network load because no republishing of DHT entries is performed. Republishing is required to counteract churn effects in P2P networks and thereby to guarantee long-term data availability. [18] proposes to republish data every 24 hours, PeSCADA does not republish data at all. Furthermore, as an extension to [18] we implemented a time to live (TTL) management for DHT entries.

4 Performance Evaluation

This Section first describes the simulation environment. Next, PeSCADA is evaluated for the two considered perturbation scenarios, i.e., data loss mitigation and detection of data integrity attacks.

4.1 Simulation Environment and Settings

We follow a simulation-based evaluation, since our full-scale system model involving many protocol layers and a large set of parameters is unfavorable to break down into an analytical model.

The simulation is implemented using OMNet++ [20], a discrete event simulator. It provides core concepts like message queues, message passing between objects, and an interpreter programming language to define nodes and networks of nodes. Other simulators exist, but the availability of extensions like INET [1] and OverSim [5] revealed OMNet++'s adequacy for PeSCADA. INET provides an implementation of the IP suite to model and simulate large scale SCADA scenarios. Due to lack of open source SCADA scenario generators that could be coupled with OMNeT++, we created our own SCADA scenario generator for OMNeT++, which we made available for the community. Accordingly, we gather simulation results involving the following INET protocol implementations: ARP, IP, TCP, UDP, and OSPF [22].

OverSim builds upon INET and provides the implementation of different P2P overlay networks. We performed simulations using OverSim's Chord [26] and Kademlia [18] protocol implementations. Both protocols provide a DHT as application layer on top of the P2P network. SCADA sensor data is replicated

within a local DHT to provide it to an MTU in case the regular SCADA application communication is perturbed. The simulation results provided later in this section show the performance of a local DHT which consists of 8 through 512 RTUs acting as peers and 64 sensors per RTU. At startup each sensor chooses a random but fixed sampling period in the range of 1 to 30 sec. The simulation terminates after 600 sec. Faults and attacks are initiated at $t = 100s$ to provide simulated nodes sufficient time for self-configuration tasks and the P2P network setup. The system wide packet drop rate is set to 10^{-3} to model an unreliable underlay network. The TTL for DHT entries is set to 300 seconds.

The following key is an example for the addressing scheme of DHT key/value pairs: `RAW_042_017_20100102122124`. The key represents raw sensor data of RTU 42's sensor number 17, processed and replicated by the RTU on January, 2nd 2010 at 12:21:24. Clearly, key calculation for a sequence of keys is trivial in case of static sensor intervals. In case that the sequence calculation is not trivial, other mechanisms exist which are omitted here.

4.2 Case Studies

Data Loss Mitigation. A router is set to a dead state. Consequently, the OSPF [22] protocol detects this unresponsive router and initiates a route repair. The time span between router crash and the completion of the reconfiguration process is bypassed via P2P, because packets routed across the dead router get lost. MTUs run expectancy timers for sensor messages, and in case a message is indicated to be overdue, the MTU requests this missing message via the P2P network. Simulations use a replication degree of $k = 2$, i.e., two copies of each DHT key/value pair exist in the network to provide basic redundancy. Therefore, we simulate a SCADA network with a mesh topology, such that alternative routing paths may be taken up to a certain amount of node crashes. In our simulation, MTUs run expectancy timers for sensor messages, and in case a message is indicated to be overdue, the MTU requests after a short waiting period the missing data via the P2P network.

Detection of Data Integrity Attacks. An arbitrary RTU is set to a malicious state leading to sensor messages in transfer being corrupted. For each received SCADA sensor message on an MTU, the same datum is requested from q different replica locations. With the multiple received messages, the receiver can check for their SCADA payload equality. Simulations use a replication degree of $k = 3$, i.e., three copies of each DHT key/value pair exist in the network.

4.3 Metrics

To quantify the benefit of our approach, we define several metrics. The evaluation criteria can be split up according to three dimensions, namely *reliability*/*security*, *timeliness*, and *overhead*.

Reliability is measured as a percentage of received messages, that would be lost without the P2P mechanism. The metric’s formula is:

$$Reliability = \frac{\#Receipts}{\#Requests}$$

#Receipts is the number of messages received via DHT and *#Requests* is the total number of missing messages that have been requested.

Security is evaluated as percentage of discovered corrupted messages. The metric’s formula is:

$$Security = \frac{\#Identified}{\#TotalInjected}$$

#Identified is the number of identified corrupted messages, *#TotalInjected* is the total number of corrupted messages that have been injected. Both metrics value ranges are [0, 1], where 1 indicates 100% reception or discovery, depending on the respective scenario.

Timeliness is examined in terms of latency from a request until its completion.

Overhead is evaluated in terms of the number of messages sent/received per peer, and the corresponding incoming and outgoing network traffic per peer. P2P traffic is split up into three subcategories: (i) Application (DHT), (ii) peer discovery (lookup), and (iii) overlay maintenance.

4.4 Simulation Results

In terms of timeliness evaluation, PeSCADA recovers lost messages within the range of 3 to 7 sec using Kademlia [18] and 2 to 5 sec using Chord [26].

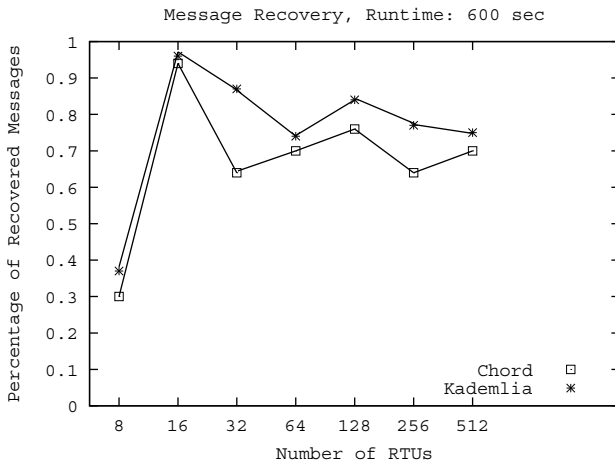


Fig. 5. Success Rate of Lost Message Recovery (Chord & Kademlia)

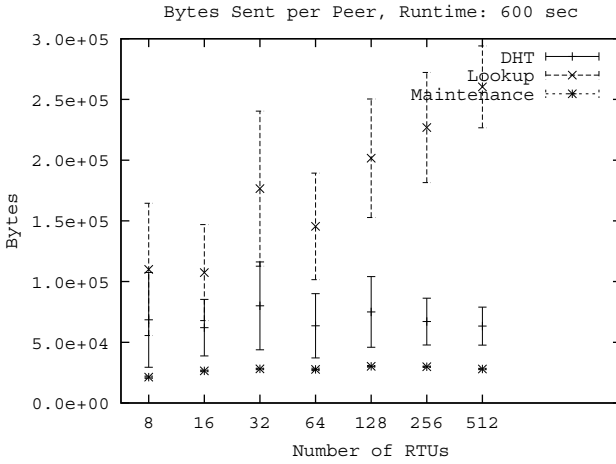


Fig. 6. Sent Messages (Chord)

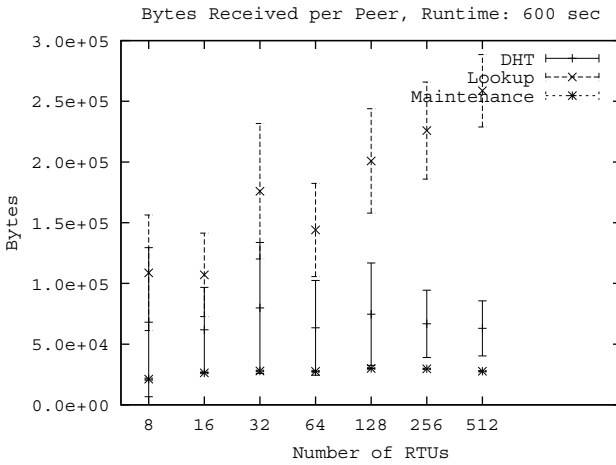


Fig. 7. Received Messages (Chord)

