DEPEND 2011

The Fourth International Conference on Dependability

ISBN: 978-1-61208-149-6

August 21-27, 2011

Nice/Saint Laurent du Var, France

DEPEND 2011 Editors

Pascal Lorenz, University of Haute Alsace, France

Syed Naqvi, CETIC, Belgium

DEPEND 2011

Foreword

The Fourth International Conference on Dependability (DEPEND 2011), held between August 21-27, 2011 in Nice/Saint Laurent du Var, France, continued a series of special events related to the new challenges in dependability on critical and complex information systems

Most of critical activities in the areas of communications (telephone, Internet), energy & fluids (electricity, gas, water), transportation (railways, airlines, road), life related (health, emergency response, and security), manufacturing (chips, computers, cars) or financial (credit cards, on-line transactions), or refinery& chemical systems rely on networked communication and information systems. Moreover, there are other dedicated systems for data mining, recommenders, sensing, conflict detection, intrusion detection, or maintenance that are complementary to and interact with the former ones.

With large scale and complex systems, their parts expose different static and dynamic features that interact with each others; some systems are more stabile than others, some are more scalable, while others exhibit accurate feedback loops, or are more reliable or fault-tolerant.

Inter-system dependability and intra-system feature dependability require more attention from both theoretical and practical aspects, such as a more formal specification of operational and non-operational requirements, specification of synchronization mechanisms, or dependency exception handing. Considering system and feature dependability becomes crucial for data protection and recoverability when implementing mission critical applications and services.

Static and dynamic dependability, time-oriented, or timeless dependability, dependability perimeter, dependability models, stability and convergence on dependable features and systems, and dependability control and self-management are some of the key topics requiring special treatment. Platforms and tools supporting the dependability requirements are needed.

To deal with dependability, sound methodologies, platforms, and tools are needed to allow system adaptability. The balance dependability/adaptability may determine the life scale of a complex system and settle the right monitoring and control mechanisms. Particular challenging issues pertaining to context-aware, security, mobility, and ubiquity require appropriate mechanisms, methodologies, formalisms, platforms, and tools to support adaptability.

We take here the opportunity to warmly thank all the members of the DEPEND 2011 technical program committee as well as the numerous reviewers. The creation of such a broad and high quality conference program would not have been possible without their involvement. We also kindly thank all the authors that dedicated much of their time and efforts to contribute to the DEPEND 2011. We truly believe that thanks to all these efforts, the final conference program consists of top quality contributions.

This event could also not have been a reality without the support of many individuals, organizations and sponsors. We also gratefully thank the members of the DEPEND 2011 organizing committee for their help in handling the logistics and for their work that is making this professional meeting a success.

We hope the DEPEND 2011 was a successful international forum for the exchange of ideas and results between academia and industry and to promote further progress in the area of dependability.

We hope Côte d'Azur provided a pleasant environment during the conference and everyone saved some time for exploring the Mediterranean Coast.

DEPEND 2011 Chairs

Advisory Chairs
Reijo Savola, VTT Technical Research Centre of Finland, Finland
Sergio Pozo Hidalgo, University of Seville, Spain
Manuel Gil Perez, University of Murcia, Spain
Petre Dini, Concordia University, Canada / China Space Agency Center - Beijing, China

Industry Liaison Chairs
Piyi Yang, Wonders Information Co., Ltd., China
Timothy Tsai, Hitachi Global Storage Technologies, USA

Research/Industry Chair
Michiaki Tatsubori, IBM Research Tokyo, Japan

Special Area Chairs
Cross-layers dependability
Szu-Chi Wang, National Ilan University, Taiwan

Hardware dependability
Peter Tröger, Hasso Plattner Institute / University of Potsdam, Germany

Empirical assessments
Marcello Cinque, University of Naples Federico II, Italy

Security and Trust
Syed Naqvi, CETIC, Belgium

DEPEND 2011

Committee

DEPEND Advisory Chairs

Reijo Savola, VTT Technical Research Centre of Finland, Finland
Sergio Pozo Hidalgo, University of Seville, Spain
Manuel Gil Perez, University of Murcia, Spain
Petre Dini, Concordia University, Canada / China Space Agency Center - Beijing, China

DEPEND 2011 Industry Liaison Chairs

Piyi Yang, Wonders Information Co., Ltd., China
Timothy Tsai, Hitachi Global Storage Technologies, USA

DEPEND 2011 Research/Industry Chair

Michiaki Tatsubori, IBM Research Tokyo, Japan

DEPEND 2011 Special Area Chairs

Cross-layers dependability
Szu-Chi Wang, National Ilan University, Taiwan

Hardware dependability
Peter Tröger, Hasso Plattner Institute / University of Potsdam, Germany

Empirical assessments
Marcello Cinque, University of Naples Federico II, Italy

Security and Trust
Syed Naqvi, CETIC, Belgium

DEPEND 2011 Technical Program Committee

Afonso Araújo Neto, University of Coimbra, Portugal
José Enrique Armendáriz-Iñigo, Universidad Pública de Navarra, Spain
Yudistira Dwi Wardhana Asnar, University of Trento, Italy
Steffen Bartsch, TZI - Universität Bremen, Germany
Jorge Bernal Bernabé, University of Murcia, Spain
Andrey Brito, Universidade Federal de Campina Grande, Brazil
Antonio Casimiro Costa, University of Lisbon, Portugal
Zhe Chen, Université de Toulouse, France
Vicent Cholvi, Universitat Jaume I - Castellón, Spain
Marcello Cinque, University of Naples Federico II, Italy
Vincenzo De Florio, University of Antwerp, Belgium & IBBT, Belgium

Table of Contents

Towards Virtual Fault-based Attacks for Security Validation

R. Leveugle, M. Ben Jrad, P. Maistri
TIMA Laboratory
Grenoble University (Grenoble INP, UJF, CNRS)
46 Avenue Félix Viallet - 38031 Grenoble Cedex - FRANCE
{Regis.Leveugle, Mohamed.Ben-Jrad, Paolo.Maistri}@imag.fr

***Abstract*— Applications increasingly rely on secure embedded systems or "trusted hardware". ASICs (or smart cards) are typically used for high security but SRAM-based FPGAs are also appealing to implement lower-cost and flexible systems. In both cases, designers need a validation of the achieved level of security before they undergo long and costly official security qualification. This paper presents a methodology to accurately evaluate at design time the robustness level with respect to fault-based attacks, without resorting to costly equipments. Practical results are shown. The same methodology can be used in other contexts, for example to evaluate the robustness with respect to particle hits and radiations in spatial or aeronautics electronics although in this case, error models are in general easier to handle.**

***Keywords*— security, dependability, design time robustness evaluation, SRAM-based FPGAs, ASICs**

I. Introduction

Embedded hardware-software integrated systems are today the heart of many products. Most application domains are concerned since embedded systems bring new features, added value and competitiveness due to innovation. In many cases, such systems are critical either from a safety or from a security point of view. In both cases, the systems must undergo accurate validations before they are subject to official qualification procedures. In this paper, we will focus on the validations required to guarantee the expected level of robustness with respect to natural or intentional (malicious) perturbations, with a special focus on the later i.e., so-called fault-based attacks.

Due to the overwhelming costs induced by recent technologies, many applications cannot afford developing a specific integrated circuit. They therefore make use of programmable devices (often called FPGAs). For several reasons that will be detailed in the next section, FPGAs configured by uploading data in a volatile memory (SRAM) are very appealing platforms. However, their configuration can easily be perturbed due to the sensitivity of the SRAM to many perturbation sources. When critical functions such as for example crypto-processors are implemented in such a FPGA, it is therefore necessary to analyze and mitigate the effect of errors not only in the user logic and flip-flops but also in the configuration memory.

We will demonstrate in the next sections that under practical attack conditions (using a laser or power glitches) a large number of bits can be simultaneously modified. The most usual error models employed to analyze the effects of natural perturbations (e.g., single bit-flip, or SEU) are therefore not adequate in such situations. Although fault injection techniques can be used to evaluate robustness level and the efficiency of some protection mechanisms [1], the main problem remains the injection of realistic error patterns.

Of course, the best solution to demonstrate the robustness of a given design is to implement it and to put it under real perturbation conditions, for example in a particle accelerator for accelerated testing or under a laser for malicious attacks. However, using such equipments is not always possible due to their availability and to the cost such experiments induce. A new methodology is therefore required in order to reduce the global validation and qualification costs. Our proposal is to use fault injection campaigns as a first validation step, but with accurate error patterns in order to achieve a sufficient precision. These error patterns are first gathered during platform characterizations and can be re-used for several versions of the design, or several designs. The implemented analysis environment is flexible and can mimic different types of error sources. It only requires qualifying a single time the implementation target (e.g., the selected FPGA family) in the considered environments. Our evaluation environment has been implemented for Virtex II/Virtex II Pro devices but can be extended to other FPGAs.

We detail in section II the global context and motivations for this study. Section III summarizes practical results obtained under various attack conditions on several platforms. Section IV presents how such results can be re-used to perform accurate and low-cost security evaluations. Some results obtained with the new methodology are then shown in section V.

II. Context and Motivations

As previously mentioned, the behavior of an integrated system can be perturbed in several ways. Natural perturbations can occur for example due to ionizing radiations, particle hits or electromagnetic interferences. Malicious perturbations using for example a laser or voltage glitches can be used to discover secret data stored in a circuit. Such fault-based attacks have become one of the main threats for systems with high security requirements. An early example of fault-based attack is Lenstra's attack on RSA [2], taking advantage of the computations based on the Chinese Remainder Theorem. This attack combines fault injection and crypto-analysis to discover the secret key stored in a circuit. Similar attacks have been published for other types of cryptographic primitives, in particular DES or AES [3, 4]. In the case of malicious perturbations, the errors generated in the circuit are often much more complex than those generated by natural perturbations. We will therefore focus on malicious attacks in this paper, although the presented methodology can also be used in other contexts. We will also focus in the sequel on transient effects i.e., spurious modifications of some data stored in the circuit. Several bits can be simultaneously modified but without damage to the chip; restoring the erroneous data can therefore restart a correct computation.

Secure systems are often implemented in ASICs i.e., circuits specifically manufactured for a given application. In the case of large production volumes, for example for Pay-per-view television access, specific circuits allow developers to achieve both low cost and maximum protection. However, the development costs induced by recent technologies are quickly increasing and ASICs are no longer affordable for many applications. In most cases, they are replaced by programmable devices (often generically called FPGAs). Among those devices, three main categories exist. Some devices can be programmed only once (e.g., by fuses or antifuses). Once programmed, they are quite close to an ASIC in terms of characteristics. Their main drawbacks are a limited offer and quite high prices. Another device category makes use of non-volatile memories to store the device configuration. The best devices in this category use today Flash memory that is quite robust with respect to perturbations although some critical parts (e.g., control logic) are more sensitive. Such devices are more flexible but slightly less robust than devices based on fuses. Finally, the third category of devices makes use of volatile memory (SRAM) to store the configuration. These devices are less costly than the others and they generally exploit the most advanced technologies. They offer the largest device complexity, so the best integration level. They can easily be reconfigured to add new features or correct some bugs in the system and they often have partial reconfigurability capabilities that can be used to optimize the system implementation and behavior. Specific protections are available to protect the configuration against cloning (e.g., by encrypting it). SRAM-based FPGAs are therefore very appealing for many applications. However, their main drawback is the sensitivity of the configuration SRAM to natural or malicious perturbations. Errors occurring in a SRAM-based FPGA can therefore affect not only the user logic and flip-flops but also the configuration memory. In the first case, the problem is similar to perturbations occurring in an ASIC and modifies either manipulated data or the application control flow. Errors in the configuration memory are in general much more difficult to cope with since several effects can be induced. Manipulated data can be modified, but the function of the circuit can also be changed and will remain erroneous (although the effect is not destructive) until at least part of the circuit is reconfigured. In some cases, there may also be no effect at all because this bit has no active role in the application definition. Identifying the actual effect of errors in the configuration memory is therefore quite difficult, especially when multiple bits are simultaneously modified. Several types of design techniques can be used to make a given application more robust, but they are more limited than in the case of an ASIC design, since they have to be compatible with the existing features in the FPGA; it is for example not possible to add some sensors to detect a given kind of attacks. Achieving and validating a given security level is therefore difficult. Before using costly experimental equipments such as lasers, the designer must justify that the implemented functions are well protected against realistic errors. In most cases, the efficiency of the protection mechanisms is evaluated at design time by fault injections (either based on simulation or emulation) [1]. One of the limitations is the accuracy of the error model typically used during the injections, that may adequately represent some types of perturbations but not necessarily all types. The approach proposed in Section IV aims at overcoming this limitation. The need for specific error pattern characterization is first illustrated in Section III.

III. Error Patterns with respect to Attack Types

A. *Results of Laser-based Attacks*

One of the most efficient means for fault-based attacks is today a laser. Such equipment allows the attacker to have a very good control on the error location, both in space and time. However, several types of lasers exist and they can be used in several manners; actual attack effects depend on these attack conditions.

We will summarize here some results obtained during practical attack experiments, in order to illustrate the variability of the possible effects.

We reported in [5] an attack campaign done on an ASIC manufactured in the ST HCMOS 130 nm process, with 6 metal layers. The ASIC implemented several versions of a Montgomery multiplier for RSA acceleration, two of them with parity-based protections (Protected1 and Protected2). The laser was a pulsed Yag laser with a green output at 532 nm, 6 ns impulsions and an energy tunable from 0 to 100%, with the possibility to control the spot size. During the experiments, we used a large spot size and we only used the energy "zero" that means the lowest possible energy level, corresponding to some "leakage beam". As shown in Figure 1, this very low energy was sufficient to create many errors in the circuit as demonstrated by the number of alarm signals that were activated, each of them corresponding to a different subset of the logic. This is especially interesting because the result was obtained without any special preparation of the circuit. We just opened the package and attacked the circuit front side through the metal layers.

Similar attacks on this circuit were attempted using a much more sophisticated laser [6] part of the ATLAS platform, the pulsed laser facility of the University of Bordeaux, dedicated to laser testing and analysis of integrated circuits. We used an ultra-short pulsed laser source (1 ps) with a wavelength tunable from 780 nm to 1000 nm and microscope objectives giving adjustable spot sizes ranging from 1 µm to 20 µm. The available laser pulse energy on the attacked circuit is adjustable up to typically

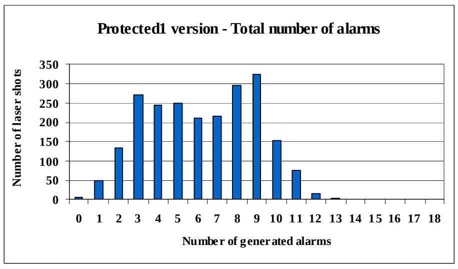
1 nJ. In that case, it was not possible to generate errors when attacking the circuit front-side.

This case study demonstrates that (1) complex error patterns can be generated, far from the usual single bit-flip or even multiple bit-flip (MBU) models and (2) the attack effects strongly depend on the perturbation source. As a matter of fact, the most sophisticated equipment is not always the most efficient to be used in attack contexts.

Experiments have then been carried out on SRAM-based FPGAs, especially Xilinx Virtex II devices. The reason was the availability of a tool called SEFEA-ProD that allowed us to analyze in details the configuration and data errors in these devices [7]. The v1000 FPGA used in the experiments was manufactured in a 0.15 µm CMOS process, with 8 metal layers. It is encapsulated Flip-Chip and was attacked backside through the substrate. Experiments were done on the ATLAS platform [8] but also with other lasers. One of them had a wavelength near 900 nm, a power of a few Watts and several focus levels corresponding to 8 µm, 20 µm, 40 µm and 100 µm expected spot diameters [9, 10]. For the experiments reported in [9, 10], the die was thinned by a mechanical process until a residual thickness of 30 µm to ensure a good optical transmission of the light in the active layers of the device. The various experiments allowed us to draw some conclusions.

First, in most cases, a medium or large number of configuration bits are modified even after a single shot but single-bit errors can also be obtained [10].

Second, one type of bit-flip (from 1 to 0) has a much higher probability than the other. This was explained by the structure of the memory elements [10].



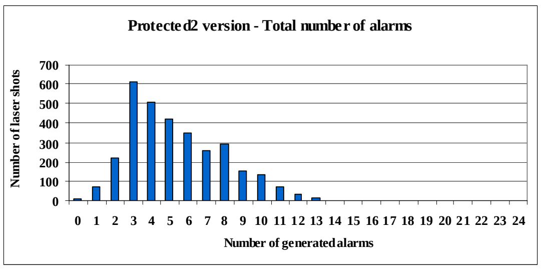


Figure 1. Repartition of shots with respect to the number of asserted detection bits in the protected Montgomery circuits.

Third, it was shown that in spite of multiple bit flips a non negligible percentage of errors does not affect the basic functionality. This is partly due to the unused resources in the FPGA for a given design, but also to a redundant encoding of interconnection control bits in the architecture.

Finally, the detailed analysis of the modified configuration bits demonstrated noticeable differences in terms of repartition depending on the laser energy and on the spot focalization. As an example, it was shown in [9] that the slice inputs and the Hex lines were the most sensitive elements in the CLB tiles (Configuration Logic Blocks), with 70% of errors impacting such connections. However, the most sensitive element between these two was dependent on the laser spot size. With a laser spot size of 40 µm, the probability to modify a slice input control bit was larger than the probability to flip a control bit of Hex lines. With the 8 µm spot, the trend was opposite.

B. Attack Parameters and Adversary Model

Results summarized in the previous section clearly show that it is not possible to use a simple error model to represent all possible attacks. An attacker can modify several important parameters and he will use the best values to achieve his goals. The robustness of a given design must therefore be validated taking into account a given adversary model based on some knowledge about the implementation technology, on the attack equipment that is supposed to be available to the attacker and on the configuration parameters for this equipment.

When laser-based attacks are concerned, the main parameters are related to:

- Manufacturing technology, internal architecture and internal topological organization of the circuit,
- Type of laser and configuration (wavelength, energy, pulse duration, spot sizes),
- Design implementation (placement in the device).

This implies first that a real validation of the design robustness can only be done quite late in the design flow; the final placement and routing must be achieved before.

Also, the validation cannot be done without taking into account realistic error patterns for the different attack conditions. Limiting the evaluation to a generic error model (such as single bit-flips or multiple bit-flips with a given maximum multiplicity) will not be in general realistic. On the other hand, single bit-flips can occur and therefore cannot be completely neglected. Also, when multiple errors occur in the configuration, all the potential combinations of erroneous bits are not realistic since they depend on the laser parameters but also on the physical organization of the configuration bits in the FPGA array. Making multiple injections in the configuration on the basis of a random sampling is therefore not accurate even if very large multiplicity values are taken into account. A more accurate representation of possible error patterns is therefore necessary to achieve significant validation campaigns based on fault injections.

C. Glitch-based Attacks

In the previous sections, we only considered laser-based attacks. Attacks can also be based on other types of perturbations, for example glitches on the power or clock signals. In general, the clock of a secured circuit is protected (e.g., by using an internal oscillator, or a digital locked loop in a FPGA) so we will focus here on glitches induced on the power lines that are more difficult to mitigate when FPGAs are used. Depending on the glitch intensity and on the occurrence time, the effects can be very different. As an example, Figure 2 [11] shows results with several intensity and polarity of glitches. The attack was made on a v1000 device running an AES encryption. The glitches were triggered either on the rising clock edge or on the falling clock edge, with intensity between 5V and 80V, during the encryption cycles 49 to 60. As illustrated in the figure, the average number of bits modified in the FPGA configuration is very large (several hundreds in most cases) and clearly depends on the injection time within the clock period [11]. In that case, injections on the falling clock edge led to about twice erroneous bits.

The need for a good adversary model, with an accurate representation of possible error patterns, is therefore also confirmed in that case.

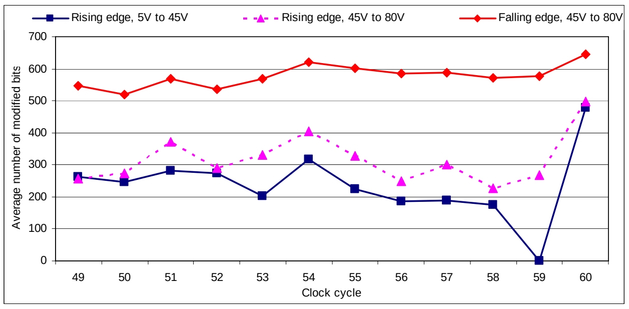


Figure 2. Average number of modified configuration bits in a v1000 device running an AES encryption for several types of power glitch attacks.

IV. Methodology Overview

A. *Fault injection techniques*

Several approaches have been used to evaluate at design time the dependability level of a circuit. We will focus here on approaches able to inject errors in both user flip-flops and configuration memories for SRAM-based FPGAs. In that case, simulation techniques are in general not adequate because there are almost no simulation models available including the configuration memory of commercial devices.