Recovery rates for the router crash scenario are given in Figure 5 in terms of the reliability metric. Kademlia provides success rates above 75% with 16 peers or more. Chord ranges between 65% and 95% with 16 peers or more. Recovery failures occur due to two different causes: (i) The P2P network communication is partially disturbed due to the router crash and therefore unable to satisfy all requests or (ii) P2P messages get lost due to the packet error rate. Fluctuations in the success rates occur due to PeSCADA’s replication and routing scheme: In case the RTUs that are affected by the router crash are responsible for the specific address space range of the requested datum, the MTU cannot retrieve the

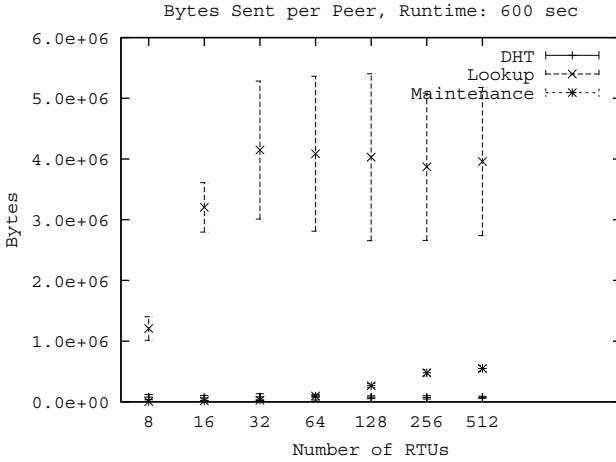


Fig. 8. Sent Messages (Kademlia)

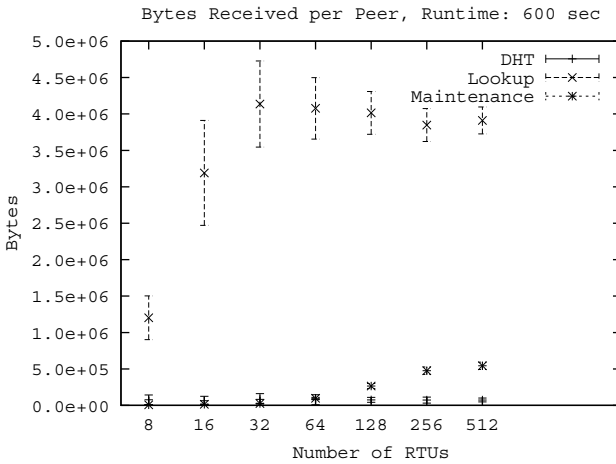


Fig. 9. Received Messages (Kademlia)

data. Our simulations show that Kademlia has increased robustness compared to Chord. However, Kademlia communication overhead exceeds Chord. Figure 8 shows the average amount and the 95% confidence interval of Kademlia messages sent per peer. Figure 9 shows received messages for Kademlia. Compared to Kademlia, Chord requires an order of magnitude less data bytes for DHT messages (for both sending and receiving), as can be seen for the corresponding RTU numbers across Figures 6 and 7 as well as Figures 8 and 9.

For the data integrity attack scenario, PeSCADA is able to discover 90% of the faked messages by requesting each datum from $q = 3$ different peers for comparison. If the SCADA message corresponds to the message received from at least two of three peers, it is regarded as valid. In other cases, the SCADA message is discarded and counted as corrupted message. Consequently, a message copy that is retrieved from the DHT is used. Traffic consumption in this scenario is clearly higher than in the previous scenario, since each received message is requested threefold.

5 Related Work

Several related approaches to increase SCADA system resilience exist in the literature. The approaches presented in this Section differ in terms of the employed technology (P2P based, pub/sub based, MPLS), requirements (data provision, group communication, fast delivery), and communication paradigm (many-to-one, one-to-many, many-to-many).

Some of the previous works [6][11][29] propose a non-intrusive communication infrastructure that is built from scratch. Our approach is minimally intrusive as we build the protection layer on existing already deployed SCADA systems while keeping the change of the existing SCADA system minimal.

P2P for protecting CIs has been proposed before. In [6], the authors qualitatively discuss the advantages and disadvantages of using Chord [26] for the surveillance of power grids. The main contribution is an agent-based layer on top of a P2P network to improve message exchange reliability. In [11] the authors propose a hybrid unstructured overlay network without central indexing services to increase power grid surveillance and monitoring resilience. Although the higher timeliness requirements in comparison to structured networks are mentioned, no evaluation is provided to compare both overlay classes.

In contrast to our approach, pub/sub models [14][13][4] require message broker nodes which conduct the message reception and forwarding mechanism. The notion of message brokers represent like super peers in hybrid P2P systems potential weak points in the system.

[14] proposes a pub/sub middleware to meet the strong timeliness requirements for data delivery within power grid SCADA systems. It aims at satisfying the needs of the electric power system, i.e., low latency and reliable delivery of data produced anywhere in the network and to multiple interested sites. The middleware is based upon a pub/sub communication model, which provides data item transport from a source (the publisher) to various sinks (subscribers) without requiring the publisher to track its subscribers. Experiments show, that the performance in terms of forwarding latency and reliability is sufficient according to power grid requirements.

[29] focuses on power grids and proposes an information architecture aiming at increased reliability. The architecture is twofold, addressing operational and planning aspects, where the first has stronger timeliness requirements than the second. To meet security and reliability requirements, a potpourri of technologies is applied: Multi-protocol label switching (MPLS), Virtual Private Networks

(VPN), and firewalls. Finally, different redundancy topologies are introduced with different scalability properties. [29] provides a solution that requires fundamental changes to the SCADA systems in terms of both, software and hardware.

6 Conclusion and Future Work

We presented PeSCADA, a middleware solution to increase SCADA system resilience in the presence of faults or attacks using a P2P based approach. Our approach consists of building a self-organized structured P2P overlay on top of the SCADA network and in exploiting the inherent path redundancy and data replication to enhance the resilience of the SCADA system. We considered two highly probable fault/attack case studies, namely node crashes and data integrity attacks. In the first case, our solution detects delayed/lost messages through message expectancy timers and requests copies from the overlay network. In the second case, our solution requests for each regularly received SCADA message the same datum from the overlay for comparison, in order to detect data manipulations without any use of cryptographic methods.

As ongoing work, we are customizing PeSCADA to the specific needs of power grids. Furthermore, we are considering the mitigation of further perturbations of SCADA systems through P2P technology.