One approach proposed to perform fault injection campaigns is based on partial reconfiguration of a circuit prototype implemented on a SRAM-based FPGA [12-13]. The approach was primarily developed to inject transient errors in the user logic but had the potential to study the effect of errors in the configuration memory as well. The control of the reconfiguration was made by a host PC connected to the FPGA board. In [14], the authors proposed to control the partial reconfiguration by a program executed on an embedded processor. Such an "endo-reconfiguration" has the advantage to considerably reduce the number of data to be exchanged between the FPGA component and the host PC, thus accelerating the fault injection process. The approach is made possible by using the ICAP interface (Internal Configuration Access Port) provided by Xilinx for the Virtex FPGAs. Only SEUs (Single Event Upsets i.e., single bit-flips) were considered. Run-time reconfigurability is also used in [15] to inject faults in Look-Up Tables (LUTs) and in user flip-flops. An approach similar to [14] is presented in [16] but takes into account some multiple and cumulative bit errors. In addition, the environment can perform error propagation analyses, but restricted to specific fault detection or tolerance features. In [17], the FLIPPER platform is introduced to emulate SEU-like faults. Partial reconfiguration is again used but on a more complex platform composed of a main board and a daughter board with the FPGA under test. Complex dedicated hardware is therefore necessary. Similarly, dedicated hardware is required by FT-UNSHADES-C [18]. In addition, although this environment uses partial reconfiguration, it also requires implementing two copies of the system under test, one being used for a golden reference. The first consequence is of course to severely limit the complexity of the system that can be evaluated on a given device. The second consequence is that it is in general not possible to have the same placement and routing of the system during the evaluation and in the final product. Since the consequence of a given bit-flip strongly depends on the placement and routing of the design, such an approach is often too intrusive to give significant results. By comparison, using the ICAP interface only requires a small part of the device logic. With a guided placement and routing, it is often possible to avoid any change in the design implementation.

A few other approaches have been reported. On-the-fly modification of the configuration bitstream of XC4000 FPGAs, during reconfiguration, by some logic on the board was proposed in [19] but requires additional board-level modifications. The approach used in [20] focused on the logic used to reconfigure the FPGA component.

In the sequel, we will give results on a platform developed using the ICAP interface, but the methodology may be adapted on other types of fault injection platforms.

B. *Virtual Fault-based Attacks on SRAM-based FPGAs*

In general, fault injection campaigns are performed on the basis of a given error model, most often the SEU model implying one bit flip at a time in a circuit. In some cases, multiple bit errors are considered, assuming a given multiplicity value and a random distribution. However, as previously explained, such models do not represent well some perturbation conditions and they are unable to accurately take into account the actual layout and sensitivity of the elements in a chip or the physical characteristics of the perturbation source. Such models may therefore lead to large errors on the robustness quantitative evaluations.

Our goal is therefore to inject more realistic patterns, based on previous pre-characterization of the technological target. This pre-characterization can be done once for a given device and a given perturbation source (e.g., some particle flux, some electromagnetic fields or some type of laser with a given focus and a given energy). It can be done statically on the idle circuit, if possible using a device configuration that covers most of the possible configuration patterns for the CLBs and embedded memory blocks. It can also be done dynamically on the circuit running a given application. In the later case, results can be more precise with respect to this application but can induce some bias if the analysis has finally to be done for a different application.

The actual error patterns obtained during this device pre-characterization are recorded and analyzed, e.g., with a tool like SEFEA-ProD [7], indicating all erroneous bits obtained in a bitstream after readback. Then the absolute values of the erroneous bits for each recorded error pattern are abstracted in order to obtain error coordinates that are relative to a CLB.

The abstracted relative coordinates are stored in a database that is then used during the fault injection campaigns performed at design time using partial reconfiguration, without resorting again to the physical perturbation source. Each relative error pattern in the database can be relocated i.e., injected into any CLB (or memory block) used in the device for a given design and at any clock cycle during the application execution for dynamic robustness evaluations. Of course, this relocation is only possible thanks to the regularity and repeatability of a FPGA structure. Another advantage of this regularity, in the case of laser-based pre-characterizations, is that only a small representative part of the device has to be scanned; the effects induced in the whole CLB matrix can then be inferred. This can noticeably reduce the laser availability requirements for the device pre-characterization. The relocation process is illustrated in Figure 3.

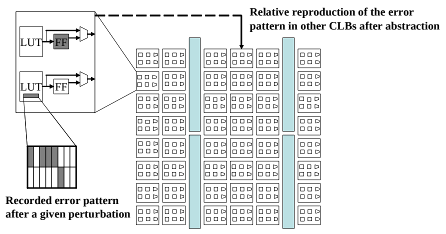

Relative reproduction of the error pattern in other CLBs after abstraction
LUT
FF
LUT
FF
Recorded error pattern after a given perturbation

Figure 3. Illustration of error pattern relocation.

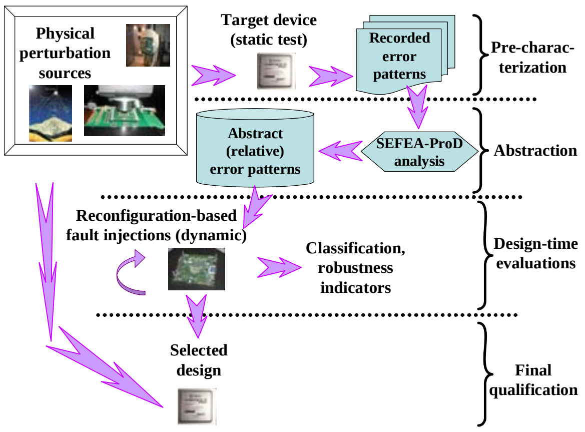

Physical perturbation sources
Target device (static test)
Recorded error patterns
Pre-charac-terization
Abstract (relative) error patterns
SEFEA-ProD analysis
Abstraction
Reconfiguration-based fault injections (dynamic)
Classification, robustness indicators
Design-time evaluations
Selected design
Final qualification

Figure 4. Global robustness evaluation flow.

Using the generated database, the same campaign can easily be run on several versions of a design in order to compare several implementations or several protection schemes. Furthermore, the consequence of any pattern obtained once in the device during the pre-characterization can be assessed at any position in the whole device and at any time. Let us notice that this would not be possible for example under a particle beam (for natural perturbations) due to the random distribution and the usually small number of events obtained during a given beam slot. A more comprehensive and realistic robustness analysis can thus be achieved at low cost with our methodology. The global flow is summarized in Figure 4.

C. *Virtual Fault-based Attacks on ASICs or some FPGAs*

For FPGAs with permanent or error-immune configurations, the same methodology can be applied but limited to errors in the user flip-flops.

In the case of ASICs (or any masked circuits) the relocation principle presented in the previous section is in general not applicable. However, the same concept can be used for regular circuits (parallel architectures, multi-cores …), limited to the repetitive structures.

Another possible use concerns the evaluation of software-related robustness. In the case of a microprocessor (or DSP), for example, the pre-characterization allows recording possible error patterns within the processor core. It is then possible to use the proposed approach to evaluate off-beam the effect on robustness of software modifications, provided a synthesizable model (or a very accurate simulation model) exists for the microprocessor. A processor prototype can be implemented on the FPGA and only error patterns in user flip-flops are injected.

V. Implemented Framework and Results

Our environment is implemented on a Virtex II-Pro component, using one of the embedded PowerPC processors to manage the fault injection campaign and the ICAP IP to perform the partial endo-reconfiguration. This environment is easily portable to other Virtex families. It allows the injection of any single- or multiple-bit error in the configuration and in the user flip-flops of the Configuration Logic Blocks (CLBs). The fault injection can be triggered at any time during the execution of an application onto the FPGA. The originality of the platform lies in the database we added to store the realistic error patterns, as presented in the previous section.

The methodology has been applied in the case of the Leon2 microprocessor, implemented on a Virtex II Pro device, and running several program examples chosen to be representative of several application areas: Fir is a FIR filter, Mtmx is a matrix multiplication, Sieve is a computation of prime numbers with the Sieve of Eratosthenes and AES is a standard encryption/decryption function. The results illustrate the impact of realistic error patterns compared to the most used model in the literature i.e., single bit-flips in the user flip-flops. The relative error patterns were derived from previous experiments [9]. The database included 5435 error patterns. Each pattern involved between 1 and 41 erroneous configuration bits, with an average of 11.7 bits per pattern. A single pattern can involve bits in several LUTs (from 1 to 6 LUTs in the patterns used for the experiments, 1.7 on an average). For each program, we have injected the relative error patterns in the CLBs used to implement the

Leon2 integer pipeline. We used statistical fault injection and the number of randomly selected injection time and location was chosen so that the margin of error on the classification results is 5% with 95% confidence. The number of injections performed for a given pipeline stage and a given program ranges from 8,000 to 160,000. The injected error patterns were classified, based on the system behavior after the injection, as "Silent" (no effect on the application results), "Error" (wrong result but the system is still alive and can perform other computations), "Failure" (wrong program termination, unexpected behavior, uncertain future behavior) or "Crash" (fully unrecoverable error and system behavior, until reset). The results are summarized in Table I for errors globally injected in the five pipeline stages. They clearly show that realistic error patterns lead to much more application failures. So an evaluation based on single bit-flips noticeably under-estimates the failure probability and is not acceptable for security analyses.

TABLE I. EXPERIMENTAL RESULTS OF FAULT INJECTION CAMPAIGNS

		Silent	Error	Failure	Crash
Fir	SEU FF	54.61%	11.63%	10.88%	22.89%
	Error patterns	3.40%	0.98%	60.54%	35.08%
Mtmx	SEU FF	52.59%	12.03%	14.55%	20.83%
	Error patterns	7.61%	0%	31.67%	60.72%
Sieve	SEU FF	47.77%	12.11%	23.46%	16.66%
	Error patterns	4.90%	0%	40.08%	55.02%
AES	SEU FF	57.96%	8.52%	24.67%	8.85%
	Error patterns	18.41%	0.52%	44.21%	36.85%

VI. CONCLUSION

Results shown in this paper demonstrate that classical fault injection campaigns based on single bit-flips noticeably overestimate the robustness of a design, at least for the type of perturbation considered here. The proposed methodology leads to more accurate robustness evaluations, with a minimum use of costly equipments since pre-characterization is done only once for a given device and a given perturbation source.

In this paper, we focused on malicious attacks because error patterns are in general more complex than those obtained under natural conditions. However, the same strategy can be used on the basis of a few events obtained for example under a particle beam, to take advantage of regular structures and increase the confidence in robustness. This approach will be used in further work to evaluate the efficiency of error mitigation techniques.

REFERENCES

[1] R. Leveugle, "Early analysis of fault-based attack effects in secure circuits", IEEE Transactions on Computers, vol. 56, no. 10, October 2007, pp. 1431-1434

[2] A. K. Lenstra, "Memo On RSA Signature Generation In The Presence Of Faults", private communication (available from the author), September 28, 1996.

[3] R. Anderson and M. Kuhn., "Low Cost Attacks on Tamper Resistant Devices", 5th International Workshop on Security Protocols (IWSP), 1997, LNCS 1361, Springer-Verlag, pp. 125-136

[4] J. Blömer and J.-P. Seifert, "Fault based cryptanalysis of the AES", e-Print Archive of the IACR, 2002, http://www.iacr.org/.

[5] R. Leveugle et al., "Experimental evaluation of protections against laser-induced faults and consequences on fault modelling", Design, Automation and Test in Europe Conference (DATE), April 16-20, 2007, pp. 1587-1592

[6] V. Pouget, D. Wan, P. Jaulent, A. Douin, D. Lewis, and P. Fouillat, "Recent developments for SEE testing at the ATLAS laser facility", Proc. of 15th Single-Event Effects Symposium, 2006

[7] V. Maingot, J. B. Ferron, R. Leveugle, V. Pouget, and A. Douin, "Configuration errors analysis in SRAM-based FPGAs: software tool and practical results", Microelectronics Reliability, Elsevier, vol. 47, no. 9-11, September-November 2007, pp. 1836-1840

[8] V. Pouget et al., "Tools and methodology development for pulsed laser fault injection in SRAM-based FPGAs", 8th Latin-American Test Workshop (LATW), March 12-14, 2007, pp. 167-172

[9] G. Canivet et al., "Detailed analyses of single laser shot effects in the configuration of a Virtex-II FPGA", 14th IEEE International On-Line Testing symposium, Rhodes, Greece, July 6-9, 2008, pp. 289-294

[10] G. Canivet et al., "Characterization of effective laser spots during attacks in the configuration of a Virtex-II FPGA", 27th IEEE VLSI Test Symposium (VTS'09), May 3-7, 2009, pp. 327-332

[11] G. Canivet, "Analyse et conception sécurisée de plates-formes reconfigurables", PhD thesis, Grenoble INP, Grenoble, France, September 2009 (in French).

[12] L. Antoni, R. Leveugle, and B. Fehér, "Using run-time reconfiguration for fault injection in hardware prototypes", IEEE Int. Symposium on Defect and Fault Tolerance in VLSI Systems, 2000, pp. 405-413

[13] L. Antoni, R. Leveugle, and B. Fehér, "Using run-time reconfiguration for fault injection applications", IEEE Transactions on Instrumentation and Measurement, vol. 52, no. 5, October 2003, pp. 1468-1473

[14] L. Sterpone and M. Violante, "A new partial reconfiguration-based fault-injection system to evaluate SEU effects in SRAM-based FPGAs", IEEE Transactions on Nuclear Science, vol. 54, issue 4, part 2, August 2007, pp. 965-970

[15] P. Kenterlis et al., "A low-cost SEU fault emulation platform for SRAM-based FPGAs", 12th IEEE International On-Line Testing symposium, Como, Italy, July 10-12, 2006, pp. 235-241

[16] C. Bolchini, F. Castro, and A. Miele, "A fault analysis and classifier framework for reliability-aware SRAM-based FPGA systems", 24th IEEE Int. Symposium on Defect and Fault Tolerance in VLSI Systems, Chicago, IL, USA, October 7-9, 2009, pp. 173-181

[17] M. Alderighi et al., "Evaluation of single upset mitigation schemes for SRAM bassed FPGAs using the FLIPPER fault injection platform", 22nd IEEE Int. Symposium on Defect and Fault Tolerance in VLSI Systems, Rome, Italy, September 26-28, 2007, pp. 105-113

[18] L. Sterpone, M. A. Aguirre, J. N. Tombs, and H. Guzmán-Miranda, "On the design of tunable fault tolerant circuits on SRAM-based FPGAs for safety critical applications", Conference on Design, automation and test in Europe (DATE), 2008, pp. 336-341

[19] M. Alderighi et al., "A fault injection tool for SRAM-based FPGAs", 9th IEEE International On-Line Testing symposium, Kos, Greece, July 7-9, 2003, pp. 129-133

[20] M. Alderighi et al., "A tool for injecting SEU-like faults into the configuration control mechanism of Xilinx Virtex FPGAs", 18th IEEE Int. Symposium on Defect and Fault Tolerance in VLSI Systems, November 3-5, 2003, pp. 71-78

On Time, Conflict, Weighting and Dependency Aspects of Assessing the Trustworthiness of Digital Records

Jianqiang Ma*†, Habtamu Abie†, Torbjørn Skramstad* and Mads Nygård*
* *Department of Computer and Information Science*
Norwegian University of Science and Technology, Trondheim, Norway
Email: {majian, torbjorn, mads}@idi.ntnu.no
† *Norwegian Computing Center, Oslo, Norway*
Email: {Jianqiang.Ma, Habtamu.Abie}@nr.no

***Abstract*—In the area of digital records management, the reliability of a digital record's operator varies over time, and consequently affects the trustworthiness of the record. The quality of the reliability of the operator is a measure of the quality of the record's evidential value that is, in turn, a measure of the record's trustworthiness. In assessing the trustworthiness of a record using evidential value, it is essential to combine evidence from various sources, which may be conflicting and/or interdependent. In this paper we describe our research on these problems, and develop a trustworthiness assessment model which addresses these problems and integrates a beta reputation system in combination with a forgetting factor to assess the temporal aspects of the evidential value of an operator, a weighting mechanism to detect and avoid conflicts, and a weighted sum mechanism to combine dependent evidence. Our results show that the integrated model can improve the objective assessment of the trustworthiness of digital records over time using evidential value as a measure of trustworthiness.**

***Keywords*-Trustworthiness Assessment, Trust, Digital Record Management.**

I. Introduction

In the area of digital records management, research on the trustworthiness of digital records is an issue that has received much attention mainly in two areas, Security [1], [2] and Trustworthy Repositories [3], [4]. In our previous work [5], we proposed a complementary method, which assesses the trustworthiness of digital records based on their evidential values using the Dempster-Shafer (D-S) theory of evidence [6]. Four challenges to the assessment model have been identified, i.e., time, conflict, weighting, and dependency aspects. In this paper, we improve our previous model by addressing all four aspects.

This paper describes the time aspect of the trustworthiness assessment model using the reliability of a digital record's operator, since the reliability of an operator varies over time, and, which in turn, affects the assessment result of the trustworthiness of the digital record. Inspired by [7], historical information about the behaviours of operators, which can be obtained from the logs of digital repositories, is used to evaluate their reliability. Correct or incorrect behaviours of operators can be recognised as positive or negative ratings of their reliability. In this way, the widely-researched reputation system mechanism [8]–[10] can be used here for the evaluation of operators' reliability. In this paper, we adopt the beta reputation system [8] to evaluate the reliability of operators, and to create a function to map the evaluated result to the mass function, which can later be used in the D-S theory for the assessment of the trustworthiness of digital records.

Many researchers have criticized the way conflicts are handled in the D-S theory [11], [12]. When using the D-S theory to combine evidential values of evidence around digital records, we have paid attention to these criticisms. We first investigate how to detect conflicts between evidence, and then avoid those conflicts by assigning different weighting, since different evidence may have different importance to the assessment. In addition, we study the combination of evidential values from interrelated evidence, since Dempster's rule of combination is based on independent evidence. We use an alternative approach to combine evidential values from dependent evidence.

As the time aspect deals with evidential values assigned to records' operators, and the other three aspects deal with the combination of evidential values, these four aspects together improve the assessment model by increasing the quality of the evidential values assigned to the evidence, and by improving the way they are combined.

We note that the investigation and evaluation of operators' reliability over time is not fundamentally different from research in the domain of reputation systems [8]. The applicability to the area of trustworthiness assessment of digital records and the integration with the D-S theory are two of the main contributions of this paper. The third is the integration of a conflict detection mechanism, a weighting mechanism, and a dependent evidence combination mechanism in the D-S theory for assessing the trustworthiness of digital records.

The rest of this paper is organised as follows. Section II describes the related work. Section III briefly introduces the Trustworthiness Assessment Model proposed in our previous work. Section IV, V, VI, and VII give an account of our research on the temporal, conflict, weighting and dependency aspects of the trustworthiness assessment model, respectively. Finally, the conclusion and future work are presented in Section VIII.

II. Related Work

Reputation systems allow users to rate on an agent that they have had a transaction with, and use these ratings to assess the reliability of the agent. It was first used in online shopping websites, such as eBay, to assess the trustworthiness of online sellers [13], and was later further developed in the Peer-to-Peer (P2P) networks area to assess the reliability of agents [9], [10]. An extensive survey and overview of trust and reputation systems can be found in [14]. Among these reputation systems, the beta reputation system together with a forgetting factor proposed by Jøsang [8] is capable of assessing one's reliability at a particular time, based on historical information. It has been adopted in many areas [15]. In this paper, we adopt the beta reputation system to evaluate the reliability of records' operators in the trustworthiness assessment model because of its "flexibility and simplicity as well as its foundation on the theory of statistics". The beta reputation system was presented as a stand-alone mechanism in [8]. By mapping the evaluation results to the basic belief assignments (bbas), we integrate it with the D-S theory for the assessment of the trustworthiness of digital records.

The D-S theory of evidence has been applied in many different areas to combine evidence from different sources [16], [17]. One feature of the D-S theory that has received criticisms is its way of handling conflicts [11], [12]. Many alternatives to Dempster's rule of combination have been proposed. The most famous alternatives are conjunctive [18], disjunctive [19] and Yager's [20] combination rules. In the area of belief conflict detection, Josselme et al. [21] proposed a method for measuring the distance between two bbas. Liu [22], however, argued that by only using this distance one cannot distinguish whether two bbas are in conflict or not. Consequently, after formally defining the conflict between two bbas, the author proposed an approach to analyse conflicts, which uses the mass of the combined belief allocated to the empty set before normalisation and the distance between betting commitments (see Section V). Our method of detecting possible conflicts is similar to this method. It differs from it in that our method resolves these conflicts by assigning different weighting to the sources.