References

1. INET Framework, <http://inet.omnetpp.org>
2. PostgreSQL, <http://www.postgresql.org/>
3. Androutsellis-Theotokis, S., Spinellis, D.: A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Comput. Surv.* 36(4), 335–371 (2004)
4. Banavar, G., Chandra, T., Mukherjee, B., Nagarajarao, J., Strom, R.E., Sturman, D.C.: An efficient multicast protocol for content-based publish-subscribe systems. In: *ICDCS 1999: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, p. 262. IEEE Computer Society Press, Washington (1999)
5. Baumgart, I., Heep, B., Krause, S.: OverSim: A Flexible Overlay Network Simulation Framework. In: *Proceedings of 10th IEEE Global Internet Symposium (GI 2007) in conjunction with IEEE INFOCOM 2007*, pp. 79–84 (2007)
6. Beitollahi, H., Deconinck, G.: Analyzing the Chord Peer-to-Peer Network for Power Grid Applications. In: *Fourth IEEE Young Researchers Symposium in Electrical Power Engineering*, p. 5 (2008)
7. Bowen III, C.L., Buennemeyer, T., Thomas, R.: Next generation SCADA Security: Best Practices and Client Puzzles. In: *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop, 2005. IAW 2005, June*, pp. 426–427 (2005)
8. Castro, M., Druschel, P., Ganesh, A., Rowstron, A., Wallach, D.S.: Secure routing for structured peer-to-peer overlay networks. *SIGOPS Oper. Syst. Rev.* 36(SI), 299–314 (2002)
9. Codd, E.F.: *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., Boston (1990)

10. Bakken, D.: Smart Grid Data Delivery Service, http://ec.europa.eu/research/conferences/2009/ict-energy/pdf/dave_bakken-en.pdf
11. Deconinck, G., Rigole, T., Beitollahi, H., Duan, R., Nauwelaers, B., Van Lil, E., Driesen, J., Belmans, R., Dondossola, G.: Robust overlay networks for microgrid control systems. In: DSN 2007, 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Edinburgh, U.K., June 25-28, p. 6 (2007)
12. Dinger, J., Hartenstein, H.: Defending the sybil attack in p2p networks: taxonomy, challenges, and a proposal for self-registration. In: The First International Conference on Availability, Reliability and Security, ARES 2006, April 2006, p. 8 (2006)
13. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. *ACM Comput. Surv.* 35(2), 114–131 (2003)
14. Gjermundrod, H., et al.: GridStat: A Flexible QoS-Managed Data Dissemination Framework for the Power Grid. *IEEE Transactions on Power Delivery* 24(1), 136–143 (2009)
15. Ketel, M.: A mobile agent based framework for web services. In: ACM-SE 47: Proceedings of the 47th Annual Southeast Regional Conference, pp. 1–6. ACM, New York (2009)
16. Khelil, A., Jeckel, S., Germanus, D., Suri, N.: Benchmarking of P2P Technologies from a SCADA Systems Protection Perspective. In: MOBILIGHT 2010: Inproceedings of the 2nd International Conference on Mobile Lightweight Wireless Systems (to appear 2010)
17. Krutz, R.L.: Securing SCADA Systems. Hungry Minds Inc. (2005)
18. Maymounkov, P., Mazières, D.: Kademia: A peer-to-peer information system based on the xor metric. In: IPTPS 2001: Revised Papers from the First International Workshop on Peer-to-Peer Systems, pp. 53–65. Springer, London (2002)
19. Papazoglou, M.P., Heuvel, W.J.: Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal* 16(3), 389–415 (2007)
20. Pongor, G.: OMNeT: Objective Modular Network Testbed. In: MASCOTS 1993: Proceedings of the International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems, pp. 323–326. The Society for Computer Simulation, International, San Diego (1993)
21. Pridgen, A., Julien, C.: A secure modular mobile agent system. In: SELMAS 2006: Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems, pp. 67–74. ACM, New York (2006)
22. RFC Standards Track: RFC 2328, OSPF Version 2
23. Rinaldi, S., Peerenboom, J., Kelly, T.: Identifying, understanding, and analyzing Critical Infrastructure Interdependencies. *IEEE Control Systems Magazine* 21(6), 11–25 (2001)
24. D’Antonio, S., Romano, L., Khelil, A., Suri, N.: INcreasing Security and Protection through Infrastructure REsilience: the INSPIRE Project. In: Setola, R., Geretshuber, S. (eds.) CRITIS 2008. LNCS, vol. 5508, pp. 109–118. Springer, Heidelberg (2009)
25. Sandhu, R., Zhang, X.: Peer-to-peer access control architecture using trusted computing technology. In: SACMAT 2005: Proceedings of the tenth ACM symposium on Access control models and technologies, pp. 147–158. ACM, New York (2005)
26. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM 2001: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, pp. 149–160. ACM, New York (2001)

27. Suri, N., Bradshaw, J.M., Breedy, M.R., Groth, P.T., Hill, G.A., Jeffers, R., Mitrovich, T.S., Pouliot, B.R., Smith, D.S.: Nomads: toward a strong and safe mobile agent system. In: AGENTS 2000: Proceedings of the fourth international conference on Autonomous agents, pp. 163–164. ACM, New York (2000)
28. Urdaneta, G., Pierre, G., van Steen, M.: A survey of DHT security techniques. ACM Computing Surveys, http://www.globule.org/publi/SDST_acmcs2009.html (to appear)
29. Xie, Z., et al.: An information architecture for future power systems and its reliability analysis. IEEE Power Engineering Review 22(6), 60–60 (2002)
30. ZigBee Alliance: <http://www.zigbee.org>, <http://www.zigbee.org>

ISO/DIS 26262 in the Context of Electric and Electronic Architecture Modeling

Martin Hillenbrand^{1,*}, Matthias Heinz^{1,*}, Nico Adler^{2,*}, Klaus D. Müller-Glaser^{1,*},
Johannes Matheis³, and Clemens Reichmann³

¹ Institute for Information Processing Technology, KIT, Germany
{hillenbrand, heinz, klaus.mueller-glaser}@kit.edu

² FZI Forschungszentrum Informatik, Germany

adler@fzi.de

³ aquintos GmbH

{matheis, reichmann}@aquintos.com

Abstract. The draft international standard under development ISO 26262 describes a safety lifecycle for road vehicles and thereby influences all parts of development, production, operation and decommissioning. All systems affected by the standard, like anti-trap protection or advanced driver assistance systems, contain hierarchical electric and electronic parts. After publishing the final version, they all should be designed, assessed and documented to the demands of ISO 26262.

The intercommunication structure of the distributed automotive control system, consisting of electronic control units (ECU), sensors and actuators, and functions computed by this control system, are specified by the electric and electronic architecture (EEA). In the context of the ISO 26262, the EEA contributes to the intercommunication of distributed, safety related functions plus the determination of architectures.

This article discusses the impact of the standard on the EEA development and the handling of safety requirements demanded by ISO 26262 during early development phases.

Keywords: Automotive, Architecture modeling, Functional Safety, ISO 26262.

1 Introduction

The increase of number, complexity and interaction of electric and electronic systems in a vehicle, bears growing challenges for development activities in the automotive domain. Besides the decreasing development time, the increasing distribution of functions and their computing control system, the draft international standard for functional safety of road vehicles ISO/DIS 26262 (International Organization for Standardization / Draft International Standard 26262) requires attention. Automotive systems demand safety and reliability, which results in consideration of safety requirements with the same level of priority as the functional requirements of the system to develop [1].

* This research work was supported by the Ministerium für Wissenschaft, Forschung und Kunst Baden-Württemberg (AZ: 32-720.078-1/14).

In the aerospace domain, safety considerations, methods, guidelines and certifications are applied for a long time [2] [3], establishing a safety lifecycle. State of the art processes, concerning safety in the automotive domain, base on hazard analysis, failure mode and effect analysis (FMEA) [4], fault tree analysis (FTA) [5], Markov chains and reviews. A standardized safety lifecycle is not yet applied in the automotive domain.

ISO 26262 [6] standardizes a safety lifecycle process, concurrent to the already applied development processes, which, in the automotive industry, are based on the V-Model '97 [7]. ISO 26262 is the interpretation of the DIN EN 61508 [8] [9] for road vehicles with a maximum weight of 3,5t. The draft standard has been published in July 2009, its publication as international standard is expected for 2011.

ISO 26262 enlarges the concept and design space by another dimension. This is why good processes and well applied tool support are mandatory to compete with legal requirements and in the meantime develop safe vehicles, containing forward-looking technologies.