Regarding the dependency aspect of different sources, Ferson et al. [23] have done a thorough analysis of dependencies in the D-S theory and probabilistic modelling, including copulas and Fréchet bounds. The weighted sum operator was proposed by McClean and Scotney [16] for the integration of distributed databases. They proved that "the weighted sum operator is a mass function and it is both commutative and associative". It was later adopted by Hong et al. [17] to combine bbas of dependent sensor data in smart homes. They assigned equal weight to the dependent sensors. However, this might not be true in most cases. In this research, we integrate the weighted sum operator into our assessment model to solve the dependency problem by assigning different weighting to sources based on their importance to the assessment.

III. The Trustworthiness Assessment Model

In this section, we briefly introduce the model for the assessment of the trustworthiness of digital records in order to improve the understanding of the aspects addressed in this paper. For detail information about the model, readers are referred to our previous paper [5].

In order to assess the trustworthiness of digital records, we have identified, analysed and specified a list of evidence that shall be stored in the metadata related to digital records [24]. These metadata, named Evidence-Keeping Metadata (EKM), are a subset of the Record-Keeping Metadata [25], but limited only to the metadata, which contain evidence of the trustworthiness or untrustworthiness of digital records. A digital record associated with its EKM can be structured as a tree based on a proposed record's life-cycle model [24].

As shown in Fig. 1, the trustworthiness of a digital record is built up of trustworthiness during different phases of the record's life cycle, which in turn can be categorised by the trustworthiness of various components. Finally, the trustworthiness of each component is assessed using evidence stored in EKM. After receiving the linguistic evidential values of EKM as well as their "trustworthiness hypotheses" (either trustworthy or untrustworthy) from a panel of experts, the assessment model maps them into bbas, and uses these bbas in the D-S theory to assess the trustworthiness of a digital record from the bottom to the top.

In [5], we have proposed this model and identified a number of challenges that still need to be met, to wit the temporal aspect, conflicts, dependencies, and weighted differences among EKM. In the ensuing sections, we describe these challenges, respectively. Note that the model uses bbas assigned to the EKM as basic units for the assessment of the trustworthiness of digital records. Hence, as long as the solutions to these challenges can be mapped to bbas, it is fairly easy to integrate the solutions into the trustworthiness assessment model.

IV. Temporal Aspect

In this section, we describe the temporal aspect of the trustworthiness assessment model briefly introduced above. Inspired by [7], we studied the temporal aspect by looking into the reliability changes of digital records' operators over time. Using long-term observations, the history of digital records as well as the behaviours of records' operators can be logged. On the basis of that historical information, the behavioural patterns of the operators can be learnt, which

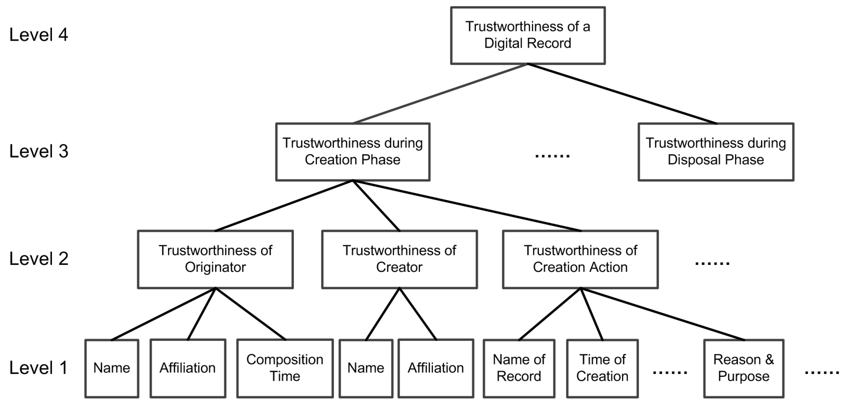

Level 4
Level 3
Level 2
Level 1
Trustworthiness of a Digital Record
Trustworthiness during Creation Phase
......
Trustworthiness during Disposal Phase
Trustworthiness of Originator
Trustworthiness of Creator
Trustworthiness of Creation Action
......
Name
Affiliation
Composition Time
Name
Affiliation
Name of Record
Time of Creation
......
Reason & Purpose
......

Figure 1. Structure of the EKM for the assessment of the trustworthiness of a digital record [24]

provide us with the possibility to evaluate the reliability of the operators' operations on the records.

Good and bad behaviours of an operator P can be documented and used to learn his/her behavioural pattern. A good behaviour means that the operation on a digital record does not compromise the trustworthiness of the record. While a bad behaviour means that the operation decreases the trustworthiness of the record. Good or bad behaviours of P can be discovered through verifications performed by one or more other operators or by verification software. The way of detecting good or bad behaviours in a digital library system is outside the scope of this paper. Instead, this paper focuses on how to learn P's behavioural patterns.

The reliability of P can be predicted using the numbers of good and bad behaviours. It can further be interpreted as the evidential value of P, because both reliability and evidential value present the degree to which P can be used as evidence to prove the trustworthiness of the record P operated on. The reliability - evidential value - of P varies along with the accumulation of the number of behaviours. In this study, the beta reputation system [8] is adopted to assess the reliability - evidential value - of P.

In our case, we map a good behaviour of P to a good feedback on P, and a bad behaviour to a bad feedback. Thus, the evidential value of P can be calculated as:

$$EV(P) = \frac{g+1}{g+b+2} \tag{1}$$

with the restriction that $g, b \geq 0$, where g is the number of good behaviours that have been exhibited by P, and b is the number of bad behaviours that have been exhibited by P.

In order to integrate the evaluated evidential value of P into the trustworthiness assessment model and combine it with evidential values of other EKM, $EV(P)$ needs to be mapped to the bbas defined in the D-S theory. In addition, a "trustworthiness hypothesis" $H_P \in \{true, false\}$ should be specified, where $H_P = true$ or $H_P = false$ mean that P (presented as "Name" of operator in Fig. 1, e.g., "Name of Creator"), as evidence, can be used to prove that P's higher level node is either trustworthy or untrustworthy.

When P exhibits more good behaviours than bad behaviours ($g > b$), it shows that P tends to be reliable, and should be used to prove that its higher level node is trustworthy to a certain degree, thus, $H_P = true$, and vice versa. In the case when $g = b$, $H_P = \phi$, which means that it cannot prove its higher level node is either trustworthy or untrustworthy. When $H_P = false$, instead of using $\frac{g+1}{g+b+2}$, the evidential value of P is assigned as $EV'(P) = 1 - \frac{g+1}{g+b+2} = \frac{b+1}{g+b+2}$, since in this case, it is used to present the degree of its higher level node's untrustworthiness.

It is obvious that both $EV(P)$ and $EV'(P)$ are in the interval $[0.5, 1]$. Since bba is in the interval $[0, 1]$, it is necessary to scale $EV(P)$ and $EV'(P)$ into that interval. Hence, the mapping rules are defined as follows:

$$\text{if } g > b\text{, then } H_P = true \text{ and } \begin{cases} m_P(T) = \frac{2g+2}{g+b+2} \\ m_P(\overline{T}) = 0 \\ m_P(U) = \frac{g-b}{g+b+2} \end{cases} \tag{2}$$

$$\text{if } g < b\text{, then } H_P = false \text{ and } \begin{cases} m_P(T) = 0 \\ m_P(\overline{T}) = \frac{2b+2}{g+b+2} \\ m_P(U) = \frac{b-g}{g+b+2} \end{cases} \tag{3}$$

$$\text{if } g = b\text{, then } H_P = \phi \text{ and } \begin{cases} m_P(T) = 0 \\ m_P(\overline{T}) = 0 \\ m_P(U) = 1 \end{cases} \tag{4}$$

where $g, b \geq 0$ are the numbers of good or bad behaviours of P, as defined in Equation (1).

As presented in Equation (1), the evidential value of P changes with the accumulation of good or bad behaviours. However, from the long-term perspective, the old behaviours may be less relevant in the revelation of P's evidential value, because the behavioural pattern of P might have changed. Therefore, we introduce the forgetting factor δ proposed in [8] into the calculation in order to reduce the impact of the old behaviours on modelling of P's current behavioural pattern. That is, exhibiting G good behaviours at an older time t_1 equals exhibiting $G\delta^{t_2-t_1}$ good behaviours at a more recent time t_2, where $t_2 > t_1$ and $0 \leq \delta \leq 1$. When $\delta = 0$, it forgets every old behaviours, and uses the most recent behaviour to calculate the evidential value of P. In other words, the old behaviours have no impact on the calculation at all. When $\delta = 1$, it never forgets, and the old behaviours have the same impact as the recent behaviours on the calculation of the evidential value of P.

In the calculation of the trustworthiness of a digital record, users of the calculation system can assign a number between 0 and 1 as the forgetting factor, to designate how much the old behaviours should impact the calculation. Since the operator identity and the time of each operation on the digital record are already documented as EKM, together with the forgetting factor assigned by the user, they can be used to calculate the evidential value of an operator for every operation. Hence, even if the same operator operated on the same digital records, his/her evidential value can be different due to the different operation times. For example, operator P created a digital record at time t_1, and later (say after months), migrated the record to another repository at time t_2, the evidential value of P for those two operations could be different, because the reliability of P may be different at time t_1 and time t_2. In this way, the assessed trustworthiness of the digital record will be more accurate than using the same evidential value of P for different operation times.

After adopting the beta reputation system as well as the forgetting factor, the assessment method for the trustworthiness of digital records that reflects the temporal aspect is as follows.

$$\begin{aligned} m_{record}(T) &= m_{creation}(T) \oplus m_{modification}(T) \oplus \\ &\quad m_{migration}(T) \oplus m_{retrieval}(T) \oplus m_{disposal}(T) \\ &= m_{Originator}(T) \oplus m_{Creator}(T) \oplus m_{CreationAction}(T) \\ &\quad \oplus \ldots \oplus m_{DisposalExecutor}(T) \oplus m_{DisposalAction}(T) \\ &= m_{EKM_1}(T) \oplus \ldots \oplus m_{EKM_m}(T) \\ m_{record}(\overline{T}) &= m_{EKM_1}(\overline{T}) \oplus \ldots \oplus m_{EKM_m}(\overline{T}) \\ m_{record}(U) &= m_{EKM_1}(U) \oplus \ldots \oplus m_{EKM_m}(U) \end{aligned} \tag{5}$$

where $\{EKM_1 \ldots EKM_m\}$ stands for all the EKM related to the digital record - nodes in Level 1 in Fig. 1. Particularly, for $EKM_i \in \{EKM_1 \ldots EKM_m\}$, which stands for the reliability of the operator P at a certain time, its bbas are calculated based on Equation (2), (3), and (4).

V. Conflict Aspect

In our research, the D-S theory is used to combine the evidential values from different EKM. However, the way of handling conflicts in the D-S theory has received some criticisms [11], [12]. It is defined in [22] that "a conflict between two beliefs in D-S theory can be interpreted qualitatively as one source strongly supports one hypothesis and the other strongly supports another hypothesis, and the two hypotheses are not compatible." Here we present an example of a conflict that may happen in the trustworthiness assessment model. Suppose there are two experts E_1 and E_2 who assign evidential value for a piece of EKM, say EKM_1. E_1 suggests that EKM_1 is a strong evidence, which supports that its higher level node is trustworthy, hence, bba of EKM_1 assigned by E_1 is $m_1(T) = 0.8, m_1(\overline{T}) = 0$, and $m_1(U) = 0.2$. Actually, experts assign linguistic evidential values to EKM that will later be mapped to numerical evidential values and further bbas of EKM by the trustworthiness assessment model. For simplicity, we only present the mapped bbas here. T and $\overline{T}$ are propositions that EKM_1's higher level node is either trustworthy or untrustworthy. U is the universal set. E_2 also suggests that EKM_1 is a strong evidence, however, it supports that its higher level node is untrustworthy, and the bba assigned by E_2 is $m_2(T) = 0, m_2(\overline{T}) = 0.8$, and $m_2(U) = 0.2$. Using Dempster's rule to combine the assignments from two experts, the result is $m_{12}(T) = 0.44, m_{12}(\overline{T}) = 0.44$, and $m_{12}(U) = 0.12$, which is very near to the average of the two bbas. This is not a good way of handling conflicts, because it hides the conflicting opinions between experts, which may further lead to an imprecise assessment of the final results. Thus, it is necessary to prevent conflicts occuring, and to detect them if they do.

A. *Conflict Detection*

In this section, we demonstrate the method for detecting conflicts from different sources. This method is proposed in [22], which uses two indicators to detect conflicts between two bbas, i.e., the combined belief allocated to the empty set before normalisation ($m_\oplus(\phi)$) and the distance between their betting commitments.

As defined in [26], the pignistic probability function $BetP_m$ associated to bba m on the universe is:

$$BetP_m(\omega) = \sum_{A \subseteq U, \omega \in A} \frac{1}{|A|} \frac{m(A)}{1 - m(\phi)} \tag{6}$$

where $|A|$ is the cardinality of subset A on U. The transformation from bba m to $BetP_m$ is called the pignistic transformation. It can be further extended to 2^U that $BetP_m(A) = \sum_{\omega \in A} BetP_m(\omega)$. $BetP_m(A)$, referred to as the *betting commitment to* A in [22], presents the total mass value that A can carry.

Thus, the distance between two betting commitments to A from two sources is considered as the maximum of the differences between their betting commitments to all the subsets, defined as

$$\Delta BetP_{m_1}^{m_2} = max_{A \subseteq U}(|BetP_{m_1}(A) - Bet_{m_2}(A)|) \quad (7)$$

While the combined belief allocated to the empty set before normalisation ($m_{\oplus}(\phi)$) in the D-S theory is defined as

$$m_{\oplus}(\phi) = \sum_{B,C \subseteq U, B \cap C = \phi} m_1(B)m_2(C) \quad (8)$$

It is discussed in [22] that the sole use of either $m_{\oplus}(\phi)$ or $\Delta BetP_{m_1}^{m_2}$ cannot detect the conflicts between two beliefs, they should be used together in order to detect conflicts. Thus, based on the definition presented above, two beliefs m_1 and m_2 are defined as in conflict if and only if both $\Delta BetP > \epsilon$ and $m_{\oplus}(\phi) > \epsilon$, where ϵ is the factor that indicates the tolerance of conflict. The higher ϵ is, the more tolerance of conflict the system is.

Let us see the two beliefs in the example above where

$$\begin{aligned} &m_1(T) = 0.8, && m_1(\overline{T}) = 0, && m_1(U) = 0.2, \\ &m_2(T) = 0, && m_2(\overline{T}) = 0.8, && m_2(U) = 0.2. \end{aligned}$$

Using Equation (6), (7), and (8), the $\Delta BetP_{m_1}^{m_2}$ and $m_{\oplus}(\phi)$ of m_1 and m_2 are calculated as

$$\begin{aligned} &BetP_{m_1}(T) = 0.9, && BetP_{m_1}(\overline{T}) = 0.1, && BetP_{m_1}(U) = 1, \\ &BetP_{m_2}(T) = 0.1, && BetP_{m_2}(\overline{T}) = 0.9, && BetP_{m_2}(U) = 1, \\ &\Delta BetP_{m_1}^{m_2} = 0.8, && m_{\oplus}(\phi) = 0.64. \end{aligned}$$

Thus, if assigning $\epsilon = 0.6$, the opinions from those two experts will be recognised as in conflict, while if assigning $\epsilon = 0.7$, they will not be seen as in conflict, even though $\Delta BetP_{m_1}^{m_2} > \epsilon$.

When conflicts occur, it does not necessarily mean that the use of D-S theory to assess the trustworthiness of digital records is wrong, but simply that, due to the conflicts in beliefs from difference sources, the results may be imprecise. Thus, together with the assessment results, the assessment model will also inform users that conflicts between different elements have occurred during the assessment. Users can consider this information together with the assessment result to determine whether a digital record is trustworthy or not.

B. Conflict Avoidance

By analysing the sources of conflicts in the assessment model, it can be found that conflicts may exist among experts' opinions, EKM, components, and life-cycle phases of a digital record. For a certain piece of EKM, due to the differences in observations or experience, experts can have different opinions on its use as evidence, therefore, different evidential values and trustworthiness hypotheses may be assigned to it, which may further induce conflicts among bbas assigned to this piece of EKM. It is also similar for other elements in the assessment model, such as EKM, components, and life-cycle phases.

Notice that until now, all the elements in the assessment model have been recognised as equally important. However, it is more realistic to assign different weighting to different elements. For instance, some of the experts may have more knowledge or experience than others, hence, their opinions deserve to be considered as more important than others'. In addition, by assigning different weighting, many conflicts can be avoided, because beliefs from less weighted sources will be discounted.

In the following section, we discuss the weighting difference as well as how to discount bbas in the D-S theory.

VI. Weighting Aspect

Weighting difference can be used to differentiate the importance among different elements in the assessment model. Also, as presented in the section above, it is a way to avoid conflicts from different sources.

The discounting method in the D-S theory [6] is introduced to assign weighting to elements, as shown in Equation (9).

$$m_1^{discounting}(A) = \begin{cases} \alpha m_1(A) & \text{if } A \neq U \\ \alpha m_1(U) + (1-\alpha) & \text{if } A = U \end{cases} \quad (9)$$

where α ($0 \leq \alpha \leq 1$) is the weighting assigned to $m_1(A)$.

Recall the example introduced in Section V, suppose expert E_1 has more knowledge and experience than expert E_2, and they are assigned with different weighting as $\alpha_1 = 0.9$ and $\alpha_2 = 0.4$, respectively. Using the discounting method in Equation (9), new bbas of the two experts are

$$\begin{aligned} &m_1'(T) = 0.72, && m_1'(\overline{T}) = 0, && m_1'(U) = 0.28, \\ &m_2'(T) = 0, && m_2'(\overline{T}) = 0.32, && m_2'(U) = 0.68. \end{aligned}$$

Then apply Equation (6), (7), and (8), $\Delta BetP_{m_1}^{m_2}$, and $m_{\oplus}(\phi)$ of the two bbas can be calculated as

$$\begin{aligned} &BetP'_{m_1}(T) = 0.86, && BetP'_{m_1}(\overline{T}) = 0.14, && BetP'_{m_1}(U) = 1, \\ &BetP'_{m_2}(T) = 0.34, && BetP'_{m_2}(\overline{T}) = 0.66, && BetP'_{m_2}(U) = 1, \\ &\Delta BetP'^{m_2}_{m_1} = 0.52, && m'_{\oplus}(\phi) = 0.23. \end{aligned}$$

In this case, if the conflict tolerance is still set to $\epsilon = 0.6$ as in Section V-A, bbas from those two experts will no longer be recognised as in conflict, hence conflict avoided.

Another issue arising together with the use of weighting difference is how the weighting of each element can be assigned. In the absence of a completely objective weighting assignment method, Wang and Wulf [27] use the Analytic Hierarchy Process (AHP) to identify the importance of different elements. This approach can also be used in the trustworthiness assessment model to assign different weighting to different elements. However, due to limitations of space, we will not discuss it any further here. For details of this approach, readers can refer to [27].

After assigning different weighting to different elements, the calculation of the trustworthiness of a digital record (similar to Equation (5)) is

$$
\begin{aligned}
m_{record}(T) &= m^{discounting}_{creation}(T) \oplus m^{discounting}_{modification}(T) \oplus \\
&\quad m^{discounting}_{migration}(T) \oplus m^{discounting}_{retrieval}(T) \oplus m^{discounting}_{disposal}(T) \\
&= m^{discounting}_{Originator}(T) \oplus m^{discounting}_{Creator}(T) \oplus m^{discounting}_{CreationAction}(T) \\
&\quad \oplus \ldots \oplus m^{discounting}_{DisposalExecutor}(T) \oplus m^{discounting}_{DisposalAction}(T) \\
&= m^{discounting}_{EKM_1}(T) \oplus \ldots \oplus m^{discounting}_{EKM_m}(T) \\
m_{record}(\overline{T}) &= m^{discounting}_{EKM_1}(\overline{T}) \oplus \ldots \oplus m^{discounting}_{EKM_m}(\overline{T}) \\
m_{record}(U) &= m^{discounting}_{EKM_1}(U) \oplus \ldots \oplus m^{discounting}_{EKM_m}(U)
\end{aligned}
$$

VII. Dependency Aspect

In the trustworthiness assessment model, dependencies may exist in some pieces of the EKM. For instance, “Name of the Creator” and “Affiliation of the Creator” of a digital record are interrelated. Since Dempster’s rule of combination is based on independent evidence, it is not suitable for the combination of evidence from interrelated EKM. Thus, an alternative approach should be found to combine dependent EKM in the trustworthiness assessment model.

In this work, we adopt the weighted sum operator [16] as the alternative approach for combining dependent EKM for its applicability. Because the output of the weighted sum operator is still a basic belief assignment, it can be easily integrated into the trustworthiness assessment model.