ISO 26262 is based on the system architecture. The vehicle itself consists of different systems; their electric and electronic architecture is modeled in the according development process. The design and development of the EEA of a vehicle is based on the work products from preceding development phases like the design of a broadly defined system architecture concept. In the future it has to consider the results of analysis, considerations and classification of safety aspects, demanded by ISO 26262. The electric and electronic (EE) part of the system architecture is iteratively refined and detailed during the development process. The impact of ISO 26262 to the modeling of the EEA and the contribution of the EEA modeling towards the fulfillment of the overall safety concept is discussed in this paper.

The following chapter gives a short overview of ISO/DIS 26262. Chapter 3 presents the modeling of EE architectures with respect to safety aspects. The involvement of the EEA development in the safety lifecycle is described in chapter 4. Chapter 5 and 6 describe the flow of interpreting and formatting work products from preceding safety analysis and their handling during EEA development. The relations between classes of the EEA meta-model, depicted in a UML [10] class diagram, and safety aspects are presented in chapter 7. Chapter 8 summarizes the work and gives an outlook to further activities.

2 ISO 26262 Lifecycle

Figure 1 depicts an overview of ISO 26262. During the concept phase, the item is defined ([6] part 3, chapter 5). The item represents a system, an array of systems or a function, to which the ISO 26262 is applied ([6] part 1, chapter 1.69). Based on the item, a hazard analysis and risk assessment is performed, in which hazards are classified and assigned with an automotive safety integrity level (ASIL) [11]. Based on these hazards, safety goals (SG) are determined, and the ASIL that was determined for the hazardous event is assigned to the safety goal ([6] part 3, chapter 7). Functional safety requirements (FSR) are derived from the SGs, inheriting the ASIL from the SG, and are allocated to elements from a preliminary architectural draft of the item ([6] part 3, chapter 8).

In the product development phase on system level ([6] part 4), technical safety requirements (TSR) are formulated to describe how to implement the functional safety concept, containing the FSRs, along with the implementation of the functional concept. During this phase, the system gets more and more refined by partitioning into hierarchical sub-system structures. By the level of granularity, where a sub-system can be refined or realized in software (SW) or hardware (HW) only, the phases for product development on HW and SW level are applied ([6] part 5 and 6). TSRs must be specified ([6] part 4, chapter 6) on each level of system and sub-system granularity, followed by the system design ([6] part 4, chapter 7) on the same level of granularity.

During further development ([6] part 5 and 6), HW and SW systems are refined and TSRs are specified. After specification, implementation and integration on HW and SW level, HW and SW components are integrated step by step. This system integration is covered by [6] (part 4).

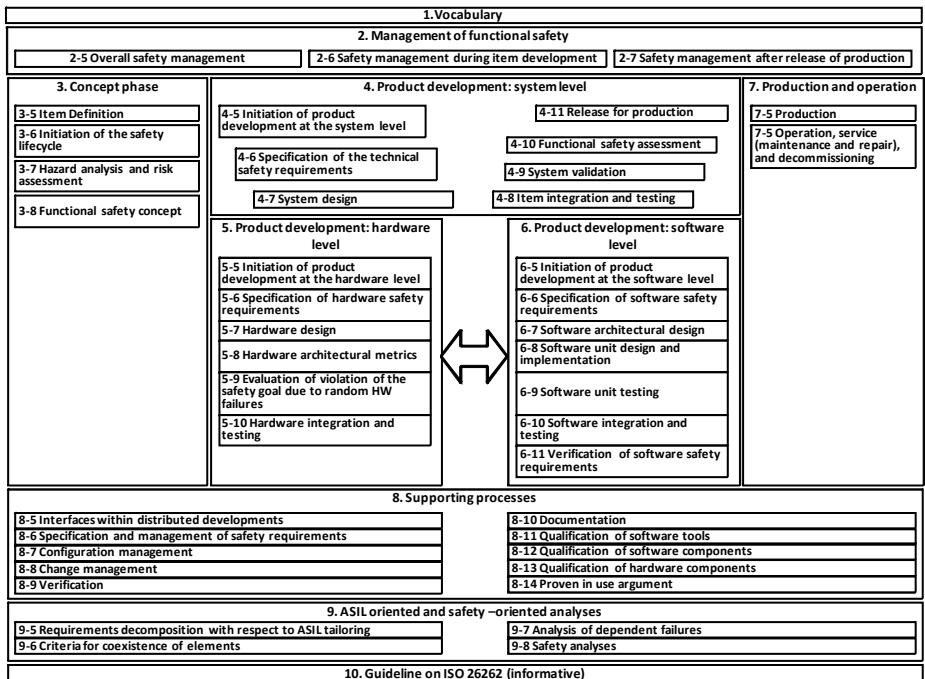


Fig. 1. Overview of ISO 26262

3 EEA Modeling

The development of modern vehicles has to consider numerous technical and functional aspects. Reliability and maintainability as well as usability, functional safety, comfort and performance influence the overall cost function for vehicle design. The demanded quality has to be established and integrated during the design phase.

The tool PREEvision [12] facilitates designing, modeling, comparing and evaluating the electric and electronic architecture of a vehicle in the system design phase [7] of a vehicle development [12] to achieve the optimal overall architectural design. For the first time, all data of the EEA design can be considered in one model. This facilitates the usage of metrics to make an assessment of the EEA. The following section gives a short overview about EEA modeling using PREEvision.

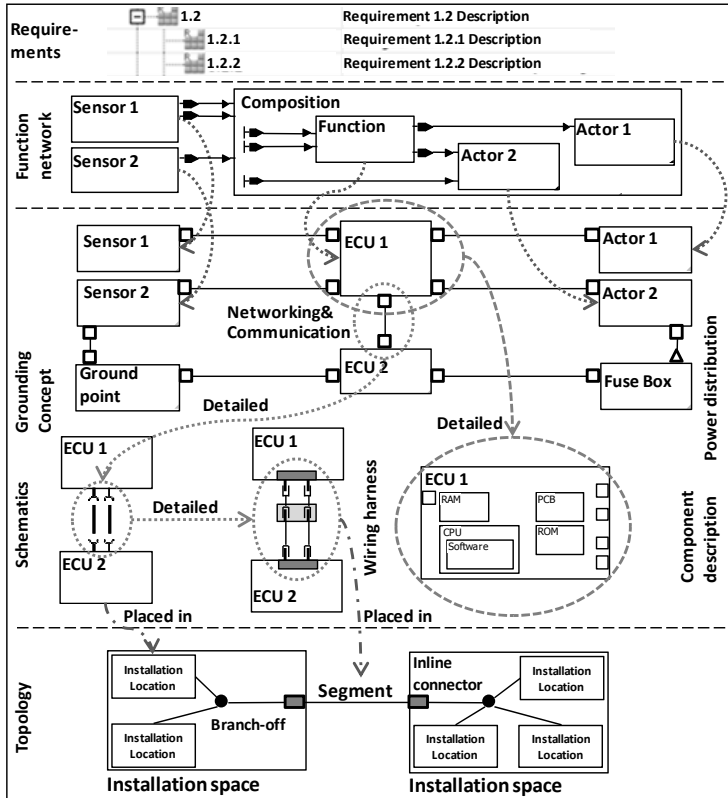


Fig. 2. EEA layered architecture

An EEA model designed in PREEvision is based on the layered architecture (Figure 2) [13]. Each layer depicts a modeling view to the EEA. The EEA model contains requirements-, software-, hardware- and networking-information, which are represented by artifacts in a model tree. Artifacts from the model tree can be visualized in one or more diagram types, each specific to a particular view. Each diagram type offers a specific view on the architecture-model, its artifacts and connections (Function Network Diagram (FN), Component Network Diagram (CMP), Schematic Diagram, Wiring Harness Diagram, etc.). Mappings model the relations between artifacts across diagram borders.

Based on underlying rule- and meta-models, model query rules can be described in PREEvision and later executed on the EEA model for evaluation and consistency checking [14]. These rules can also be used to browse the EEA model for chains of artifacts with a particular relationship.

The consideration of safety aspects and the work products from preceding safety analysis during the EEA design phase, delivers resilient input data for further development phases. Wrong architectural decisions can have devastating impact on safety, reliability and operability of the vehicle and the amount of expenses resulting from mandatory fixes during the later development. Functional safety has to be ensured top down through the development process.

Further, there are methods and strategies applicable during EEA modeling, which have the potential to reduce development and documentation effort (in the context of the safety case [15]) of HW and SW elements caused by allocation, distribution, decomposition and coexistence of safety aspects and EEA artifacts.

It is of severe importance to implement the draft international standard for functional safety ISO 26262 into the vehicle development process and therewith into the phase of EEA development. The next chapter discusses relations and influences.

4 EEA Development in the Context of ISO 26262

A brief overview of the ISO 26262 was given in chapter 2. This chapter considers the role of the EEA development during the processes specified by the draft international standard.

The derivation of safety goals in the EEA tool PREEvision is presented in [16]. The functional safety concept phase will be performed by the role of the safety expert at the original equipment manufacturer (OEM). However, the role of the safety expert at the OEM may consult the EE architect during item definition, the preliminary architectural assumption or the allocation of FSRs to elements of this architectural assumption.

The specification of TSRs during product development on system, HW and SW level, might be performed by both roles, EE architect and safety expert.

The development and refinement of the EEA lies in the responsibility of the EE architect. The compliance of the developed system to the ISO 26262 lies in the responsibility of the safety manager. Although the EEA is not a system within the meaning of the ISO 26262, the development process of the EEA strongly correlates with the ISO 26262 [6] (part 4, chapter 7). This includes the allocation of TSRs ([6], part 4, chapter 7.4.5) to the elements of the system architecture, which are artifacts of the EEA model, the accomplishment of ASIL decomposition ([6] part 4, chapter 7.4.2.5) if adaptable and the assessment for the meeting of coexistence criteria ([6] part 4, chapter 7.4.2.3). Coexistence will be detailed in chapter 6.

The development and implementation of electronic control units (ECUs), including HW and SW parts, is usually accomplished by tier one suppliers. Because of that, subdivision of sub-systems down to the HW- or SW-only level, which are covered by [6] (part 5 and 6), will be mainly performed by the tier one suppliers and their consulting counterpart at the OEM.

The role responsible for a specific electric and electronic component specifies the component HW down to its internal elements, including microcontrollers and memory, and sets up the interacting architecture of functions. This interacting architecture is comparable with AUTOSAR software components (SW-C) put onto a virtual functional bus (VFB) [17].

The EEA does not go further in subdividing the modeled systems into hardware- and software-only systems. Therefore the EE architect is not directly involved in the system development on hardware or software level. Nonetheless, advising during further development phases is possible.

5 Presentation, Import and Interpretation of Safety Requirements

At an OEM, safety requirements will be entered into a requirement tracking tool like DOORS®. At the initiation of the EEA modeling, safety requirements comprising SGs, FSRs and TSRs, if they are already formulated, are imported from the requirement tracking tool into the EEA modeling tool PREEvision as requirement artifacts. If FSRs and TSRs are available, an initial set of safety requirements to be imported to the EEA modeling tool, can be derived from the safety requirements definitions of former production series.

According to ISO 26262, safety requirements comprise several attributes, not all are relevant to be considered for the development of the EEA. SGs express a statement in textual form and have the attribute ASIL. Both should be available in the EEA model. Although the SGs are not directly allocated to artifacts of the EEA ([6] part 3, chapter 8.1), they are needed to track deriving of FSRs. Following the ISO 26262 lifecycle, FSRs are allocated to the elements of the preliminary architectural concept for the item ([6] part 3, chapter 8.2).

Due to the level of abstraction used at the specification of the preliminary architectural concept (Figure 4), the information of all aspects must be partitioned to different diagrams, obligatorily increasing the level of detail.

From the attributes of the FSRs allocated to the elements of the preliminary architectural concept, besides their ASIL, functional redundancy aspects ([6] part 3, chapter 8.4.2.3), warning concepts ([6] part 3, chapter 8.4.2.4), timing constraints ([6] part 3, chapter 8.4.2.3) and additional performance properties (bus load, wire cross-section, etc.) are important, because of their correlation with the elements to model in the EEA and their relationship.

Before and during the development on system level, TSRs are specified for refinement and realization purposes of the functional safety concept. The specification of the first set of TSRs is based on the preliminary architectural concept. TSRs also comprise attributes and considerations ([6] part 4, chapter 6.4.1). Detection, indication and control mechanisms of the system itself (internal architecture) or for devices interacting with the system (sensors, actuators or control mechanisms), can be depicted by high level SW functions (artifacts of the FN of the EEA) or wired connections between the item's HW artifacts and external devices (artifacts of the CMP of the EEA). More details about how this detection, interaction and control, as well as the enabling or achieving of a safe state works, is described by activities. The application of the EEA modeling, to develop a static architecture, does not cover the modeling of functional behavior.

SGs should be structured to keep the derivation structure to FSRs and TSRs traceable. During the safety concept and the system development, safety requirements are derived from each other considering additional aspects (environmental, interfacing, architectural, etc.). Similar specialized requirements derived from different abstract requirements can be combined. Therefore, the hierarchical structure with SGs as root, FSRs on the second level and TSRs on the third level is not feasible. We suggest a flat hierarchy of safety requirements with separate packages for SGs, FSRs and TSRs. The correlation between the requirements is then modeled by links, which additionally allows 1..n relations bottom up, from TSRs over FSRs to SGs. The correlation between the requirements can be tracked and visualized on demand by the execution of predefined model query rules browsing the EEA model for artifacts bearing specific relations.

Generally, the most important attributes of safety requirements are the ASIL and the type of requirement (SG, FSR or TSR). Among others, these attributes require typecasting, to enable automatable analyses on the EEA model based on ASIL and safety requirement types by the application of model query rules.

6 How to Handle Safety Requirements during EEA Development

Based on the preliminary architectural concept and assumptions, used during the development of the safety concept or EEAs from former production series, the EEA is set up. Figure 3 depicts exemplary safety requirements and their attributes ASIL and requirement type, organized and formatted according to the preceding discussion in a PREEvision requirements table. These requirements are allocated to artifacts of the EEA during further EEA development activities. Safety goals and functional safety requirements as well as the following explanations are leant on the example of a powered sliding door from [6] part 10. The technical safety requirements depicted in Figure 3 are additional examples, why their ASIL-cells are grayed out.