The weighted sum operator ($\widehat{\oplus}$) is defined in Equation (10)

$$
m_1 \widehat{\oplus} m_2(A) = \frac{w_1}{w_1 + w_2} m_1(A) + \frac{w_2}{w_1 + w_2} m_2(A), \quad \text{where } w_1, w_2 \geq 0. \tag{10}
$$

In [17], when using the weighted sum operator, all dependent elements are equally weighted. While in our research, different weighting is assigned in the combination of dependent EKM. Note that the weighting here is different from the weighting in Section VI. The weighting in Section VI presents the importance of one element in the assessment of the trustworthiness of a digital record, whereas the weighting here denotes the different importance of dependent elements in the determination of their combined result, for example, in the assessment of the trustworthiness of a creator. In the case where the creator creates the record on behalf of his/her organisation, the affiliation should be recognised as more important. Thus, in the adoption of the weighted sum operator, the affiliation is heavily weighted, say $w_{name} = 2, w_{aff.} = 10$, for instance. In another case, the creator only creates a record for personal use, where the affiliation is recognised as less important, and thus, is less weighted, say $w_{name} = 10, w_{affiliation} = 4$.

To differentiate weighting for the weighted sum operator, the AHP method as mentioned in Section VI can also be used. In addition, users of the trustworthiness assessment model may want to assign weighting to EKM based on their different use.

After integrating the weighted sum operator into the trustworthiness assessment model, the calculation of the trustworthiness of a digital record (similar to Equation (5)) is

$$
\begin{aligned}
m_{record}(T) &= m^{discounting}_{EKM_1}(T) \oplus \ldots \oplus (m_{EKM_i} \widehat{\oplus} \\
&\quad m_{EKM_j}(T))^{discounting} \oplus \ldots \oplus m^{discounting}_{EKM_m}(T) \\
m_{record}(\overline{T}) &= m^{discounting}_{EKM_1}(\overline{T}) \oplus \ldots \oplus (m_{EKM_i} \widehat{\oplus} \\
&\quad m_{EKM_j}(\overline{T}))^{discounting} \oplus \ldots \oplus m^{discounting}_{EKM_m}(\overline{T}) \\
m_{record}(U) &= m^{discounting}_{EKM_1}(U) \oplus \ldots \oplus (m_{EKM_i} \widehat{\oplus} \\
&\quad m_{EKM_j}(U))^{discounting} \oplus \ldots \oplus m^{discounting}_{EKM_m}(U)
\end{aligned}
$$

where EKM_i and EKM_j are interrelated EKM, combined using the weighted sum operator.

VIII. Conclusion and Future Work

In this paper, we have described the time, conflict, weighting and dependency aspects in the assessment of the trustworthiness of digital records. We used the beta reputation system together with a forgetting factor to evaluate the reliability of records’ operators over time, which is used as the evidential value of the operators and is the value assigned to a basic belief assignments (bba), which is integrated into the assessment model. Proceeding on the basis of assigned conflict toleration, we detected conflicts between the evidence gained from different sources by examining two factors, the mass of the combined beliefs allocated to the empty set before normalisation, and the distance between betting commitments. Discounting is used to assign weighting differences and avoid conflicts in Evidence-Keeping Metadata (EKM). Finally, we used the weighted sum operator to combine dependent evidence. Because solutions to these four problems are all based on changes in the bbas, they are easily integrated into our model for the assessment of the trustworthiness of digital records.

Our results show that by adopting and carefully revising the reputation systems and the Dempster-Shafer (D-S) theory, they can be integrated in our model and improve the objective assessment of the trustworthiness of digital records over time.

In future work, we shall look into the verification and validation of the trustworthiness assessment results.

REFERENCES

[1] H. M. Gladney, "Trustworthy 100-year digital objects: Evidence after every witness is dead," *ACM Transaction on Information System (TOIS)*, vol. 22, no. 3, pp. 406–436, 2004.

[2] H. M. Gladney and R. A. Lorie, "Trustworthy 100-year digital objects: durable encoding for when it's too late to ask," *ACM Transaction on Information System (TOIS)*, vol. 23, no. 3, pp. 299–324, 2005.

[3] Center for Research Libraries, "Trustworthy repositories audit & certification: Criteria and checklist," Jul. 2008, [accessed 01-Apr-2009]. [Online]. Available: http://www.crl.edu/PDF/trac.pdf

[4] Consultative Committee for Space Data Systems, "Reference model for an open archival information system (OAIS)," National Aeronautics and Space Administration, Jan. 2002.

[5] J. Ma, H. Abie, T. Skramstad, and M. Nygård, "Assessment of the trustworthiness of digital records," in *Fifth IFIP WG 11.11 International Conference on Trust Management*, ser. IFIP Advances in Information and Communication Technology (AICT). Springer, Jun. 2011, pp. 300–311.

[6] G. Shafer, *A Mathematical Theory of Evidence*. Princeton University Press, 1976.

[7] B. Alhaqbani and C. Fidge, "A time-variant medical data trustworthiness assessment model," in *Proceedings of the 11th international conference on e-Health networking, applications and services*, 2009, pp. 130–137.

[8] A. Jøsang and R. Ismail, "The beta reputation system," in *Proceedings of the 15th Bled Electronic Commerce Conference*, vol. 160, 2002, pp. 17–19.

[9] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina, "The eigentrust algorithm for reputation management in p2p networks," in *Proceedings of the 12th International World Wide Web Conference*. ACM Press, 2003, pp. 640–651.

[10] L. Xiong and L. Liu, "Peertrust: Supporting reputation-based trust for peer-to-peer electronic communities," *IEEE Transaction on Knowledge and Data Engineering, IEEE Transactions on*, vol. 16, no. 7, pp. 843–857, 2004.

[11] P. Smets, "Analyzing the combination of conflicting belief functions," *Information Fusion*, vol. 8, no. 4, pp. 387–412, 2007.

[12] L. A. Zadeh, "A simple view of the dempster-shafer theory of evidence and its implication for the rule of combination," *AI Magazine*, vol. 7, no. 2, pp. 85–90, 1986.

[13] P. Resnick and R. Zeckhauser, "Trust among strangers in internet transactions: Empirical analysis of ebay's reputation system," *Advances in Applied Microeconomics: A Research Annual*, vol. 11, pp. 127–157, 2002.

[14] A. Jøsang, R. Ismail, and C. Boyd, "A survey of trust and reputation systems for online service provision." *Decision Support Systems*, vol. 43, no. 2, pp. 618–644, 2007.

[15] Y. L. Sun, Z. Han, W. Yu, and K. Liu, "Attacks on trust evaluation in distributed networks," in *Information Sciences and Systems, 2006 40th Annual Conference on*. IEEE, 2006, pp. 1461–1466.

[16] S. McClean and B. Scotney, "Using evidence theory for the integration of distributed databases," *International Journal of Intelligent Systems*, vol. 12, no. 10, pp. 763–776, Oct. 1997.

[17] X. Hong, C. D. Nugent, M. D. Mulvenna, S. I. McClean, B. W. Scotney, and S. Devlin, "Evidential fusion of sensor data for activity recognition in smart homes," *Pervasive and Mobile Computing*, vol. 5, no. 3, pp. 236–252, 2009.

[18] P. Smets, "The combination of evidence in the transferable belief model," *IEEE Transaction on Pattern Analysis and Machine Intelligence*, vol. 12, pp. 447–458, May 1990.

[19] D. Didler Dubois and H. Prade, "Representation and combination of uncertainty with belief functions and possibility measures," *Computational Intelligence*, vol. 4, no. 3, pp. 244–264, 1988.

[20] R. R. Yager, "On the dempster-shafer framework and new combination rules," *Information Sciences*, vol. 41, no. 2, pp. 93–137, 1987.

[21] A.-L. Jousselme, D. Grenier, and É. loi Bossé, "A new distance between two bodies of evidence," *Information Fusion*, vol. 2, no. 2, pp. 91–101, 2001.

[22] W. Liu, "Analyzing the degree of conflict among belief functions," *Artificial Intelligence*, vol. 170, no. 11, pp. 909–924, 2006.

[23] S. Ferson, R. Nelsen, J. Hajagos, D. Berleant, J. Zhang, W. Tucker, L. Ginzburg, and W. Oberkampf, "Dependence in probabilistic modeling, dempster-shafer theory, and probability bounds analysis," *Albuquerque, New Mexico: Sandia National Laboratories*, pp. 1–141, 2004.

[24] J. Ma, H. Abie, T. Skramstad, and M. Nygård, "Development and validation of requirements for evidential value for assessing trustworthiness of digital records over time," *Journal of Information*, (to appear).

[25] National Archives of Australia, "Australian government recordkeeping metadata standard," Tech. Rep., Jul. 2008, [accessed 01-Apr-2009]. [Online]. Available: http://www.naa.gov.au/Images/AGRkMS_Final%20Edit_16%2007%2008_Revised_tcm2-12630.pdf

[26] P. Smets, "Decision making in the tbm: the necessity of the pignistic transformation," *International Journal of Approximate Reasoning*, vol. 38, no. 2, pp. 133–147, 2005.

[27] C. Wang and W. A. Wulf, "Towards a framework for security measurement," in *20th NIST-NCSC National Information Systems Security Conference*, 1997, pp. 522–533.

Specification and Verification of the Triple-Modular Redundancy Fault Tolerant System using CSP

Lanfang Tan, Qingping Tan, Jianli Li
School of Computer
National University of Defense Technology
Changsha, China
tlf1022@126.com, eric.tan.6508@gmail.com, ljl_003@163.com

***Abstract*—A case study on the application of Communicating Sequential Processes (CSP) to the specification and verification of fault tolerant systems is presented. The Triple-Modular Redundancy (TMR) mechanism is a classical design technique for tolerating hardware errors. By specifying the behavior of the faultless module as a CSP process, the behavior of TMR system suffering from hardware errors can be verified as a refinement of the one of the faultless module.**

***Index Terms*—TMR System; fault tolerance; verification; CSP**

I. INTRODUCTION

Fault tolerance is generally accomplished by using redundancy in hardware, software or combination. There are three basic types of redundancy in hardware and software: spatial redundancy, time redundancy, and hybrid. TMR scheme has been one of the most popular fault tolerant mechanisms using spatial redundancy [1]. In TMR systems, computions are replicated into three modules running in parallel and their outputs are voted using a voter circuit. A single fault in any of the redundant modules will not produce an error at the output as the voter will select the correct result from the two working modules and recover the fault. There are numerous examples of dependable systems using the TMR technique [2].

Though the principle of TMR fault tolerant system is straightforward, evaluating system's behavior in the presence of faults constitutes another significant problem, especially for the complicated systems[8][9][10]. In other words, it was not possible to determine whether the behavior described in these requirements would provide the desired level of fault tolerance. Fault injection techniques have emerged as an important method for evaluating fault tolerant systems. However, they cannot cover all fault scenarios. Therefore, they cannot guarantee that all fault consequence has been investigated. This motivates us to explore a formal verification approach that targets a complete validation.

In [3], temporal logic of actions (TLA) [4] is used to specify and validate TMR fault tolerant system. The programs running on three processors are represented as transition systems. Physical faults in the system are modelled as a set of "fault actions" which perform state transformations in the same way as the other program actions. Assuming that errors will not occur on two modules synchronously, the fault tolerance property of TMR system can be verified as the refinement of a non-fault-tolerant program.

In this paper, we propose an approach for the formal verification of TMR fault tolerant systems using CSP [5], which is a member of a class of formal methods known as process algebras. By specifying the property of a faultless module with a CSP process, we prove that TMR fault tolerance system can still satisfy the property in spite of hardware errors. The verification process can be absolutely automatic based on model checking support tool FDR2 [6] of CSP.

The rest of this article is organised as follows. The next section briefly introduces the language of CSP used in this paper; Section III considers the specification for a faultless module; Section IV describes the TMR fault tolerance system suffering from hardware errors; Section V verifies the effectiveness of TMR mechanisms and discusses the use of model checking tool FDR as an automated means of verifying the fault-tolerant design; Section VI concludes with some remarks on the use of CSP in formal verification of fault tolerant systems.

II. THE LANGUAGE OF CSP

CSP is a language where processes proceed from one state to another by engaging in (instantaneous) events [5]. A process is a component that encapsulates some data structures and algorithms for manipulating that data. It interacts with enviroment through synchronised message passing along channels, or events. The set of all events in the interface of a process P, written $\alpha.P$, is called its alphabet. However, the interface events are not as autonomous actions under the control of a single process but intended as synchronization between the participating processes.

The language of CSP used in this paper is described in Figure 1, which is defined by the following pseudo Backus-Naur form definitions. In Figure 1, c denotes an event, A is a set of events and b is a Boolean expression. The Skip is the process that does nothing but terminates successfully. The prefix process c -> P is ready to perform event c and waits until the environment prepares well event c. Once the event c is performed, the subsequent behaviour of c -> P will be that of process P. In the sequential composition P; Q, the combined process firstly behaves as P and then Q becomes active immediately after the termination of P. The internal choice P $\sqcap$ Q waits to perform events that either P or Q is ready to engage in. Once an event of a component is performed, the subsequent behaviour is given by this component. Selecting either P or Q depends on its internal environment.

The parallel combination P|A|Q only synchronises on events in A, and interleaves on all other events. The hiding operator P \ A makes a given set of events in A internal, thus beyond the control of its environment. The prefix choice a: A-> Pa is ready to perform any event from set A until one is chosen. Pa is its subsequent behaviour, which is dependent on the chosen event a. A process can be defined to allow the input on channel *in* of any item *x* in a set *M*, and the value *x* determines the subsequent behavior [5]:

$$in?x{:}M \rightarrow Q(x) \cong a{:}in.M \rightarrow Pa$$

where the set $in.M=\{in.m \mid m\in M\}$ and $Pin.m=Q(m)$ for every $m \in M$. The atomic synchronization events here are of the form *in.m*. The output prefix has the form $out!x \rightarrow P$ and this is simply a shorthand for $out!x \rightarrow P$.

The indefinite loop process P* repeats the actions of P after the successful termination of P. The condition operator if b then P else Q fi selects either P or Q according to the Boolean expression b.

P := c -> P | P ; P | P $\sqcap$ P |

P |A| P | P \ A | a: A -> P_a |

P* | if b then P else P

Figure 1. The CSP operators

III. SPECIFICATION OF A FAULTLESS MODULE

In this section, we want to model the faultless module as a CSP process that represents the general computing model. As Figure 2 illustrates, a faultless module can be abstracted as a computation process, which consists of a processor and a memory. Assume that the program to be performed in the module is deterministic and sequential, consisting of data segment and text segment. During program executing, the processor either executes an instruction in text segment to change the content of data segment, or issues write operation to the memory.

Therefore, the processor can be abstracted as a function *d*= *funp*(*l*), where *d* denotes the content of data segment and *l* denotes the next instruction to be executed. For each input program, the mapping function *funp* is determined. The write operations can be performed just as certain instructions are executed, such as store instructions. So we specify the behaviour that the processor needs write data *d* to the memory by an assertion NeedWrite(*d*), whose value is true if and only if the store instruction is performed. When the processor issues write operations defined as Output (*d*), the memory updates data segment, which is represented by a function Update(*d*).

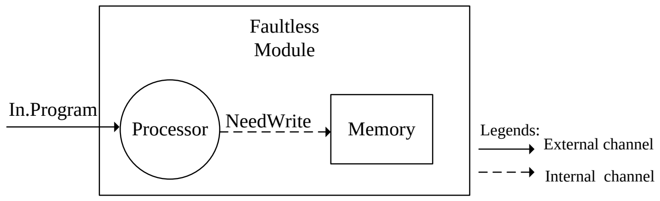


Figure 2. The faultless module

As is mentioned above, we can illustrate the behaviour of the faultless module in the CSP language in Table I.

TABLE I SPECIFICATION FOR THE FAULTLESS MODULE

1	Faultless Module: = Processor | { Output(*d*) } | Memory \ { Output(*d*) }
2	Processor: =*in*? Program → (if (Exited(*l*)) then Skip else (if (NeedWrite(*d*)) then *out*! Output (*d*) else *funp* (*l*) fi.)fi.)* →Processor
3	Memory: = (*in*? Output(*d*) → Update (*d*)) → Memory

The faultless module is encoded as a parallel combination construction, with two processes synchronizing on the event Output (*d*). The CSP process is defined by the expression on the right-hand-side of the definition ":=" symbol. Processor specifies the module initially inputs a program through channel *in* and then executes in terms of the input program. During executing, the Processor either exits the program and prepares for the next input program, or continues executing depending on the Boolean expression Exited(*l*). When Processor output *d* through channel in, Memory updates the values of the data segment and then returns to its initial state preparing for the next write operation.

IV. SPECIFICATION OF THE TMR SYSTEM

The TMR system allows parallel execution of the three modules on three processors thereby providing tolerance of certain permanent and transient hardware faults. Suppose that the hardware faults only occur on processors and memories are protected by error correcting codes (ECC) mechanism [7].

The principle of the TMR system is shown in Figure 3. It works as follows. Once a program is input, the three processors start executing. When a processor needs writing data in memory, it issues a signal "Needwrite" to the Voter. When all signals to write memory arrive, the voter chooses the correct data to be written into memory and then sends the answer message to the three processors.

We may specify the behavior of the Voter by the following CSP process in Table II. It is a sequential composition, where the process first waits the signals to write data and then choose the correct data. The notations of answer message are defined as follows:

1) Ack !"0": denotes the data to be written into memory if three processors are equal.
2) Ack !"i"(i=1, 2,3): denotes the data to be written of processor_i is not equal to the data of the other two

processors.

3) Ack!"4": denotes the data to be written of three processors are not equal to each other. Of course, this case is not possible unless the Voter works abnormally.

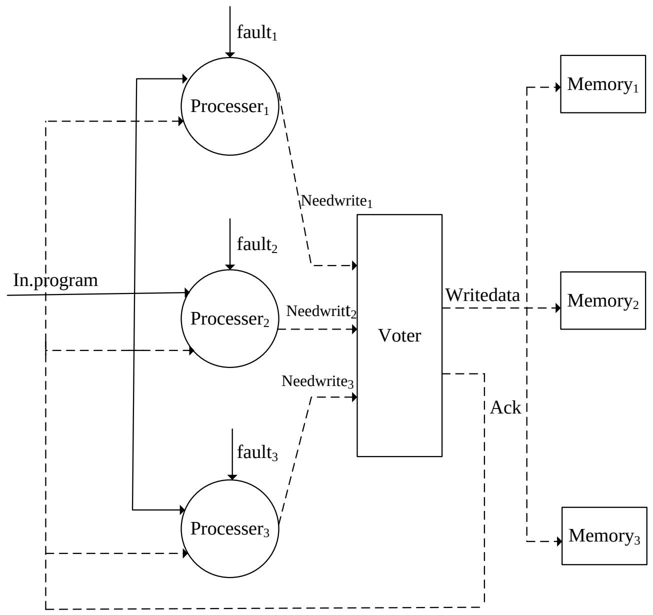


Figure 3. The TMR system

TABLE II SPECIFICATION FOR THE VOTER OF THE TMR SYSTEM

1	Voter: = (in? Needwrite$_1$ $\longrightarrow$ SKIP ||in? Needwrite$_2$ $\longrightarrow$ Skip || in?Needwrite$_3$ $\longrightarrow$ SKIP);
2	If (d1==d2 and d2==d3) then (Writedata!(d1) || Ack ! 0)
3	else if (d1==d2 and d2!=d3) then (Writedata!(d2) || Ack ! 3)
4	else if (d1==d3 and d2!=d3) then (Writedata!(d1) || Ack ! 2)
5	else if d3==d2 and d1!=d3) then (Writedata!(d3) || Ack ! 1)
6	else (Ack !4)
7	$\longrightarrow$ Voter

Similarly, the behavior of the Modules can be expressed as Processor$_i$ and Memory$_i$ in CSP language, which is shown in Table III. The Process$_i$ is designed as a sequential composition, which begins with an internal choice waiting to perform events. If a fault occurs, the data values will be corrupt, which is expressed by a function Wrong (d). We can speculate that once a fault occurs, it will be responded by the answer message and then the data will be recovered, which is described as Recover(d). Memory$_i$ process is similar to the Memory process of the faultless module.

TABLE III SPECIFICATION FOR THE PROCESSOR OF THE TMR SYSTEM

1	Processor$_i$:= (in?fault$_i$ $\longrightarrow$ (Wrong$_i$ (d) $\longrightarrow$ SKIP) $\sqcap$ (in? Program $\longrightarrow$ if (Exited(l)) then Skip else
2	(if (NeedWrite$_i$) then out! Output$_i$(d)
3	else fun_p(l) fi.)*$\longrightarrow$ SKIP) $\sqcap$
4	(Ack?x $\longrightarrow$(if in?Ack.x= i then Recover$_i$(d)
5	else SKIP))
6	$\longrightarrow$Processor$_i$ (i=1,2,3)
7	Memory$_i$: = (In? Output(d) $\longrightarrow$ Update (d)) $\longrightarrow$ Memory$_i$ (i=1,2,3)

As mentioned above, the TMR system can be illustrated in Table IV. It is designed as a parallel combination construction containing two processes. The first process is that three processors and Voter synchronize on events "Output" and "Ack". The second process describes the synchronizing between the first process and the three memories on event "Writedata". All the events are internal to the TMR system, thus the hiding operator is adopted.