List of Safety Goals and Safety Requirements		ASIL	Type
Safety Goals			
1.1	Not to open the door while the vehicle speed is higher than 15 km/h	ASIL_C	SG
Functional Safety Requirements			
2.1	The door actuator will only open the door when powered by the PSDM	ASIL_C	FSR
2.2	The DSC will send the accurate vehicle speed information to the PSDM	ASIL_C	FSR
2.3	The PSDM will allow the powering of the actuator only if the vehicle speed is below 15 km/h	ASIL_C	FSR
Technical Safety Requirements			
3.1	The information about the actual vehicle speed should be actualized with a cycle time of 100ms.		TSR
3.2	The transmission of the information of the actual vehicle speed should be secured by a CRC		TSR
3.3	The plausibility of successive information about the actual vehicle speed should be verified by the receiver		TSR
3.4	The wheel rotation sensors should be diagnosed by the connected ECU		TSR
3.5	A failure at a wheel rotation sensor should be signaled by a yellow warning light		TSR
3.6	If a failure at a wheel rotation sensor is recognized, an according information should be stored in the ECU memory		TSR
3.7	The wheel rotation speed should be measured with an accuracy of at least 30 rad/min		TSR
3.8	The status of the door lock should be monitored all 500 ms		TSR
3.9	On a motion of the vehicle between 0,1 and 15 km/h, an ajar door should be signaled to the driver with a yellow warning light		TSR
3.10	On a motion of the vehicle above 15 km/h, an ajar door should be signaled to the driver with a red warning light		TSR
3.11	The button should be activated longer than 200ms to trigger an action.		TSR
3.12	If the button is pressed more than 10 times within 30sec, the function should be blocked for 5 min.		TSR
3.13	If a false activity of the sliding door actuator is recognized, the controlling and actuator should be transferred in a safe state.		TSR
3.14	The actuator activity of the sliding door should be monitored		TSR

Fig. 3. Safety requirements in PREEvision, example powered sliding door from [6] part 10

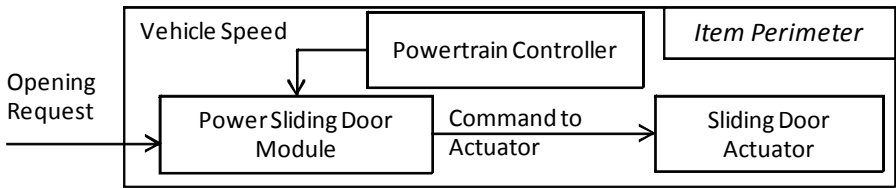


Fig. 4. Item perimeter of a powered sliding door

Figure 4 depicts a preliminary architectural concept of the system in block diagram form, used during safety considerations in the concept phase. The opening of the sliding door during driving activities, and the possibility of people getting seriously injured by accidentally falling out, was identified as hazard for the item (powered sliding door) and classified with ASIL C ([6] part 10).

This simplified architecture, presenting static as well as functional and communication aspects, will in the next steps be refined to a detailed EEA model, presented by PREEvision FN and CMP diagrams.

A FN is set up to realize the demanded functionality. ISO 26262 contains aspects towards safety considerations of hardware-software-interfaces ([6] part 4, chapter 7.4.6) like interrupts, timing consistency, data integrity, memory management, etc. ([6] part 4, Annex B). Regarding the AUTOSAR layered architecture, these are services of the AUTOSAR Basic Software (operating system, communication, micro-controller abstraction, etc.) [17]. These functions and services are not subject of the EEA model. FSRs and TSRs are allocated / mapped to function blocks realizing the functionality and considered under safety aspects ([6] part 4, chapter 7.4.5). Figure 5 depicts the function network including the mapped safety requirements displayed in external boxes.

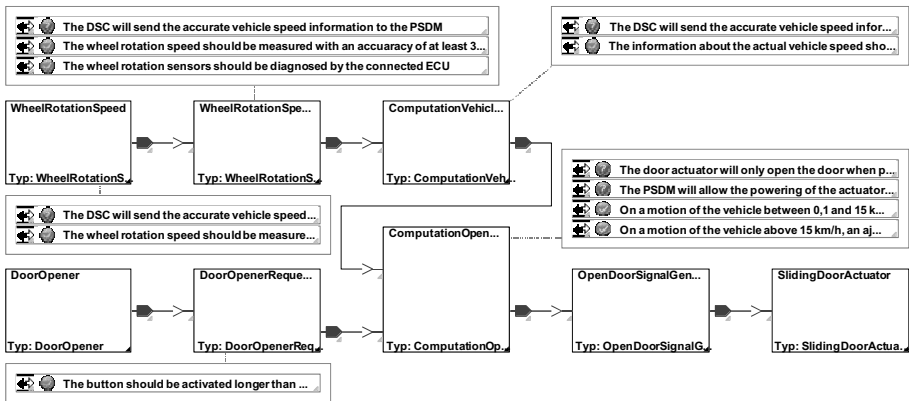


Fig. 5. Function network (FN)

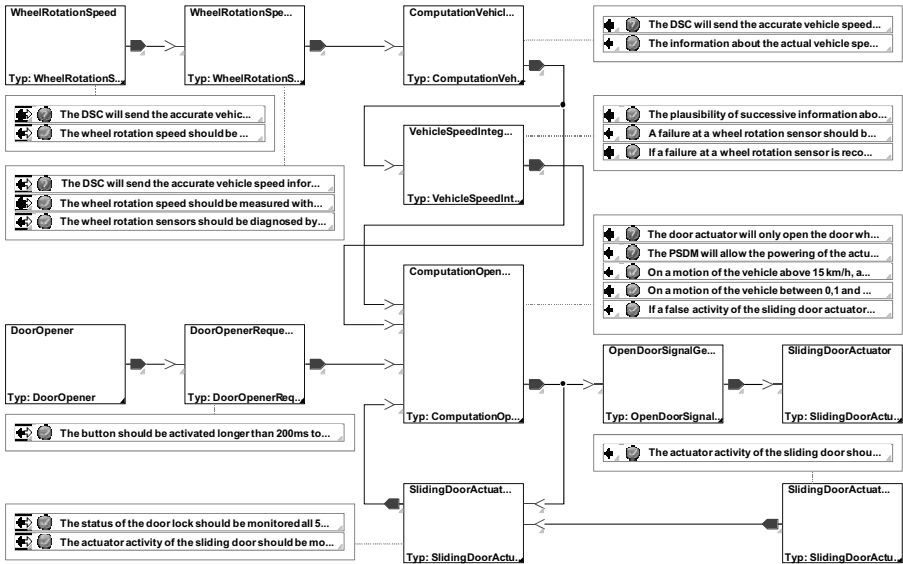


Fig. 6. Refined function network

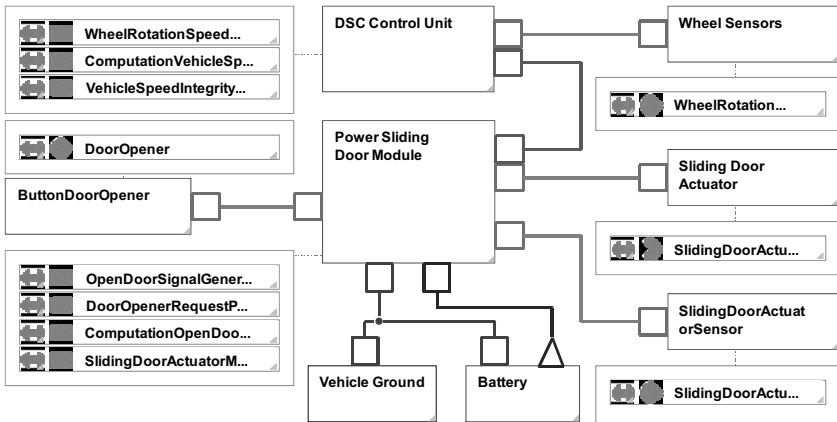


Fig. 7. Component network

If some safety requirements, like plausibility checking or actuator monitoring were not mapped, because no function blocks were available, contributing to the fulfillment of these particular safety requirements, the FN needs additional refinement.

Figure 6 depicts the refined FN. The remaining safety requirements are allocated.

The modeled software architecture must be computed by the computation nodes of the CMP. After setting up the CMP, the functions from the FN are mapped to their computation nodes of the CMP (Figure 7). The mapped functions are depicted in external boxes.

Every electric component including sub-systems and external interfaces inherits the highest ASIL from the functions mapped to them. In addition to hardware elements inheriting an ASIL by computing a function which has an ASIL assigned, there may be TSRs only affecting hardware elements like contact safe plug connections or special wires for electromagnetic compliance.