TABLE IV SPECIFICATION FOR THE TMR SYSTEM

1	TMR System := (Processor$_1$ || Processor$_2$ || Processor$_3$
2	|{ Output$_1$, Output$_2$, Output$_3$, Ack } | Voter)
3	|{Writedata}|
4	(Memory$_1$ || Memory$_2$|| Memory$_3$) \
5	\{ Output$_1$, Output$_2$, Output$_3$,WriteData, Ack }

V. VERIFICATION OF FAULT TOLERANCE

The verification of fault tolerance for the TMR system amounts to showing that the behaviour of the TMR system suffering from hardware faults is a refinement of the behavior of a faultless module, as stated in the following lemma:

Lemma 1: Faultless Module ≦ TMR System

Proof: One straightforward way to show the refinement relationship is to apply semantics preserving transformation upon the process definition based on the algebraic rules associated with CSP operators [5]. This approach explores or enumerates manually all possible states of a process defined by parallel combination. Also, since here we target at the behavioral properties rather than the functional property of the process, an abstract version of TMR system which ignores the functional aspect including values, variables, and Boolean expressions can be obtained. Table V shows the proof for Lemma1.

TABLE V THE PROOF OF THE TMR SYSTEM

1	TMR system= (Process$_i$|{ NeedWrite$_i$, Ack } | Voter) |{WriteData}| Memory$_i$\{ NeedWrite$_i$, Ack,WriteData } i=1,2,3	(1)
2	Fill the definitions of Processor$_i$,Memory$_i$ and Voter in (1)	
3	Apply the following algebraic laws (2) to simply (1)	
4	P || (R; Q) = (P || R); (P|| Q)	(2)
5	Apply the following algebraic laws (2) and (3) to simply (1)	
6	(P $\sqcap$ Q)|| R= (P || R) $\sqcap$ (Q|| R)	(3)
	Assume that two faults cannot occur synchronously, (1) can be simplified as (4)	
8	Faultless Module$_{i=1,2}$; Ack?x->SKIP|| (fault$_i$? $\longrightarrow$ (Wrong (d) $\longrightarrow$SKIP)|| Voter ;	(4)
9	Assume that the Recover(d) can restore the program effectively, apply the following laws to simply(4)	
10	SKIP ; P=P (a$\longrightarrow$P);Q=a$\rightarrow$(P;Q)	
11	(4) can be simplified as following :	
12	Faultless Module$_{i=1,2,3}$	

However, it is too laborious to verify the property of the TMR system manually. Fortunately, the FDR model checking tool [6] can be used to verify the above lemma automatically. To verify whether the TMR system suffering from hardware fault is a refinement of the faultless module, the failures-divergence model [6] in which the possible behaviours of a process are denoted by a set called its failures-divergence is adopted. A process Q is a refinement of another process P, written as P [FD= Q, if (and only if) the failures-divergence set of the former is contained in the failures-divergence set of the latter. Firstly, the system is defined as a CSP process. Then, it is translated into the input language of FDR. A listing of the FDR2-compatible source can be found in the appendix. While a detailed introduction to the semantics of FDR is beyond the scope of this paper, the appendix specifies the TMR system obtained by translating the CSP process into the input language of FDR.

VI. CONCLUSION AND FUTURE WORK

This paper has shown how the FDR refinement checker for CSP can be used to specify and verify the fault tolerant system. Firstly, a faultless module that is abstracted as a computation model is encoded as a CSP process. Then the TMR system suffering from hardware errors is illustrated in CSP. By verifying the TMR system is a refinement of a faultless module, the fault tolerance property of the TMR system can be verified. Moreover, by the model checking tool FDR, the verification is performed automatically.

The work with more similarity to ours is described in [3], but ours is much more general and practical. Our work just needs to encode the system in CSP processes, and then the verification can be performed automatically by FDR. FDR searches the state space of the system until it either finds an undetected error or exhausts the state space. This search is automatic in the sense that it does not require user guidance once the system has been modeled in CSP. On the contrary, the work in [3] validates the correctness of fault tolerant systems using axioms and proof rules. It can only be used by experts who are well-drained in logical reasoning and have considerable experience. Thus it is not practical.

However, we only focus on the write operation between the processor and the memory. In order to simplify the process description, the read operation that the processor reads data from the memory is not considered. In general, it is just an early work. In the future, we will investigate the read operation and apply the method to some complicated system.

APPENDIX

{- The idea of this script is to prove that the TMR system is a refinement of the behavior of a faultless module -}
-- Event definitions
Output (d) =yes|no
Tags_ack= {0, 1, 2, 3}
NeedWrite= NeedWrite$_1$| NeedWrite$_2$| NeedWrite$_3$
Data=d$_1$|d$_2$|d$_3$
--channel declarations
Channel Processor_in, Proceesor_in$_1$, Proceesor_in$_2$, Proceesor_in$_3$
channel Memory_in, Memory_in$_1$, Memory_in$_2$, Memory_in$_3$: Output
channel Voter_ack : Tags_ack
channel Voter_in: NeedWrite
channel Voter_WriteData
-- The specification is for the processor of the faultless module
Processor=Processor_in? Program-> (if (NeedWrite) out! Output (d).yes else funp(l) fi.) ->Processor
-- The specification is for the memory of the faultless module
Memory = (Memory_in? Output(d).yes > Update (d)) -> Memory
-- The specification is the faultless module
Faultless Module= (Processor [| {| Output(d)|} |] Memory) \ {| Output(d) |}
-- The specification is the Voter of the faultless module
Voter: = (Voter_in? Needwrite$_1$ --> SKIP || Voter_in? Needwrite$_2$ --> SKIP || Voter_in?Needwrite$_3$ -> SKIP);
if (d$_1$==d$_2$ and d$_2$==d$_3$) then (Voter_WriteData!d$_1$ || Voter_ack ! 0)
else if (d$_1$==d$_2$ and d$_2$! =d$_3$) then (Writedata! d2 || Voter_ack! 3)
else if (d$_1$==d$_3$ and d$_2$! =d$_3$) then (Writedata! d1 || Voter_ack! 2)
else d$_3$==d$_2$ and d$_1$!=d$_3$) then (Writedata!d$_3$ || Voter_ack ! 1)
--> Voter
-- The specification is the Processor1 of the faultless module, it is similar to Processor2 and Processor3
Processor$_i$ = ((in?fault$_i$ -> (Wrong (d$_i$) -> SKIP) [] In? Program-> (if (NeedWrite$_i$) Out! Output$_i$(d$_i$)
else funp(l) fi. --> SKIP);
Ack? ->(if in?ack.x= i then Recover$_i$(d)
else SKIP);
Processor$_i$ (i=1,2,3)
Memory$_i$= (In? Output(d) -> Update (d)) -> Memory$_i$ (i=1,2,3)
-- Finally we put it all together, and hide internal communication
TMR System = (Processor$_1$ || Processor$_2$|| Processor$_3$ |{Output$_1$, Output$_2$, Output$_3$,Ack}| Voter) |{Writedata}|(Memory$_1$|| Memory$_2$ || Memory$_3$)\{ Output$_1$, Output$_2$, Output$_3$,Ack, Writedata}
--The Specification of Faultless Module ≦TMR System
assert Faultless Module [FD= TMR System

REFERENCES

[1] Kang G. S. and Hagbae K., "A Time Redundancy Approach to TMR Failures Using Fault-State Likelihoods," IEEE Trans. on Computers, vol. 43, pp. 1151-1162, Oct. 1994.
[2] Siewiorek D. P. and R. S. Swarz, "Reliable Computer Systems: Design and Evaluation," Digital Press, 1992.
[3] Liu Z.M. and Joseph M., "Specification and verification of fault tolerance, timing and scheduling," ACM Trans. on Programming language and systems, vol. 21, pp. 46-49, Jun. 1999.
[4] Lamport L, "The temporal logic of actions," ACM Trans. on Programming language and systems, vol. 16, pp. 872-923, Nov. 1994.
[5] Hoare C.A.R, "Communicating Sequential Processes," Prentice Hall, 1985.
[6] Formal Systems (Europe) Ltd, FDR2 User Manual, 2005.
[7] Lin S. and Costello D. J., "Error Control Coding: Fundamentals and Applications", second edition, Prentice Hall: Englewood Cliffs, 2004.
[8] Subhasish M. and Edward J. M.," Word-Voter: A New Voter Design for triple Modular Redundant systems", Proc. IEEE Symp. VLSI Test, IEEE Press, pp. 465-470, Aug. 2000.
[9] Kang G. Shin and Hagbae Kim, "A Time Redundancy Approach to TMR Failures Using Fault-State Likelihoods", IEEE Trans. on Computers, vol. 43, pp. 1151-1162, Oct. 1994.
[10] Lisboa C. A. L., Schuler E. and Carro L., "Going beyond TMR for Protection against Multiple Faults", IEEE Symp. Integrated Circuits and Systems Design, IEEE Press, pp. 80-85, Sept. 2005.

Efficient and Scalable Steady-state Dependability Verification

Diana El Rabih and Nihal Pekergin
LACL, University of Paris-Est Créteil,
61 avenue Général de Gaulle 94010, Créteil, France
Email : delrabih@u-pec.fr, nihal.pekergin@u-pec.fr

***Abstract*—We have proposed to perform statistical model checking by combining perfect sampling and statistical hypothesis testing based on single sampling plan method in order to verify steady-state formulas. This approach allows us to consider very large monotone models and to verify rare event properties efficiently. In this paper, we extend our proposed approach by implementing different statistical methods in our verification engine and by comparing their efficiency when we verify steady-state dependability properties for large non monotone models. We show that SPRT statistical method is generally more efficient than the other statistical methods. Moreover, we show that our statistical verification approach is efficient and scalable when we consider large non monotone models.**

***Index Terms*—Statistical model checking, Perfect simulation, Dependability verification, Continuous Stochastic Logic (CSL)**

I. Introduction

Probabilistic model checking is an extension of the formal verification methods for systems exhibiting stochastic behavior. The system model is usually specified as a state transition system, with probabilities attached to transitions, for example Markov chains. A wide range of quantitative performance, reliability, and dependability measures can be specified using temporal logics such as Continuous Stochastic Logic (CSL) defined over Continuous Time Markov Chains (CTMC) [2] and Probabilistic Computational Tree Logic (PCTL) defined over Discrete Time Markov Chains (DTMC) [2]. There are two distinct approaches to perform probabilistic model checking: the numerical model checking based on the computation of transient-state or steady-state distributions of the underlying Markov chain and the statistical model checking based on statistical methods and on sampling by means of discrete event simulation or by measurement. Statistical model checking techniques constitute an interesting alternative to numerical model checking techniques for large scale systems. In the last years, different statistical model checkers have been proposed [6][15][20] especially for properties specified by time-bounded until formulas. In the statistical model checker MRMC [8] statistical model checking of CSL steady-state property has been also considered.

We have proposed in [13][14] to perform statistical probabilistic model checking by combining perfect simulation and statistical hypothesis testing based on the single sampling plan method in order to check steady-state properties of large Markovian models. Perfect simulation is an extension of Monte Carlo Markov Chains (MCMC) methods allowing to obtain exact steady-state samples of the underlying Markov chain thus it avoids the burn-in time problem to detect the steady-state. Propp and Wilson have designed the algorithm of coupling from the past to perform perfect simulation [9]. A web page dedicated to this approach is maintained by them (http://research.microsoft.com/en-us/um/people/dbwilson/exact/). As a perfect sampler, we use ψ^2 proposed in [18], designed for the steady-state evaluation of various monotone queueing networks [19]. This tool [18] permits to simulate the stationary distribution or directly a cost function or a reward of large Markov chains. The significant advantage of perfect sampling is that it provides an *unbiased* sampling of the steady-state distribution, hence the accuracy of the verification result only belongs to the statistical testing. In other words, we ensure the correctness of our results considering a specified precision level. We have compared in [10][11][12], the numerical model checker PRISM [7], the statistical module of MRMC [8] and our statistical verification engine when they are applied to the verification of steady-state properties for very large models. We have shown the efficiency and the scalability of our approach to consider very large monotone models and to verify rare event properties efficiently.

In this paper, we extend our proposed approach by implementing in our verification engine other statistical methods existing in the litterature and by comparing their efficiency when we verify steady-state dependability properties. In fact, we consider two non-monotones queueing networks, such as network of queues with negative clients, and with coxian phase-type servers to show the efficiency and the scalability of our proposed approach also in the case of non monotone models. This paper is organized as follows: Section 2 briefly presents the temporal logic CSL, the perfect sampling and our proposed approach for statistical verification based on perfect sampling. We give a brief introduction of the implemented statistical methods in Section 3. Section 4 is devoted to the case studies. First we present the models. Next, we compare and analyze the results of our experiments. Finally, in Section 5 we summarize the conclusions and provide the future works.

II. Statistical Model Checking by Perfect Sampling

A. Continuous stochastic logic (CSL)

CSL is a branching-time temporal logic with state and path formulas and it is a powerful mean to state properties over

CTMCs. Thus it is useful to specify and to verify performance and dependability measures as logical formulas over CTMCs [1]. The steady-state operator (formula) $\psi = \mathcal{S}_{\bowtie\theta}(\varphi)$ lets us to analyze the long-run behaviour of the system. The steady state formula $\mathcal{S}_{\bowtie\theta}(\varphi)$ asserts that the steady-state probability for the set of the states satisfying φ meets the bound $\bowtie \theta$, where θ is a probability threshold, $\bowtie$ a comparison operator, for example $\bowtie \in \{<, >, \leq, \geq\}$, φ is a state formula (a boolean expression of state properties).

B. Perfect sampling and statistical verification

Propp and Wilson [9] have introduced the perfect/exact sampling method, which is based on the backward coupling, also called the coupling from the past: by coming from a distant time $-\tau$ sufficiently far in the past, if all trajectories (trajectories that come from all possible initial states in $\mathcal{X}$ at time $-\tau$) are coupled in one state at time 0, then the sampled state is exactly distributed according to the stationary distribution. The backward coupling provides steady-state sample in a controlled finite number of steps, that could not be obtained by a forward coupling scheme unless the model have a strong stationary time, which is rare in our examples [17]. Let $\{X_n\}_{n\in\mathbb{N}}$ be an irreducible and aperiodic discrete time Markov chain with a finite space $\mathcal{X}$ and a transition matrix $P = p_{i,j}$. Let π denote the steady-state distribution of the chain: $\pi = \pi P$. The evolution of the system can be given by a stochastic recurrence:

$$X_{n+1} = \eta(X_n, e_{n+1}) \qquad (1)$$

with $\{e_n\}$ an independent and identically distributed sequence of events ($e_n \in \epsilon$). The transition function $\eta : \mathcal{X} \times \epsilon \to \mathcal{X}$ verifies the property that $Pr(\eta(i, e) = j) = p_{i,j}$ for every pair of states (i, j) and each random event e. An execution of the Markov chain is defined by an initial state x_0 and an sequence of events. The sequence of states given by Eq. 1 is called a trajectory. Trajectories are generated with the same sequence of events and if at time $t = 0$, two trajectories are in the same state, we say that they couple. The backward coupling is especially efficient when the underlying system is monotone. When the system is not monotone it is shown in [3] that the backward coupling can also be efficient. Given a partial order $\preceq$ on $\mathcal{X}$, an event e is said to be **monotone** if it preserves the partial ordering $\preceq$ on $\mathcal{X}$:

$$\forall (x, y) \in \mathcal{X} \quad x \preceq y \;\Rightarrow\; \eta(x, e) \preceq \eta(y, e) \qquad (2)$$

If all events are monotone, the global system is said to be monotone. According to an order $\preceq$ on $\mathcal{X}$, there exists a set $\mathcal{M}_{\preceq} \subset \mathcal{X}$ of extremal states (maximal and minimal states). When a Markov chain is monotone, all trajectories issued from $\mathcal{X}$ are always bounded by trajectories issued from $\mathcal{M}_{\preceq}$. Thus, it is sufficient to compute trajectories issued from $\mathcal{M}_{\preceq}$ since when they couple, global coupling also occurs. As the size of $\mathcal{M}_{\preceq}$ is usually drastically smaller than the size of $\mathcal{X}$, monotone perfect sampling significantly improves the sampling time [9]. Efficiency of simulations is also improved by functional perfect sampling [19]. The algorithm samples a reward value, according to a user defined reward function $r : \mathcal{X} \to \mathcal{R}$; The algorithm stops when all trajectories are in a set of states at time 0 that belongs to the same reward value (going further in the past will inevitably couple in a state that belong to this reward value). To combine monotone and functional perfect sampling, the reward function r must be monotone, that is $x \preceq y \Rightarrow r(x) \preceq r(y)$. As $|\mathcal{R}|$ is smaller than $|\mathcal{X}|$, this technique may lead to an important reduction of the coupling time. In a property verification context, since we focus on reward functions that correspond to properties we want to check, $\mathcal{R} = \{0, 1\}$. In our statistical verification method we propose to apply functional perfect sampling, so at time 0, we test if the rewards are coupled at reward 0 or 1. In other words, we test if it is a positive or negative sample. Thus we associate the reward $r_\varphi(x)$ to each state $x \in \mathcal{X}$ for a given property φ: $r_\varphi(x) = 1$ if x satisfies φ, otherwise $r_\varphi(x) = 0$. Note that, as the reward function is monotone, values 0 and 1 cover contiguous zones of the state-space. Then, an interesting phenomenon happens when the property to be checked has a small set of positive states $\{x \in \mathcal{X} | r_\varphi(x) = 1\}$ (φ corresponds to a rare property / event): coupling frequently occurs in reward value 0 and the coupling time is very short. Moreover, if $|\{x \in \mathcal{X} | r_\varphi(x) = 1\}|$ does not depend on $\mathcal{X}$ (case of saturation properties for example), then the performance of perfect sampling algorithm will be as good for very large state-spaces as for small ones. This intuition is validated by results of Section 4.

The decision method tests if φ is satisfied (positive sample) or not (negative sample) on each generated sample path by counting the number of positive samples. Then it provides decision either *Yes* if the number of positive samples is greater or equal to m (ψ is satisfied) or *No* otherwise (ψ is not satisfied). The input parameters of the algorithm are: the model defined by a labelled CTMC, M, the property φ (to be verified on each sample), the threshold parameter θ, the indifference region parameter δ, and α, β for the strength of statistical hypothesis testing. In our work, we consider ergodic Markov chains $\mathcal{M}$, hence there is a unique steady-state distribution independent of the initial state. The satisfaction property is assigned to the model but not to an inital state. (we check whether the underlying model M satisfies the steady-state formula or not). $\mathcal{M} \models \mathcal{S}_{\bowtie\theta}(\varphi)$, if the property specified by the steady-state operator $\mathcal{S}$ is satisfied by the model $\mathcal{M}$. Note that the verification of $\mathcal{S}_{\geq\theta}(\varphi)$ is the same as $\mathcal{S}_{<1-\theta}(\neg\varphi)$ and also is the same as $\neg\mathcal{S}_{<\theta}(\varphi)$.

III. Statistical methods

The statistical decision method we have used in [11][12] when performing our statistical hypothesis testing is inspired from the Single Sampling Plan (SSP) method. In this section we present the different statistical methods we implement in our statistical verification engine.