During EEA modeling, preliminarily, simplified modeled function- or component-systems with ASIL assignment are refined, which includes specification of their internal structure by sub-systems. The criteria for coexistence ([6] part 10, chapter 6) can be applied to the refined system, if it can be proven, that only some of the sub-systems are involved in fulfilling the safety requirements and that these determined sub-systems are not influenced by the others. In that case, only the safety related sub-systems inherit the ASIL. This procedure can be applied to compositions of functions in the FN and component refinement in the CMP.

Figure 8 depicts an example for the application of the criteria for coexistence. The ECU at the left hand side contains two proven independent microcontrollers, only one of them computing a safety related function. The dashed border encircles safety related elements.

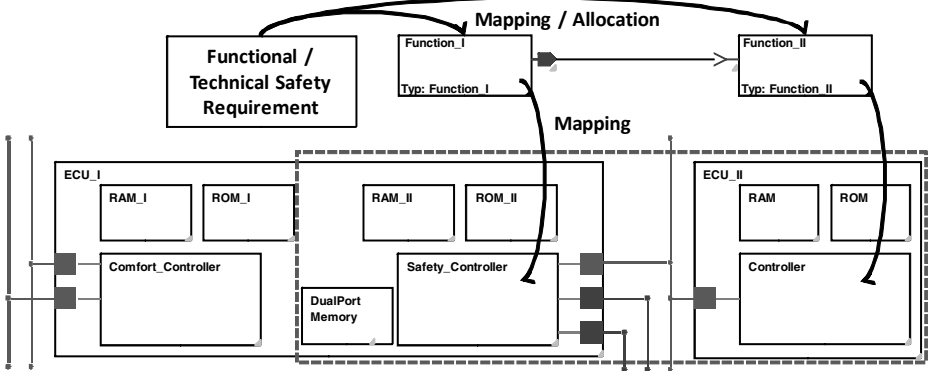


Fig. 8. Coexistence

While designing automotive systems, it will be indispensable to trace the dependencies between artifacts of the EEA, the allocated safety requirements and the ASILs according to which the architecture elements have to be developed later on. Based on the specification of model query rules, the EEA model can be browsed for chains of artifacts in a specific relationship. Based on this, for example all hardware elements involved in fulfilling a SG, all functions assigned to a specific ASIL, or all artifacts having the same safety requirement allocated, can be determined. The results of these model query rules can be displayed to support the overview of the dependencies.

Figure 9 depicts the dependency between ECU (upper right), computed function (middle) and allocated safety requirements (left) by the support of model query rules.

ISO 26262 demands the verification of the system design ([6] part 4, chapter 7.4.8). The verification of the EEA for consistency is mandatory, because work products from the EEA modeling phase are input data to following development phases. The EEA

can support to this by consistency checks executed on the model and browsing for specified inconsistency cases like not correctly inherited safety requirement, etc.

Parts 3 to 7 of the ISO 26262 contain chapters requesting development activities in the context of functional safety. Each of these chapters specifies work products as input to subsequent processes of the lifecycle or for documentation purposes used for the deployment of the safety case. Reports, documenting content and relations of the EEA model, can be automatically generated and formatted out of PREEvision and thereby support the deployment of the safety case. The content of reports is based on results of predefined model query rules.

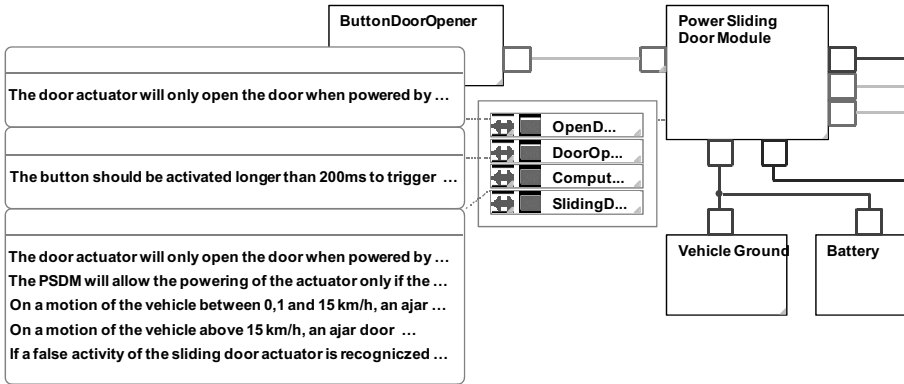


Fig. 9. Dependencies between HW elements, functions and safety requirements

7 Relations between Safety Requirements and PREEvision EEA Artifacts

The assignment of safety requirements and ASILs to artifacts of the EEA is an important step during the EEA development. Especially the determination to which artifacts of the EEA safety requirements have to be assigned and to which not, does not represent an easy task.

EAST-ADL supports determination and modeling of FSRs by safety cases. The derivation of FSRs is not supported [16]. In EAST-ADL-2.0, the relation between an item and its containing elements or systems is depicted by a composition between the class *item definition*, which is interpreted as a collection of entities defining the item that the safety case is valid for (i.e. a "system"), and the abstract class *ADLEntity* [18]. More precise statements about specializations of *ADLEntity* are not provided.

Therefore, the following chapter discusses and specializes the connection between EEA artifacts modeled in PREEvision and safety requirements. A simplified excerpt of the PREEvision meta-model, extended by safety aspects encircling the super class *Safety Related Element*, is applied for this discussion. The class diagram of the applied meta-model is depicted in Figure 10.

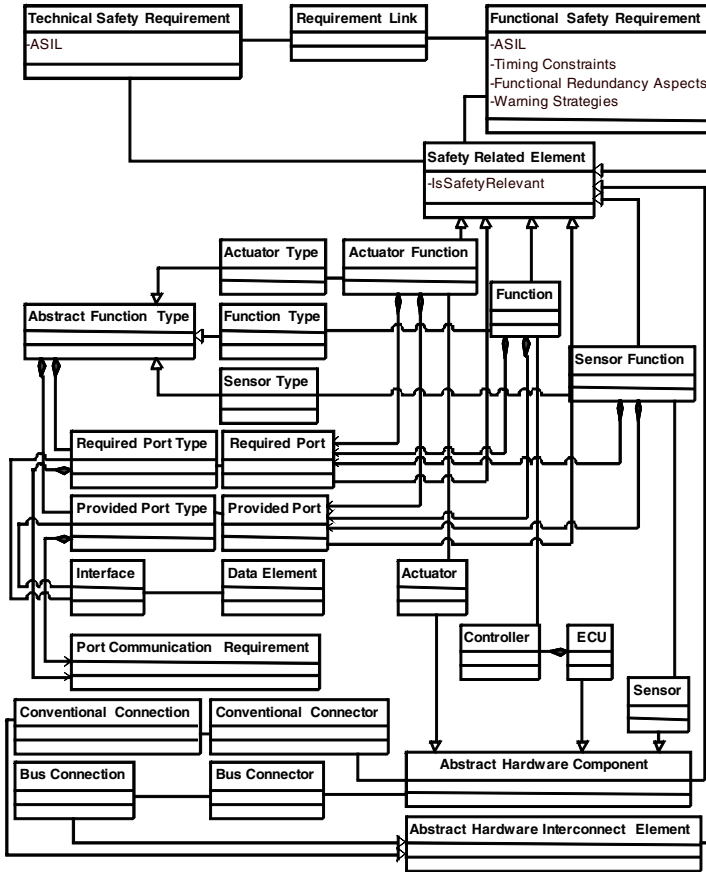


Fig. 10. Simplified EEA meta-model

The super class *Safety Related Element* adds a boolean value for safety relevance to the inheriting instance. The dependence and derivation between the safety requirements can be tracked by the associated *requirement link*.