A. Current methods

a) Statistical hypothesis testing

Suppose that we have generated n samples (simulations), and a sample X_i is a positive sample $(X_i = 1)$ if it satisfies φ and negative $(X_i = 0)$ otherwise. X_i is a random variable with

Bernoulli distribution with parameter p. Thus the probability to obtain a positive sample is p. In practice, two thresholds, p_0 and p_1 are defined in terms of the probability threshold θ, and the half-width δ of the indifference region: p_0 = θ + δ and p_1 = θ - δ. Then instead of testing $H : p \geq \theta$ against $K : p < \theta$, we test $H_0 : p \geq p_0$ against $H_1 : p \leq p_1$. In fact, the strength of the statistical test was determined by two error bounds, α and β, where α is a bound on the probability of accepting H_1 when H_0 holds (known as a type I error, or false negative) and β is a bound on the probability of accepting H_0 when H_1 holds (a type II error, or false positive). There are several methods for statistical hypothesis testing decision with constraints on error bounds (α, β) [22][21][16]:

a.1) Single Sampling Plan (SSP): It is based on the acceptance sampling with fixed sample size (n): if $\sum_{i=1}^{n} X_i \geq m$, then H_0 is accepted otherwise H_1 is accepted, where m is the acceptance threshold. The hypothesis H_1 will be accepted with probability $F(m, n, p)$ and the null hypothesis H_0 will be accepted with the probability $1 - F(m, n, p)$, where $F(m, n, p)$ is a binomial distribution: $F(m, n, p) = \sum_{i=1}^{m} C(n, i)p^i(1-p)^{n-i}$ with $C(n, i)$ is the combination of i from n. It is required that the probability of accepting H_1 when H_0 holds is at most α, and the probability of accepting H_0 when H_1 holds is at most β. These constraints can be illustrated as below:

- Pr[H_1 is accepted | H_0 is true] $\leq \alpha$, which implies $F(m, n, p_0) \leq \alpha$ $\quad (C1)$
- Pr[H_0 is accepted | H_1 is true] $\leq \beta$, which implies $1 - F(m, n, p_1) \leq \beta$ $\quad (C2)$

The sample size n and the acceptance threshold m must be chosen under these constraints and their formulas for optimal performance are given in [22].

a.2) Sequential Single Sampling Plan (SSSP): If we use a single sampling plan (n, m) and the sum of the first i observations, d_i=$\sum_{j=1}^{i} X_j$, $i < n$, is already greater than m, then we can accept H_0 without making further observations. Conversely, if $d_i + n - i \leq m$, regardless of the outcome of the remaining $n - i$ observations we already know that the sum of n observations will not exceed m, then we can safely accept H_1 after making only i observations. In the modified test procedure, after each observation, we decide whether sufficient information is available to accept either of the two hypotheses or additional observations are required.

a.3) Sequential Probability Ratio Test (SPRT): This method is based on the sequential probability ratio test [21][22]: after making the i^{th} simulation (generating the i^{th} sample), one computes the following quotient:

$$q_i = \prod_{j=1}^{i} \frac{Pr[X_j = x_j \mid p = p_1]}{Pr[X_j = x_j \mid p = p_0]} = \frac{p_1^{d_i}(1-p_1)^{i-d_i}}{p_0^{d_i}(1-p_0)^{i-d_i}}$$

where d_i denoting the number of positive samples. H_0 is accepted if $q_i \leq B$, and H_1 is accepted if $q_i \geq A$. Finding A and B with a given strength α, β is non trivial, in practice A is chosen as (1-β)/α and B as β/(1-α). Then a new test whose strength is (α^*, β^*) is obtained, but such that $\alpha^* + \beta^* \leq \alpha + \beta$, meaning that either $\alpha^* \leq \alpha$ or $\beta^* \leq \beta$. In practice, it is often found that both inequalities hold. When implementing the sequential probability ratio test, it is computationally more practical to work with the logarithm of q_i. Then we accept H_0 if $f_m \leq log\frac{\beta}{1-\alpha}$, we accept H_1 if $f_m \geq log\frac{1-\beta}{\alpha}$.

Note that, the sample size for a sequential test is a random variable, meaning that the required number of observations can vary from one use of such a test to another. Furthermore, the expected sample size typically depends on the unknown parameter p, so we cannot report a single value as was the case for acceptance sampling with fixed-size samples. The expected sample size varies with the distance of p from the indifference region (p_1, p_0). It tends to be largest when p is close to the center of the indifference region, and decreases the further away p is from the indifference region.

b) Statistical estimation

An alternative statistical solution method, based on estimation instead of hypothesis testing [6]. This approach uses n observations $x_1, ..., x_n$ to compute an estimate of p: p'=$\frac{\sum_{i=1}^{n} X_i}{n}$. The estimate is such that $Pr[|p' - p| < \delta] \geq 1 - \alpha$ $(E4)$. Using a result derived by Hoeffding [[21], Theorem 1], it can be shown that n = $\lceil \frac{1}{2\delta^2} log \frac{2}{\alpha} \rceil$ $(E5)$

is sufficient to satisfy $(E4)$. If we accept ψ as true when $p' \geq \theta$ and reject ψ as false otherwise, then it follows from $(E4)$ that the answer is correct with probability at least 1 - α if either s $\models \psi$ or s $\not\models \psi$ holds. Consequently, the verification procedure satisfies $(C1)$ and $(C2)$ with β = α. As with the solution method based on hypothesis testing, a definite answer is always generated (there is no undecided results).

c) Confidence interval

Another alternative statistical solution method based on confidence intervals has been proposed in [8]. To check whether $s \models P_{>\theta}(\varphi)$, an estimate $\tilde{p}$ of the probability p starting in s is determined using standard discrete event simulation techniques. Let ξ be the user-specified confidence of the result and $\delta^{'}$ the maximum width of the confidence interval. The probability of obtaining a correct answer to the model checking problem $s \models P_{>\theta}(\varphi)$ is now guaranteed to be at least ξ provided $\delta^{'} \leq |\theta - \tilde{p}|$. In this solution method, a slight adaptation of standard sequential confidence intervals is exploited in which the sample size and simulation depth can be adapted on demand. Although $\delta^{'} > |\theta - \tilde{p}|$, this solution method provides more accurate answers as its algorithm first simulates until the confidence interval is tighter than $\delta^{'}$ and then continues simulation until it reaches the definite answer to the model checking problem. This strategy increases the accuracy because the width of the resulting confidence interval can be much smaller than $\delta^{'}$. The penalty for this increased accuracy is an increase in the simulation times thus larger model-checking times.

B. Performance comparison of statistical methods

The estimation-based approach had been compared with the approach based on hypothesis testing in [21], by considering m = $\lfloor n\theta + 1 \rfloor$ and d=np'=$\sum_{i=1}^{n} x_i$. It had been demonstrated that $p' \geq \theta \iff d > m$. This means that the estimation-based approach can be interpreted as a single sampling plan (n, m). Therefore the approach proposed in [22], when using a single sampling plan, will always be at least as efficient as the estimation-based approach. In fact, it will be more efficient because: (i) the sample size is derived using the true underlying distribution, (ii) m is not restricted to be $\lfloor n\theta+1 \rfloor$, and (iii) β = α can be accommodated. The last property, in particular, is important when dealing with conjunctive and nested probabilistic statements. The advantage of hypothesis testing is demonstrated numerically in [21]. Note, also, that the SPRT method often can be used to improve efficiency for the approach based on hypothesis testing. In fact, if a single sampling plan is used with strength (α, β) and indifference region of half-width δ, then the sample size n is roughly proportional to log α and log β and inversely proportional to δ^2 [22]. Using the SPRT method instead of a single sampling plan can reduce the expected sample size by orders of magnitude in most cases, although the SPRT method is not guaranteed always to be more efficient.

On the other hand, the method proposed by Sen et al. in [16] is not more efficient than the methods proposed by Younes et al. in [22]. In fact, Sen et al. manually selected the sample sizes for their single sampling plans. The selected sample sizes are not sufficient to achieve the same strength as used to produce the results for the SPRT method reported by Younes et al. in [22]. Finally, the confidence intervals statistical technique requires to use confidence interval of the width $< \delta$, whereas under the same conditions in hypothesis testing we would have to use the indifference region of the width less than only $2.\delta$. This can cause confidence intervals algorithms to require more samples than needed for the ones based on the hypothesis testing.

C. Statistical model checking complexity

The time complexity of any statistical solution method for probabilistic model checking can be understood in terms of two main factors: the number of observations (sample size) required to reach a decision, as well as the time required to generate each observation that depends of perfect simulation effort (coupling time). If an observation involves the verification of a path formula over a sample trajectory then the time complexity depends also of the length of trajectory prefixes (in terms of state transitions) required to determine if a path formula holds. The sample size depends on the method used for verifying probabilistic statements, as well as the desired strength (α, β) and of θ and δ. In fact, the sample size for a single sampling plan SSP is approximately proportional to the logarithm of α and β, and inversely proportional to δ^2 . If we use a sequential test SPRT, then the expected sample size also depends on the unknown probability measure p of the set of trajectories that satisfy φ. Moreover, the perfect simulation effort (coupling time) can be both model and implementation dependent, then it can be state space dependent, but models often have structure (monotone structure) [4] that can be exploited by the simulator to avoid such dependence. The length of trajectories depends on model characteristics and the property that is being verified but may be independent of the size of the state space. The space complexity of statistical probabilistic model checking is generally modest. It is needed to store the current state of a sample trajectory when generating an observation for the verification of a probabilistic statement, and this typically requires O(log $|\mathcal{X}|$) space where $|\mathcal{X}|$ is the size of state space. For systems that do not satisfy the Markov property, it may also be needed to store additional information to capture the execution history during simulation.

IV. Experimental Study

We now evaluate two non monotone models, taken from Ψ^2 and PRISM benchmarks, on which we will base our efficiency and scalability comparison. In fact, we verify the steady-state formula for these two case studies using the numerical verification approach implemented in PRISM tool and our statistical verification approach implemented in Ψ^2 tool using different statistical solution methods (Section 3), by varying the problem size (state space size related to the maximal queue capacity). We illustrate the statistical verification time ($\approx$ N_{samp}*coupling time) in seconds, where N_{samp} is the sample size, for these case studies as a function of the maximal queue capacity (state space size) and we determine the memory limit for each case when using the verification tools. Since the considered Markovian models are ergodic (by construction), thus the steady-state probabilities are independent of the initial state. Thus, the considered steady-state formula is satisfied or not whatever the initial states.

a) Negative clients queueing network

We consider the following queueing model with both positive and negative clients (Figure 1). The non-monotonicity of this model (negative clients) is shown and its perfect sampling by envelope functions is given in [3]. We have implemented this non monotone model as a Ψ^2 model as explained in [3] and we have validated the correctness of our implementation. In fact, queueing models with negative clients, have found applications in computer communications and manufacturing settings. When a negative client arrives at the queue, it has the effect of a signal, which kills ordinary (positive) clients in the node. An example of a queueing system with both positive and negative clients (jobs) is computer networks with virus infection, which deletes jobs or failures, which causes other failures and removes jobs. Let N_{max} to be the maximal

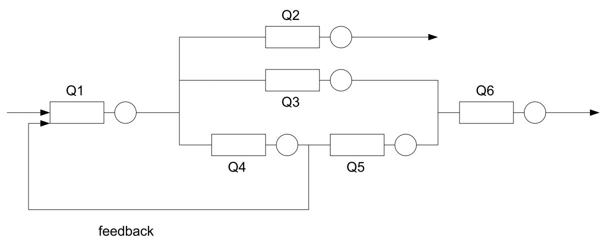


Fig. 1. Negative clients queueing network

capacity of each queue then the state space is $O((N_{max}+1)^6)$. Jobs arrive from exterior at the first queue with rates λ_1^+ (positive clients) and λ_1^- (negative clients), and exit the system from the second queue with rate μ_1 and from the sixth queue with rate μ_2. Jobs arrive also to the first queue from feedback link with rate λ_{feed}^+ and λ_{feed}^-. Jobs arrive to the i^{th} queue where $2 \leq i \leq 6$ with rates λ_i^+ (positive clients) and λ_i^- (negative clients). Also negative clients can arrive from exterior to the i^{th} queue where $2 \leq i \leq 6$ with rates κ_i^-. Let x_i denote the number of jobs currently in queue i. We define the atomic proposition that one queue of the system is full with the formula *negsysfull* = $(x_1 = N_{max}) \vee (x_2 = N_{max}) \vee (x_3 = N_{max}) \vee (x_4 = N_{max}) \vee (x_5 = N_{max}) \vee (x_6 = N_{max})$. Based on this atomic proposition, we check the following *Steady-state formula:* $S_{\leq\theta}$ (*negsysfull*) to check whether the probability that the system is full in steady-state is less than θ or not.

b) Tandem Queueing Network with coxian phase (TQN) The TQN model (Figure 2) is taken from PRISM benchmark that consists of an $M/Cox_2/1$ queue sequentially composed with an M/M/1 queue . In [11], we have implemented this non monotone model as a Ψ^2 model by using non monotone techniques (envelope function) such as defined in [3] and we have validated the correctness of our implementation. The non-monotonicity of this model is shown in [5][11]. We consider 4 TQN connected in series then our considered system consists of 4 $M/Cox_2/1$ queues. Let N_{max} be the maximal capacity of each queue then the state space is $O((N_{max}+1)^2)$ for each TQN. In each TQN, jobs arrive at the first queue with rate λ, and exit the system from the second queue with rate κ. If the first queue is not empty and the second queue is not full, then jobs are routed from the first to the second queue. In each TQN, the routing time is governed by a two-phase Coxian distribution with parameters μ_1, μ_2, and a. Here, μ_i is the exit rate for the i^{th} phase of the distribution, and 1 - a is the probability of skipping the second phase. Let x_j denote the number of jobs currently in queue j, and $x_{ph_k} \in \{1, 2\}$, for $1 \leq k \leq 4$, denote the current phase of the Coxian distribution. We define the atomic proposition that one TQN component of the overall system is full with the formula *sys-full* = $[(x_1 = N_{max}) \wedge (x_2 = N_{max}) \wedge (x_{ph_1} = 2)] \vee [(x_3 = N_{max}) \wedge (x_4 = N_{max}) \wedge (x_{ph_2} = 2)] \vee [(x_5 = N_{max}) \wedge (x_6 = N_{max}) \wedge (x_{ph_3} = 2)] \vee [(x_7 = N_{max}) \wedge (x_8 = N_{max}) \wedge (x_{ph_4} = 2)]$. Based on this atomic proposition,

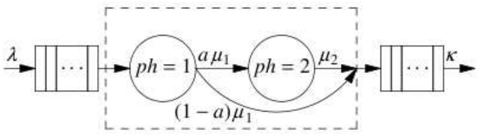


Fig. 2. Tandem queueing network with Coxian phase

we check the following *Steady-state formula:* $S_{\leq\theta}$ (*sys-full*) to check whether the probability that the system is full in steady-state is less than θ or not.

A. Experimental results

a) Negative clients network verification results: We consider λ_1^+=0.8, λ_1^-=0.2, λ_{feed}^+=0.7, λ_{feed}^-=0.3, all service rates will be state-independent with rate $\mu_1 = \mu_2 = 1$; λ_i^+=0.6, λ_i^-=0.4 and κ_i^-=0.1 for $2 \leq i \leq 6$. We give in Table I for $\theta = 0.001$ and $\epsilon = 10^{-4}$, the verification time for the considered steady-state formula $S_{<\theta}$ (*negsysfull*) by using PRISM Hybrid engine and Jacobi iterative method. Also we give in the same table for $\theta = 0.001$, $\delta = 10^{-4}/2$ respectively, and $\alpha = \beta = 10^{-2}$ the verification time for the same steady-state formula $S_{<\theta}$ (*negsysfull*) by using statistical verification methods implemented in Ψ^2 (Section 3). In fact, for $N_{max} = 21$ we obtain an out of memory message with PRISM. In all of the tables we denote by:

$PRISM$: numerical verification time in seconds for the steady-state formula by using PRISM hybrid engine.

$outm$: an out of memory message in PRISM tool.

The statistical verification time in seconds is given by combining with statistical techniques given in Section 3: $\Psi^2(SSP)$ for SSP, $\Psi^2(SPRT)$ for SPRT, $\Psi^2(SEst)$ for statistical estimation, $\Psi^2(CI)$ for confidence interval.

N_{max}	$\lvert\mathcal{X}\rvert$	$PRISM$	$\Psi^2(SSP)$	$\Psi^2(SPRT)$	$\Psi^2(SEst)$	$\Psi^2(CI)$
2	$7.29 * 10^2$	0.04	4.1	1.3	5.5	6.8
3	$4.09 * 10^3$	0.05	5.4	2.1	8.6	9.1
5	$4.66 * 10^4$	0.10	9.4	4.4	15.6	21.5
7	$2.62 * 10^5$	1.32	14.4	9.6	19.7	25.8
9	$1.00 * 10^6$	98.67	24.2	15.3	29.3	34.9
12	$4.82 * 10^6$	276.6	33.2	20.1	39.5	43.4
14	$1.13 * 10^7$	9213	42.6	29.7	54.7	67.1
21	$1.13 * 10^8$	$outm$	65.9	42.2	73.1	81.7
99	$1.00 * 10^{12}$	$outm$	98.1	53.1	126.1	155.4
999	$1.00 * 10^{18}$	$outm$	365.3	173.3	422.4	485.2
9999	$1.00 * 10^{24}$	$outm$	1315	633	1713	1929

TABLE I
NEGATIVE CLIENTS NETWORK: VERIFICATION TIME AS A FUNCTION OF STATE SPACE SIZE $\lvert\mathcal{X}\rvert$ FOR $S_{<0.001}$ (*negsysfull*)

b) Tandem network with coxian phase (4 TQN) verification results: For numerical application, for each TQN in the overall system (4 TQN in series) we consider $\lambda = 4 \times N_{max}$, $\mu_1 = 2$, $\mu_2 = 2$, $a = 0.1$ and κ =4. We give in Table II for $\theta = 0.001$ and for $\epsilon = 10^{-4}$, the verification time for the considered steady-state formula $S_{<\theta}$ (*sys-full*) by using PRISM Hybrid engine and Jacobi iterative method. Also we give in the same table for $\theta = 0.001$, $\delta = 10^{-4}/2$ respectively, $\alpha = \beta = 10^{-2}$, the verification time for the same steady-state formula $S_{<\theta}$ (*sys-full*) by using statistical verification methods implemented in Ψ^2 (Section 3). In fact, for $N_{max} = 10$ we obtain an out of memory message with PRISM.

N_{max}	$\lvert\mathcal{X}\rvert$	$PRISM$	$\Psi^2(SSP)$	$\Psi^2(SPRT)$	$\Psi^2(SEst)$	$\Psi^2(CI)$
2	$6.5 * 10^3$	0.4	7.1	4.22	8.5	9.8
3	$6.5 * 10^4$	0.5	9.4	5.12	11.4	17.1
4	$3.9 * 10^5$	1.93	17.9	8.14	20.3	22.8
5	$1.6 * 10^6$	33.2	21.8	12.3	23.6	26.4
6	$5.7 * 10^6$	150.6	34.3	21.3	39.6	44.2
7	$1.7 * 10^7$	290.6	53.2	34.1	60.9	71.5
8	$4.3 * 10^7$	476.6	78.6	57.3	98.7	117.1
9	$1.0 * 10^8$	8615	265.9	153.3	329.1	371.3
10	$2.1 * 10^8$	$outm$	386.6	233.1	422.6	492.6
99	$1.0 * 10^{16}$	$outm$	498.1	263.3	547.1	605.4
999	$1.0 * 10^{24}$	$outm$	565.3	302.1	626.3	715.2
9999	$1.0 * 10^{32}$	$outm$	1415	565.3	1826	2153

TABLE II
TQN: VERIFICATION TIME AS FUNCTION OF STATE SPACE SIZE $\lvert\mathcal{X}\rvert$ FOR $S_{<0.001}$ (*sys-full*)

B. Discussions

In Tables I and II, we have illustrated the statistical verification time ($\approx N_{samp}$*coupling time) in seconds for two non monotone models as a function of the maximal queue capacity (state space size), where N_{samp} is the sample size. In fact, the sample size, N_{samp} is the only factor that varies between the different statistical solution methods, regardless of implementation details. The sample size depends on the method used for verifying probabilistic statements, as well as the desired strength (α, β) and θ and δ. Note that, the coupling time of the perfect simulation varies with the state space size, with the implementation and with the verified property.

In Tables I and II, we show that the Single Sampling Plan (SSP) method is at least as efficient as the statistical estimation method and it will be more efficient since the sample size of the SSP method is derived using the true underlying distribution [21] (Section 3). We show also in these tables that the Single Sampling Plan (SSP) method is more efficient than the confidence intervals method, since the last method requires a smaller width of the confidence interval (Section 3). This can cause confidence intervals method to require more samples than needed for hypothesis testing based method (SSP method).

Moreover, we show in these tables that the Sequential Probability Ratio Test (SPRT) method is more efficient than the Single Sampling Plan (SSP) method. In fact, the sample size for a single sampling plan SSP is approximately proportional to the logarithm of α and β, and inversely proportional to δ^2 [21]. In our work, for SSP method we have determined the sample size using the approximation formulas given in [21]. For sequential test SPRT the expected sample size also depends on the unknown probability measure p of the set of trajectories that satisfy the property φ. In fact, for SPRT method the sample size is computed during the verification process. We show in tables I and II that using SPRT method instead of SSP method can reduce the verification time (depending on sample size) by an order of magnitude in most cases. Thus we show that SPRT statistical method is generally more efficient than the other statistical methods when performing steady state dependability verification for very large models and we show that the hypothesis testing based methods are generally more efficient than the estimation and the confidence intervals based methods. We also see in these tables that our statistical verification approach is efficient and scalable when we consider large non monotone models and it allows us to verify rare event properties efficiently on these models.

Finally, in Tables I and II we have determined the memory limit for each case when using the verification tools. There is no memory limit when using our statistical verification approach implemented in Ψ^2 tool, since the space complexity of statistical model checking is generally modest. In fact, in statistical model checking it is needed to store the current state of a sample trajectory when generating an observation for the verification of a probabilistic statement, and this typically requires O(log $|S|$) space where $|S|$ is the size of state space.