Abstract Function Type is a typecast for a function (*Actuator Type*, *Function Type* and *Sensor Type*). It is instantiated to fulfill functional safety or non-safety requirements in the FN. The purpose of the usage of an *Abstract Function Type* cannot be foreseen. The instantiation becomes a safety related element by the context of application, but not the type. Therefore the *Abstract Function Type* is not a *Safety Related Element*.

The class *Interface* specifies the communicated data between function objects. Based on their characteristic as types, for most of them, safety relevance can't be determined, caused by the unknown application of their instances. During further modeling, signals are instantiated based on the specification of *Interfaces*. If a signal is a unique communication unit, used to transfer safety related information between hardware elements, it must be regarded as safety relevant.

Like *Abstract Function Types*, *Port Types* are not safety relevant, but their instances can be.

Port Communication Requirement specifies the communication properties of ports (cycle time, etc.). This information has to be considered independently from the fact, if the instance of the specified port belongs to a safety relevant function or not.

Attributes of *Required Port* and *Provided Port* realize the interface of a function. The ASIL is inherited from the function to the ports, realizing the communication of safety relevant information. If a function fulfills safety requirements with different ASILs, all ports of the function get the highest ASIL unless criteria for coexistence is applicable. Regarding the development of the FN in the phases of EEA modeling, this concerns more compositions (systems of functions) than functions. Functions are atomic elements from the EEA point of view. Compositions outline the demanded functionalities and can be detailed during the EEA modeling, which is strongly correlated to the explained scenario.

Elements, realizing interconnection between HW elements (*Abstract Hardware Interconnect Element*), must be considered as being safety relevant. If two interconnected functions are necessary to fulfill a safety requirement, this also concerns the interaction between these functions. If both functions are processed on different HW elements, the HW elements and their interconnection elements inherit the ASIL of the safety requirement.

8 Summary and Outlook

This paper discussed the impact of the future standard for functional safety of road vehicles ISO 26262 to the development in the automotive domain, with special focus on the development of the electric and electronic architectures of vehicles. The additional engineering effort for design, analysis, assessment and documentation, which is demanded by the standard, can be reduced by the well-wrought application of tools. As presented by the handling of safety requirements and their mapping to the item, which comprises software and hardware systems, the decisions of the EE architect influences succeeding development phases of the vehicular systems and systems of systems. The presented methods for the allocation of safety requirements, the refinement of the design, the determination and application of the criteria for coexistence as well as the fast tracking and convincing presentation of the relations between safety information and the artifacts of the EEA model, enables for development of systems demanding functional safety (according to ISO 26262) and support succeeding development activities throughout the vehicle development lifecycle.

Further activities will among others concentrate on the seamless design flow from the development of the safety concept, based on simplified preliminary system architecture and the import and refinement of this architecture during the phase of EEA modeling.

References

- [1] Benz, S.: Eine Entwicklungsmethodik für sicherheitsrelevante Elektroniksysteme im Automobil. Dissertation. Bosch (2004)
- [2] SAE ARP4754. Certification Considerations for Highly-Integrated Or Complex Aircraft Systems (1996), <http://www.sae.org/technical/standards/ARP4754>

- [3] SAE ARP4761, Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment (1996), <http://www.sae.org/technical/standards/ARP4761>
- [4] VDA Verband der Automobilindustrie e.V. Produkt- und Prozess-FMEA. Band 4. Sicherung der Qualität vor Serieneinsatz. Qualitätsmanagemet-Center (QMC) (2009)
- [5] VDA Verband der Automobilindustrie e.V. Fehlerbaumanalyse (FTA). Band 4. Sicherung der Qualität vor Serieneinsatz. Qualitätsmanagemet-Center (QMC) (2009)
- [6] ISO/DIS 26262 Road vehicles – Functional safety – Part 1-10, Standard under development (2009), <http://www.iso.org>
- [7] iABG, V-Modell-97 (1997), <http://www.v-modell.iabg.de/>
- [8] DIN EN 61508-1, VDE 0803-1:2009-06. Funktionale Sicherheit sicherheitsbezogener elektrischer / elektronischer / programmierbarer elektronischer Systeme; Allgemeine Anforderungen (IEC 65A/522/CDV:2008), German Version. Beuth Verlag, Berlin-Vienna-Zurich
- [9] DIN EN 61508-2. VDE 0803-2:2009-06. Funktionale Sicherheit sicherheitsbezogener elektrischer / elektronischer / programmierbarer elektronischer Systeme; Anforderungen an sicherheitsbezogene elektrische / elektronische / programmierbare elektronische Systeme (IEC 65A/523/CDV:2008). German Version. Beuth Verlag, Berlin-Vienna-Zurich (2009)
- [10] Rupp, C., Queins, S., Zengler, B.: UML 2 glasklar. Praxiswissen für die UML-Modellierung und Zertifizierung. Carl Hanser Verlag, Munich-Vienna (2005)
- [11] Maag, B.: Functional Safety of Software Determined Systems Where is the red line? Some Snapshots (2007)
- [12] aquintos GmbH. E/E-Architekturwerkzeug PREEvision (2009), <http://www.aquintos.com>
- [13] Matheis, J., Gebauer, D., Reichmann, C., Müller-Glaser, K.D.: Ganzheitliche abstraktionsebenenübergreifende Beschreibung konsistenter Elektrik/Elektronik-Architekturen. In: Systems Engineering Infrastructure Conference Seisconf. (2008)
- [14] Gebauer, D., Matheis, J., Reichmann, C., Müller-Glaser, K.D.: Ebenenübertreifende, variantengerechte Beschreibung von Elektrik/Elektronik-Architekturen. In: Diagnose in mechatronischen Fahrzeugsystemen, pp. 142–151, Haus der Technik Fachbuch. Expert-Verlag GmbH (2008)
- [15] Bishop, P., Bloomfield, R.: A Methodology for Safety Case Development. Adelard (1999), <http://www.adelard.com>
- [16] Matheis, J.: (TBP 2009). Abstraktionsebenenübergreifende Darstellung von Elektrik/Elektronik-Architekturen in Kraftfahrzeugen zur Ableitung von Sicherheitszielen nach ISO 26262. Dissertation. aquintos (2009)
- [17] AUTOSAR development partnership. Technical Overview, Document V2.2.2, R3.1 Rev. 0001 (2008), <http://www.autosar.org>
- [18] EAST ADL 2.0 Specification. ATESSST (Advancing Traffic Efficiency and Safety though Software Technology) (2008), <http://www.atesst.org>

Author Index

- Adler, Nico 179
Cortellessa, Vittorio 1
Děcký, Martin 72
Dulay, Naranker 1
Eckardt, Tobias 52
Gagnon, Michael N. 125
Gawkowski, Piotr 109
Germanus, Daniel 161
Gibson, J. Paul 89
Grochowski, Konrad 109
Habli, Ibrahim 142
Haines, Joshua 125
Heinz, Matthias 179
Henkler, Stefan 52
Hillenbrand, Martin 179
Huang, Orton 125
Kapadia, Apu 125
Kelly, Tim 142
Khelil, Abdelmajid 161
Lallet, Eric 89
Ławryńczuk, Maciej 109
Marusak, Piotr 109
Matheis, Johannes 179
Merseguer, José 33
Mohamed, Atef 19
Mostarda, Leonardo 1
Müller-Glaser, Klaus D. 179
Raffy, Jean-Luc 89
Reichmann, Clemens 179
Rodríguez, Ricardo J. 33
Sosnowski, Janusz 109
Suri, Neeraj 161
Tatjewski, Piotr 109
Trubiani, Catia 1
Truelove, John 125
Zulkernine, Mohammad 19