V. Conclusion and future works

In this paper, we extend our proposed approach [12][13][14] by implementing different statistical methods in our verification engine and by comparing their efficiency when we verify steady-state dependability properties for large non monotone models. We show that SPRT statistical method is generally more efficient than the other statistical methods when performing steady state dependability verification for very large models. Moreover, we show that our statistical verification approach is efficient and scalable when we consider large non monotone models and lets us to verify rare event properties efficiently on these models. Also we have found that our statistical verification approach scales better with the state space size and it is faster than PRISM tool especially for large models. In the future, we plan to complete our verification results for the CSL unbounded until formulas [14].

References

[1] A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Model-checking continous-time markov chains. *ACM Trans. Comput. Log.*, 1(1):162–170, 2000.

[2] C. Baier, B. Haverkort, H. Hermanns, and J.P. Katoen. Model-checking algorithms for continuous-time markov chains. *IEEE Trans. Software Eng.*, 29(6):524–541, 2003.

[3] A. Busic, J. M. Vincent, and B. Gaujal. Perfect simulation and non-monotone (markovian) systems. In *VALUETOOLS08*. ACM, 2008.

[4] P. Glasserman and D. Yao. *Monotone Structure in Discrete-Event Systems*. 1994.

[5] G. Gorgo. Envelope perfect sampling of 2-phases coxian service in queueing networks. INRIA. 2010.

[6] T. Hérault, R. Lassaigne, and S. Peyronnet. Apmc 3.0: Approximate verification of discrete and continuous time markov chains. In *QEST06*, pages 129–130. IEEE Computer Society, 2006.

[7] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems. In *TACAS06*, volume 3922 of *LNCS*, pages 441–444, 2006.

[8] J. P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The ins and outs of the probabilistic model checker mrmc. In *QEST09*, pages 167–176. IEEE Computer Society, 2009.

[9] D. Propp and J. Wilson. Exact sampling with coupled markov chains and applications to statistical mechanics. *Random Structures and Algorithms*, 9(1 and 2):223–252, 1996.

[10] D. El Rabih, G. Gorgo, N. Pekergin, and J.M. Vincent. Steady state dependability verification for very large systems. Iacl. 2010.

[11] D. El Rabih, G. Gorgo, N. Pekergin, and J.M. Vincent. Steady state property verification: a comparison study. In *VECOS10*. eWic, British Computer Society, 2010.

[12] D. El Rabih, G. Gorgo, N. Pekergin, and J.M. Vincent. Steady state property verification for very large systems. In *International Journal of Critical Computer-Based Systems, IJCCBS11 (to appear)*, 2011.

[13] D. El Rabih and N. Pekergin. Statistical model checking for steady state dependability verification. In *DEPEND09*, pages 166–169. IEEE Computer Society, 2009.

[14] D. El Rabih and N. Pekergin. Statistical model checking using perfect simulation. In *ATVA09*, volume 5799 of *LNCS*, pages 120–134, 2009.

[15] K. Sen, M. Viswanathan, and G. Agha. Vesta: A statistical model-checker and analyzer for probabilistic systems. In *QEST05*, pages 251–252. IEEE Computer Society, 2005.

[16] Koushik Sen, Mahesh Viswanathan, and Gul Agha. On statistical model checking of stochastic systems. 3576:266–280, 2005.

[17] J. M. Vincent. Perfect simulation of monotone systems for rare event probability estimation. In *Winter Simulation Conference*, pages 528–537. ACM, 2005.

[18] J. M. Vincent and J. Vienne. ψ^2 a software tool for the perfect simulation of finite queueing networks. In *QEST07*, pages 113–114. IEEE Computer Society, 2007.

[19] J.M. Vincent and C. Marchand. On the exact simulation of functionals of stationary markov chains. *Linear Algebra and its Applications*, 386:285–310, 2004.

[20] H. L. S. Younes. Ymer: A statistical model checker. In *CAV05*, volume 3576 of *LNCS*, pages 429–433, 2005.

[21] H. L. S. Younes. Error control for probabilistic model checking. In *VMCAI06*, volume 3855 of *LNCS*, pages 142–156, 2006.

[22] H. L. S. Younes and R. G. Simmons. Statistical probabilistic model checking with a focus on time-bounded properties. *Inf. Comput.*, 204(9):1368–1409, 2006.

A Dependable Microcontroller-based Embedded System

Amir Rajabzadeh
Department of Compute Engineering
Razi University
Kermanshah, Iran
rajabzadeh@razi.ac.ir

Mahdi Vosoughifar
Department of Computer Engineering
Islamic Azad University Arak Branch
Arak, Iran
mehdi_vosoughifar@yahoo.com

***Abstract*—This paper presents a method to make a dependable microcontroller-based system for detecting any violation from the program flow caused by transient faults. The method is based on a duplication and comparison technique and employs a "synchronous interrupt" in both microcontrollers to monitor and compare the program counters (PCs) of the microcontrollers. This is done by adding an interrupt service routine in both microcontrollers and without any modification of the application programs. The method has been experimentally evaluated using AVR ATMega-32 microcontrollers. The results show that error detection coverage of the method is 100% based on the fault models. The error detection latency varies about 1184 cycles (74 μsec) to 128147 cycles (8 msec) and the execution time overhead of the method varies between 0.5% and 50% for different PC exchange interrupt frequencies. The hardware and software overheads are about 100% and less than 0.5% respectively.**

Keywords- dependable system; control flow checking method; concurrent error detection; microcontroller-based system; embedded system.

I. INTRODUCTION

We are used to hearing about extended computer applications and explosive growth in the computation ability of processors. Based on usage patterns, processor cores can be divided into four categories [1]:

1) Computational micros: they are 32-bit or 64-bit general-purpose processors, and typically deployed as the central processing unit of mainframes, workstations, and personal computers. Most commercial off-the-shelf RISC and CISC processors fall into this category. This group has accounted for less than 2% of the volume of processors shipped.

2) Embedded general-purpose micros: they are general-purpose processors, usually 32-bit processors, designed for embedded systems. These are often scaled-down versions of existing computational micros. Embedded general-purpose micros constituted about 8% of total volumes of processors shipped.

3) Digital signal processors: they are specific-purpose processors with the ability to execute arithmetic operations efficiently. This group accounted for about 10% of the volume of processors shipped.

4) Microcontrollers: they have 8-bit, 16 bit or 32-bit processor core with memory, I/O, and peripherals on a chip. Microcontrollers have been estimated to be about 80% of the processors shipped.

Embedded systems are widely used in industrial control systems [2]. Industrial control systems usually have fairly low computational requirements and low memory capacity. This is within the domain of 8-bit and 16-bit microcontrollers. Small 8-bit CPUs still dominate the market, representing about 70% of overall processor shipments [1].

These embedded systems are usually involved with some aspects of dependability issues and system failures can severely damage human life or equipments. In these systems, dependability is an important concern and error detection mechanism has a key role in designing the system. On the other hand, as the number of transistors per chip continues to grow, the error rate per chip is expected to increase [3], the fault occurrence rates are increasing by approximately 8% per chip [4]. These trends show that to ensure correct operation of embedded systems, they must employ dependability methods against transient faults.

This paper actually presents a concurrent error detection method for embedded systems based on microcontrollers. The proposed method employs the synchronous external burst interrupt in duplication microcontrollers and compares the run time program counters of the microcontrollers in a service routine. The method has been experimentally evaluated on an AVR microcontroller-based system. The results show that error detection coverage of the method is 100% based on the fault models. The error detection latency varies about 1184 cycles (74 μsec) to 128147 cycles (8 msec) and execution time overhead of the method varies between 0.5% and 50% for different PC exchange interrupt frequencies. The hardware and software overheads are about 100% and less than 0.5% respectively.

The next section depicts the related work. Section 3 discusses the error models in this experiment. Section 4 describes the proposed method. Section 5 gives method evaluation and argues over a system under test. The results are presented in section 6, and finally, section 7 summarizes and concludes the paper discussion.

II. Related Work

This section describes how embedded systems based on microprocessor or microcontroller have been equipped to detect transient faults.

To design a dependable embedded system at least two options are available:

- Using Application-Specific-IC (ASIC) processors: such as ERC32 processor [5], LEON-FT processor [6] and THOR processor [7] with internal error detection mechanisms.
- Using Commercial Off-The-Shelf (COTS) processors: such as Intel Pentium family, PowerPC and ARM processors, or AVR and PIC microcontrollers.

Designing an embedded system with fault tolerant ASIC processor is a useful way of making a dependable system. Fault tolerant ASIC processors have many facilities for tolerating faults and have a high percentage of error detection coverage.

Concurrent error detection or fault masking mechanisms in ASIC processors are often applied at VLSI, transistor, gate or RTL levels. Chip-level and behavioral-based mechanisms may be used as well. ERC32 is a 32-bit processor [5] and it is compatible with SPARC V7 ISA. The processor has been designed for embedded space flight applications. The hardening techniques in the VLSI level (layout hardening) have been applied to reach the radiation tolerance. All registers in integer and floating unit have been provided with parity bits (gate level). Program flow control has been implemented using embedded signature monitoring (behavioral-based mechanism) and master/checker mechanism at chip-level is supported by the processor. LEON-FT is a 32-bit processor [6] and it is compatible with SPARC V8 ISA. Internal cache memory and register file in the processor has been provided with error-detection in form of parity bits. Flip-flops are implemented using triple modular redundancy (TMR) and master/checker mechanism at chip-level is supported by the processor as well.

Although fault tolerant ASIC processors present a good way of designing a dependable system, nevertheless, the use of commercial off-the-shelf (COTS) processors are phenomenally popular, because it decreases the cost significantly.

COTS processors have a low or moderate percentage of error detection coverage, but short time-to-market [8], availability in the market [8], trust in products [9], low development, test equipment and maintainability cost [10] of the systems are important matters to design a low-cost dependable system. Meanwhile, engineers can make use of a wide range of facilities in available market [8], [9].

Since COTS microcontrollers have not been designed for fault tolerant applications [9], [11], they require additional methods to enhance error detection capability in these systems [9]. The use of COTS processor incurs additional error detection mechanisms that must be employed.

Concurrent error detection methods are extremely popular among dependability methods, against transient faults. Concurrent error detection mechanisms in COTS-based systems have been classified as follow:

- Structural-based mechanisms
- Behavioral-based mechanisms

Structural-based mechanisms are based upon hardware replication. For COTS-based systems, hardware replications can be applied at chip-level, such as master/checker [12] mechanism, and system-level.

Behavioral-based mechanisms extract an abstraction from the application program, memory access etc., usually performed during “compile time”, and checking the abstraction during runtime. It has been indicated that more than 70% of all transient faults lead to deviation from the program’s normal instruction execution flow, i.e., Control Flow Errors (CFE) [13]. Control Flow Checking (CFC) techniques (i.e., techniques to detect CFEs) have been known as an effective concurrent error detection method [14]. Most of the CFC techniques are using signature monitoring technique. In this technique, at setup time, the program is decomposed into basic blocks of instructions and a signature is derived from each basic block and saved somewhere, during runtime the signatures based on the basic blocks will be regenerated and compared with the saved one. CFC techniques can be implemented by pure software such as CFCSS [15] and feature specific CFC [16], pure hardware such as watchdog direct processing (W-D-P) [17] and CFCET[9], or hybrid (combined hardware-software) such as TTA[18] and CIC[19].

The workload program in CFCSS [15] is divided into basic blocks. The blocks in the program are assigned different arbitrary signatures, which are embedded into the program during compile time. A run-time signature is generated using XOR function and compared with the embedded signatures when instructions are executed.

Feature specific CFC [16] is a pure software control flow checking technique. In this technique, the program is decomposed into basic blocks of instructions and partition blocks between them. A signature is derived from each block (i.e., basic block and partition block) at the compile time, which is the number of instructions in the block. At runtime, the technique uses performance monitoring in modern COTS processors and employs their internal counters to regenerate the signatures (i.e., instructions executed in each block) and compares them with saved ones.

Usually, the big problem of software-based CFC is the weakness of detecting an error in program crash or CPU crash states. The above drawback of the software-based CFC techniques can be eliminated in hardware based approaches.

The watchdog direct processing (W-D-P) [17] and the CFCET [9] techniques are pure hardware and they do not need any program modification.

The W-D-P verifies the application program using a separate checking program executed by a watchdog processor (watchdog program). In this technique, each application program is represented by a reference control flow graph (i.e., sequencing nodes and destination nodes) and the watchdog program shadows the application program and contains one instruction for each node in the application program.

The CFCET uses the internal execution tracing feature in modern COTS processors, which provides the ability to monitor the addresses of the taken branches in a program at run-time, and an external watchdog processor to detect any violation from branch address saved at compile-time.

As these techniques control some processor pins signals to extract signatures, they cannot be applied to microcontrollers-based system.

TTA [18] and CIC [19] are hybrid CFC techniques. The TTA technique decomposes the workload program into branch-free blocks (BFBs) and partition blocks (PBs). The scheme uses an external watchdog processor and combines five error detection mechanisms. The TTA uses three timers into the watchdog processors; BFB-timer, PB-timer and WL-timer to check each BFB, PB and whole workload execution time respectively. The address mechanism in TTA sends the size of a BFB in bytes when the BFB is entered. At the same time, the watchdog processor reads the start address of the BFB from the address bus and calculates the exit address of the BFB. At the end of the BFB, the watchdog processor is signaled. An error has occurred if the calculated exit address is different from the observed exit address. The phase mechanism in TTA checks the entering and exiting of each BFB and PB.

The CIC uses two external special pins, called event-ticking pins PM0 and PM1, which can signal out when an instruction is committed into the processor pipeline. The number of instructions executed in each BFB and PB, and also whole workload program are counted externally by the watchdog processor using the processor event-ticking pins.

This paper actually presents a concurrent error detection method based on HWSW-CFC technique. The proposed method employs the synchronous external burst interrupt in duplication microcontrollers and compares the run time program counters of the microcontrollers in a service routine. The main advantages of the proposed method are:

- Instead of using high costs ASIC components, the method uses low cost COTS processors to perform on-line system-level error detection
- It can be applied to the microcontroller-based system, and can also be applied to the processors with pipeline and on-chip caches.
- It can detect control flow errors caused by data errors.
- No modification of the workload programs is required, but it needs to add an interrupt service routine.
- Program size overhead is very low (only an interrupt service routine must be added)

III. Error Models

The basic model of errors used in this work is a violation of program’s normal instruction execution flow which will be explained in this section. These violations can be caused by transient or permanent faults in the memory or address circuits [21]. Based on these faults, five types of error models are defined as follows:

Error model 1: Program Counter Error (PCE): a PCE occurs when a fault changes program counter bits and an illegal jump occurs.

Error model 2: Branch Condition Error (BCE): a BCE occurs when a data fault (data register, flag register or data memory) causes the condition of a branch instruction is changed and a taken branch changes to non-taken branch or vice versa.

Error model 3: Branch Insertion Error (BIE): a BIE occurs when one of the non-branch instructions in the program is changed to a branch instruction as the result of a fault and the branch instruction actually causes a taken branch.

Error model 4: Branch Target Modification Error (BTME): a BTME occurs when the target address of one branch instruction is modified as the result of a fault and this instruction actually causes a taken branch.

Error model 5: Branch Deletion Error (BDE): a BDE occurs when a fault causes a branch instruction of a program changes to a non-branch instruction.

IV. Proposed Method

The proposed method uses a duplication and comparison technique at chip level for checking the correctness of the program's instruction execution flow. The program flow checking is done using motivation of synchronous external interrupts in both microcontrollers and comparison the run time program counters of the microcontrollers in the service routine regularly.

The hardware part of the method is shown in Fig. 1. It contains two microcontrollers that run an identical program with an identical external clock.

Power on Reset: It resets both microcontrollers when turn power supply is on.

Pulse Generator for Synchronization: this unit generates a pulse to motivate an interrupt for synchronization of the microcontrollers to start program execution.

Pulse Generator for PC exchange: this unit generates periodic pulses to motivate interrupts periodically for exchanging and comparing PCs between the microcontrollers to check the existence of any discrepancy.

The software part of the method is shown in Fig. 2. It contains two programs that run in both microcontrollers: 1) Synchronization Program and 2) PC Exchange Routine.

The Synchronization Program synchronizes two microcontrollers' program to start. It contains a sleep instruction and an interrupt routine that sets PC to the address of the original program. This microcontroller has an internal power on reset that delays (for several milliseconds) starting the program. This delay makes the microcontrollers execution asynchronous, because the two microcontrollers do not have exactly the same delay.

The PC Exchange Routine is regularly invoked. This routine sends its own PC register to another microcontroller, and then gets another microcontroller's PC and compares two PC contents (i.e., its own PC and got PC) to check the existence of any discrepancy.

The assembly or C codes of workload programs can be used to add the extra instructions needed to implement the method. The pseudo code of the Synchronous Program and PC Exchange Routine are shown in Fig. 3.

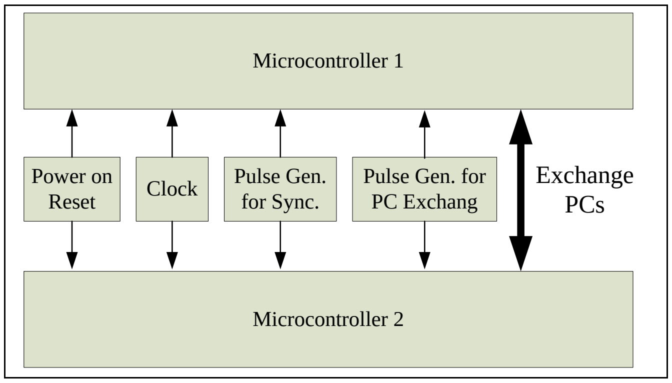


Figure 1. Hardware part of proposed method.

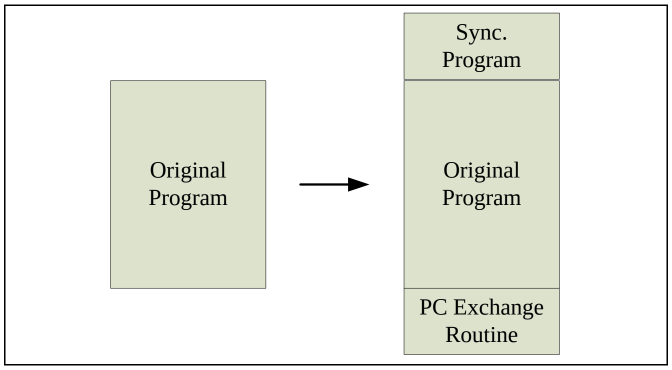


Figure 2. Software part of proposed method.

V. METHOD EVALUATION

The architecture of the experimental system is shown in Fig. 4.

```
ORG 0
Initial Ports & Interrupts
start:   sleep  /* wait until sync. interrupt is occurred*/
--------------------------------------------------------------------------------
Int_Sync_Routine(){
    Read PC from Stack
    PC = Begin       /*set PC to address of  original program*/
    Write PC to Stack
    reti      /*after return from interrupt, original program is  beginning*/
   }
--------------------------------------------------------------------------------
Begin:
/* main body of the Original Program */
--------------------------------------------------------------------------------
PC_Exchange_Routine(){
    OwnPC = Read PC from stack
    Send OwnPC  to another micro
    OtherPC = Get PC from another micro
    If ( OwnPC =! OtherPC){
          ErrorReport()
          }
    reti
    }
```

Figure 3. Pseudo codes of extra software codes

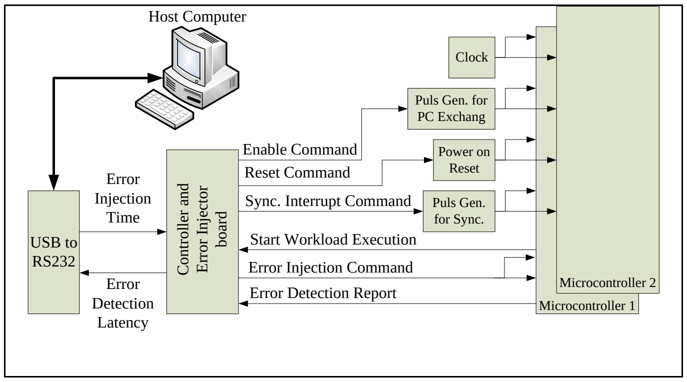


Figure 4. Software part of proposed method.

The system consists of three parts: an AVR microcontrollers board, a controller and fault injector board, and a host computer.

AVR microcontrollers board: the board has been equipped with two AVR microcontrollers that run an identical program, a 16 MHz clock generator for microcontrollers' clock pins, a monostable circuit to generate a pulse to invoke synchronization interrupt, and a clock generator with 100Hz, 1KHz and 10KHz frequencies to invoke PC exchange interrupts.

Two types of programs were executed on the AVRs board; the workload programs and a fault injector routine.

The workload programs: Three programs written in assembly language have been used in the experiment: 1) a 10 × 10 matrix multiplication ($M = A \times A^{-1}$), 2) a linked list (List) containing 100 records, and 3) a quick sort (QSort) containing 100 elements. 184 copies of the Matrix program, 124 copies of the List program, and 93 copies of the QSort program were consequently stored in the memory. These copies fill microcontrollers' flash memory (i.e., 32KB) with program codes. They were executed one after another in a loop until a fault occurs. The workload program were started when the system were reset.

The fault injector routine: The fault injection method used in these experiments is based on the software implemented fault injection (SWIFI). This paper focuses on the transient effects called SEUs (single event upsets). Several reports have mentioned that the SEU is important not only for the circuits operating in the space, but also for the digital equipments operating at the ground level [20]. It is reported in [21] that the majority (>60%) of control flow errors differ from the correct ones in only a single bit (i.e., SEU) of an address. SEUs are responsible for the modification of memory cells content (registers, internal memory, etc.). Usually, memories are protected against SEUs by means of error detecting/correcting codes (Hamming code, CRC code, Reed-Solomon code, etc.) [20]. In such cases, internal registers are of much important. Several reports have mentioned that SEU in the PC register are a major source of CFEs in comparison to other internal registers [21]. Therefore, to generate CFEs, the bits of the program counter (PC) are changed, one bit for each fault. This is done as follows: 1) the fault injector logic activates the INT0 pin of the microcontrollers, 2) the interrupt service routine reads the return address from the stack, changing a bit of the return address and then writing it back to the stack, 3) after returning from the interrupt service routine, the execution continues at an unexpected address due to the change of the value of the return address. To make sure about the coverage results, we assume that the probability distribution of the error occurring in PC bits (14 bits for 16K×16bits flash memory in AVR ATmega-32) will be uniform. The manager program on host computer issues the error injection command randomly in time during the execution of the workload program.

Controller and Error Injector board: the board has been equipped with a microcontroller and interface logic.

The interface logic establishes communication between the host computer and the controller board.

The controller board has five main tasks: 1) waiting to get a start command from the host and sending a Reset Command to reset the AVR Microcontroller Board, 2) waiting for the Start Workload Execution from the AVR Microcontroller Board and sending the Synchronization Command, 3) sending the Enable Command to activate pulse generator for PC exchange interrupts, 4) getting an Error Injection Time from the host and waiting until the time elapses and sends a command to activate INT0 pin of the two microcontrollers on AVR microcontroller board when a fault is to be injected, and 5) initialization of a timer to record the coverage and latency information.

Host Computer: The host computer contains a manager program and an offline data analyzer. The task of the host computer is to manage and control the whole experiment.

The offline data analyzer program analyses the raw data collected from the experiments and extracts the results.

VI. EXPERIMENTAL RESULTS

This section presents the experimental results of the program size overhead, execution time overhead, error detection coverage, and error detection latency. Three programs written in assembly language, i.e., quick sort (QSort), matrix multiplication (Matrix) and linked list (List), have been used in this experiment.

Error Detection Coverage: Table 1 shows error detection coverage for each workload. The basic model of errors used in this evaluation is Program Counter Errors (PCE). Although, five types of errors have been modeled in Section III, all of them change the PC finally. The changed PC causes a violation of the program normal instruction execution flow. These violations can be caused by transient faults in the memory or address circuits. The error detection coverage is 100% based on fault model for all workloads. Although, it is obvious that the method can detect all PC errors, this method has been implemented for feasibility checking and to obtain other parameters.

Program Size Overhead: The assembly (or C) codes of workload programs can be used to add the extra instructions needed to implement the method. The structure of a program after inserting the extra instructions is shown in Fig. 3. Three programs (i.e., Matrix, List, and QSort) have been used as workloads and the extra codes needed to implement the method were added to the workloads. The extra instructions inserted in the workload programs incur program size. As shown in Table 1. program size overhead is about 0.47%. This parameter achieved similar results for different workloads because several copies of each workload were consequently stored in the flash memory. These copies fill microcontrollers' flash memory (i.e., 32KB) and extra codes for each workload is constant (i.e., 152 bytes), therefore, the program size overhead is approximately constant (i.e., 0.47%).

Execution Time Overhead: The method uses synchronous external interrupts in both microcontrollers and compares their run time programs in a service routine. Interrupt handling incurs execution time. A workload is run in two cases, with presence and no presence of PC exchange interrupts, and a timer is set for measuring the relevant execution times. The execution time overhead based on different PC exchange interrupt is shown in Table 2. As Table 2 shows, the percentages of execution time overhead in the method vary between 0.5% and 50%.

Error Detection Latency: error detection latency is the average time between fault injections to error detections. A timer is set to work after each fault injection. After each fault detection, the timer is read and saved. The error detection latencies are shown in Table 2 .The mean latencies varied between 1184 and 128147 cycles for different interrupt frequencies. The latency values were calculated with respect to the processor external clock frequency which was 16 MHz.

Power Consumption Overhead: Two microcontrollers were connected together to be able to work in a duplicate configuration. The microcontrollers have all inputs connected together, but only one of them drives the outputs. It is reasonable to assume that a duplicate configuration can make duplicate of power. In this method, the total consumption of power is risen about 100%.

TABLE I. DETECTION COVERAGE AND PROGRAM OVERHEAD

	Workloads		
	QSort	***Matrix***	***List***
Errror Detection Coverage(%)	100%	100%	100%
Original Program Size (bytes)	32600 bytes	32482 bytes	32360 bytes
Extra Codes (bytes)	152 bytes	152 bytes	152 bytes
Program Size Overhead(%)	0.47%	0.47%	0.47%

TABLE II. TIME OVERHEAD AND DETECTION LATENCY

	Frequencies of interrupt		
	100Hz	***1KHz***	***10KHz***
Execution Time Overhead (%)	≈0.5%	≈5%	≈50%
Error Detection Latency (CLK)	128147 CLK	12162 CLK	1184 CLK
Error Detection Latency (msec)	8009 μsec	760 μsec	74 μsec

VII. Conclusion and Future Work

A hardware-software-based control flow checking method for COTS-microcontroller-based applications has been presented and evaluated. The method is based on duplication of microcontrollers and employs synchronous burst interrupts in both microcontrollers to monitor and compare their program counters (PCs). An implementation of the method has been experimentally evaluated. The method has been experimentally evaluated using AVR ATMega-32 microcontrollers and software-based error injection method. The results show that error detection coverage of the methods are 100% based on the fault models. The hardware and software overheads are about 100% and 0.5% respectively. The distinctive advantages of the proposed method over previous hardware-software-based error detection methods are the ability to apply in microcontrollers and the ability to detect control flow errors caused by data errors. For future works, we are going to add a system recovery mechanism after error detecting.

References

[1] J. A. Fisher, P. Faraboschi, and C. Young, ”Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools”,Morgan Kaufmann Publishers, ISBN: 1-55860-766-8, 2005.

[2] Y. He and A. Avizienis, “Assessment of the applicability of COTS microprocessors in high-confidence computing systems: a case study,” Proceedings of the international conference on dependable systems and networks (DSN2000), pp. 81–86, June 2000.

[3] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors, ”PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures”, IEEE Transaction on Dependable and Secure Computing, vol. 6, no. 2, pp. 135-148, APRIL-JUNE 2009.

[4] S. Borkar, ”Designing Reliable Systems from Unreliable Components: the Challenge of Transistor Variability and Degradation,” IEEE Micro, vol. 25, issue: 6, pp. 10-16, November-December 2005.

[5] V. Stachetti , J. Gaisler, G. Goller, and C.L. Gargasson, ”32-bit processing unit for embedded space flight applications”, IEEE Tranaction on Nuclear Science, 43(3), pp. 873–878, 1996.

[6] J. Gaisler, ”A Portable and fault-tolerant microprocessor based on the SPARC 8 architecture”, Proceedings of international conference on dependable systems and networks, pp. 409–415, June 2002.

[7] S. Asserhall, T. Petersson, and P. Blomqvist, ”RAD HARD THOR microprocessor description”, Saab Ericsson Space, Document No P-TOR-NOT-0004-SE, issue 2, Jan 1999.

[8] P. Croll and P. Nixon, ”Developing safety-critical software within a CASE environment,” Proceedings of the IEE colloquium on computer aided software engineering tools for real-time control, pp. 8, April 1991.

[9] A. Rajabzadeh and S. Gh. Miremadi, ”CFCET: A hardware-based control flow checking technique in COTS processors using execution tracing,” Elsevier Journal of Microelectronic Reliability, vol. 46, issue 5-6, pp. 959-972, May-June 2006.

[10] P. Chevochot and I. Puaut, ”Experimental evaluation of the failsilent behavior of a distributed real-time run-time support built from COTS components,” Proceedings of the international conference on dependable systems and networks (DSN-2001), pp. 304–313, July 2001.

[11] H. Madeira, R. R. Some, F. Moreira, D. Costa, and D. Rennels, ”Experimental evaluation of a COTS system for space applications,” Proceedings of the international conference on dependable systems and networks (DSN-2002), pp. 325–330, June 2002.

[12] A. Rajabzadeh, S. G. Miremadi, and M. Mohandespour, “Experimental Evaluation of Master/Checker Architecture Using Power Supply- and Software-Based Fault Injection”, Proceedings of the 10th IEEE International On-Line Testing Symposium (IOLTS 2004) Madeira Island, Portugal, pp. 239-244, July 2004.

[13] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, ”Low-Cost On-Line Fault Detection Using Control Flow Assertions,” Proceeding of the 9th IEEE International Online Testing Symposium (IOLTS'03), pp. 137-143, July 2003.

[14] A. Mahmood, and E. J. McCluskey, ”Concurrent error detection using watchdog processors-a survey,” IEEE Transaction on Computers, vol. 37, issue 2, pp. 160-174, February 1998.

[15] N. Oh, P. P. Shirvani, and E. J. McCluskey, ”Control-Flow Checking by Software Signatures”, IEEE Transactions on Reliability, vol. 51, no. 1, pp. 111-122, March 2002.

[16] A. Rajabzadeh, ”Feature Specific Control Flow Checking in COTS-based Embedded Systems”, Third IARIA International Conference on Dependability (DEPEND 2010), Venice, Italy, pp. 58-63, July 2010.

[17] T. Michel, R. Leveugle, and G. Saucier, ”A new approach to control flow checking without program modification,” Processding of the 21st international symposium on fault-tolerant computing, pp. 334-341, June 1991.

[18] S. Gh. Miremadi, J. Ohlsson, M. Rimen, and J. Karlsson, “Use of Time, Location and Instruction Signatures for Control Flow Checking”, Dependable Computing and Fault Tolerant System, IEEE Computer Society Press, vol. 10., ISBN 0-8186-7803-8, 1998, pp. 201–221.

[19] A. Rajabzadeh, S. Gh. Miremadi, and M. Mohandespour, ”Error detection enhancement in COTS superscalar processors with performance monitoring features,” Journal of Electron Testing: Theory and Applications (JETTA), pp. 553-567, 2004.

[20] B. Nicolescu, R. Velazco , M. Sonza-Reorda, M. Rebaudengo , and M. Violante, ”A software fault tolerance method for safety-critical systems: effectiveness and drawbacks”, Proceedings of the 15th symposium on integrated circuits and systems design (SBCCI-02), pp. 101-106, 2002.

[21] M. Rimen, J. Ohlsson, and J. Karlsson, ”Experimental evaluation of control flow errors”, Proceedings of the Pacific Rim international symposium on fault tolerant systems (PRFTS-95), pp. 238-243, December 1995.

Dependable Ordering Policies for Distributed Consistent Systems

Matei Dobrescu, Manuela Stoian, Cosmin Leoveanu
General IT Directorate
Insurance Supervisory Commission
Bucharest, Romania
mdobrescu@csa-isc.ro

***Abstract*—A distributed system can be characterized by the fact that the global state is distributed and that a common time base does not exist. A linearly ordered structure of time is not always adequate for distributed systems and many authors have adopted a generalized non-standard model of time which consists of vectors of clocks. The paper present an improved algorithm where these clock-vectors are partially ordered and form a lattice. By using timestamps and a simple clock update mechanism the structure of causality is represented in an isomorphic way and the causal consistency is obtained. Finally, is presented the implementation of this new algorithm which allow to compute a consistent global snapshot of a distributed system for replicated services, where messages may be received out of order.**

Keywords- *temporal ordering; distributed systems; causal consistency; events structure; clock-vectors*

I. INTRODUCTION

An asynchronous distributed system consists of several processes without common memory which communicate solely via messages with unpredictable (but non-zero) transmission delays. In such a system the notions of global time and global state play an important role but are hard to realize. Since in general no process in the system has an immediate and complete view of all process states, a process can only approximate the global view of an idealized external observer having immediate access to all processes.

The fact that a priori no process has a consistent view of the global state and a common time base does not exist is the cause for most typical problems of distributed systems. Control tasks of operating systems and database systems like mutual exclusion, deadlock detection, and concurrency control are much more dificult to solve in a distributed environment than in a classical centralized environment. The great diversity of the solutions to these problems exemplifies many principles of distributed computing to cope with the absence of global state and time. To simplify the design and the validation of algorithms for asynchronous systems, one can try to simulate a synchronous distributed system on a given asynchronous systems, simulate global time (i.e., a common clock) and simulate global state (i.e., common memory), and then use these simulated properties to obtain the desired result. The first approach is realized by so-called synchronizers [1] which simulate clock pulses in such a way that a message is only generated at a clock pulse and will be received before the next pulse. The second approach does not need additional messages and the system remains asynchronous in the sense that messages have unpredictable transmission delays. This approach has been proposed by Lamport [2]. He shows how the use of virtual time implemented by logical clocks can simplify the design of a distributed mutual exclusion algorithm. The last approach was pursued by Chandy and Lamport in their snapshot algorithm [3], one of the fundamental paradigms of distributed computing. More recent approaches ([4], [5], [6], [7], [8], [9]) proved that to maintain the data consistency, the special synchronization operations are reduced to the minimum and are delivered using a global ordering algorithm. Almost all this algorithms assure a time complexity linear to network delays by utilizing timestamp estimations.

The organization of the informational flow as a linear sequence of discrete events is inappropriate for asynchronous distributed systems, where information is distributed and perception is delayed. Distributed environments require a distributed notion of time and a theory of distributed time provides a natural framework for solving problems in distributed environments.

While a synchronous distributed computing model provides processes with bounds on processing time and message transfer delay, which can be used to safely detect process crashes and allow consequently the non-crashed processes to progress with safe views of the system state, the asynchronous model is characterized by the absence of time bounds (this model is sometimes called *time-free* model). In these systems one can only assume an upper bound on the number of processes that can crash (let denote them by m) and consequently design protocols relying on the assumption that at least $(n - m)$ processes are alive, n being the total number of processes. In a distributed environment, the main drawback is the consensus problem, that has no deterministic solution when even a single process can crash. The *consensus* problem can be stated as follows: each process proposes a value, and has to decide a value, unless it crashes, such that there is a single decided value to be proposed for assuring validity. The impossibility of solving consensus has motivated researchers to find distributed computing models, weaker than the synchronous models but stronger than the asynchronous models, in which consensus can be solved. In such a model we can describe the target in terms of distributed time, as a *timeslice* of logical simultaneity in the temporal relations expressed by a *time model*. The *timed asynchronous* model considers asynchronous processes equipped with physical clocks to ensure temporal ordering.

Resuming, one can say that the principles for temporal ordering in asynchronous distributed systems are: 1) Each machine maintains its own time; 2) There is no global shared clock; 3) Each target has a list of files on which it depends; 4) At the target one compare the associated timestamps; 5) If the target is older than some file that it depends on, then target is re-built.

A simple algorithm that respect these principles should ensure the following steps: 1) A time server maintains global notion of time; 2) Each machine periodically contacts time server asking for current global time; 3) Machine updates local time with global time. For implementation, the problem to solve is to associate with each event a logical timestamp T such that if $A \Rightarrow B$ then $T(A) < T(B)$, where $\Rightarrow$ means that event A precedes event B. Then, the ordering algorithm keeps for each i-th process a non-negative integer counter T_i, initially 0; when i-th process performs computation event, $Ti \leftarrow Ti + 1$ and when i-th process sends a message m, it computes $Ti \leftarrow Ti + 1$ and appends $T(m) \leftarrow T_i$ to m. Finally, when i-th process receives message m, $T_i \leftarrow \max\{Ti, T(m)\} + 1$. For event A at i-th process, one define $T(A) = T_i$ computed during A. A scheme for such a process is shown in figure 1 a. A better solution of Mattern is based on clock vectors [10], i.e. the i-th process keeps a vector T_i with n elements (see figure 1b). Each element $T_i[j]$ is a non-negative integer counter, initially 0. The following statements work: when i-th process performs any event, $Ti[i] \leftarrow Ti[i] + 1;$ when i-th process sends m, it also appends $T(m) \leftarrow T_i$ to m; when i-th process receives m, it also computes $T_i[j] \leftarrow max\{T_i[j], T(m)[j]\}$ for each $j \neq i$; for event A at i-th process, define $T(A) = T_i$ computed during A such that $T(A) < T(B) = [\, \forall j\!: T(A)[j] \leq T(B)[j] \;\vee \exists j\!: T(A)[j] < T(B)[j]]$.

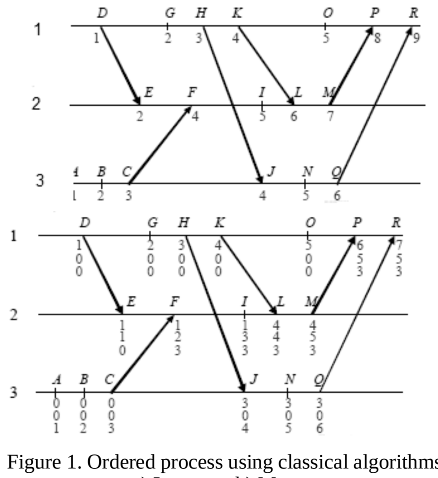


Figure 1. Ordered process using classical algorithms: a) Lamport; b) Mattern

While in some sense the snapshot algorithm computes the best possible attainable global state approximation, Lamport's virtual time algorithm is not that perfect. In fact, by mapping the partially ordered events of a distributed computation onto a linearly ordered set of integers it is losing information. Events which may happen simultaneously may get diferent timestamps as if they happen in some definite order. For some applications (in our case the objective was the ordering of events in the alerts flow of an emergency system) this defect is noticeable. In this paper, we aim at improving Lamport's virtual time concept, considering that a partially ordered system of vectors forming a lattice structure is a natural representation of time in a distributed system. In this non-standard model of time all events which are not causally related are considered simultaneous, thus representing causality in an isomorphic way without loss of information.

II. EVENT STRUCTURES

In an abstract setting, a process can be viewed as consisting of a sequence of events, where an event is an atomic transition of the local state which happens in no time. Hence, events are atomic actions which occur at processes. Usually, events are classified into three types: send events, receive events, and internal events. An internal event only causes a change of state. A send event causes a message to be sent, and a receive event causes a message to be received and the local state to be updated by the values of the message.

Events are related: Events occurring at a particular process are totally ordered by their local sequence of occurrence, and each receive event has a corresponding send event. Formally, an event structure [11] is a pair $(E;<)$, where E is a set of events, and „<" is a partial order on E called the causality relation.

Event structures represent distributed computations in an abstract way. For a given computation, $e < e'$ holds if one of the following conditions holds:

1) e and e' are events in the same process and e precedes e',

2) e is the sending event of a message and e' the corresponding receive event 3) $\exists e''$ such that $e < e''$ and $e'' < e'$.

The causality relation is the smallest relation satisfying these conditions.

A *consistency mechanism* guarantees that operations will appear to occur in some ordering that is consistent with some condition. Most of the research on this subject addressed *strong* consistency conditions like *sequential consistency* and *linearizability*. These conditions guarantee that operations appear to be executed in some sequential order that is consistent with the order seen at individual sites. Unfortunately, supporting either sequential consistency or linearizability requires a non-negligible cost. A way around this cost is to define conditions that provide weaker guarantees on the ordering of operations, and can be efficiently implemented. These conditions can be roughly classified into two categories: *weak* and *hybrid* conditions. Weak conditions provide very little guarantee on the relative ordering of events at different processes. These conditions admit very efficient implementations, but they are too weak to support conventional methods for concurrency control. Hybrid conditions distinguish between two types of operations, *strong* and *weak*. Strong operations appear to be executed atomically, in some sequential order that is consistent with the order seen at individual processes. The only guarantees provided for weak operations are those implied by their interleaving with strong operations. When the *consistency mechanism* offers *hybrid* conditions, one can define the synchronization as hybrid too.

Let's now consider that a model of a distributed consistent system (DCS) system is composed of a finite set of sequential processes P_1, P_2,… P_n, one for each node. The processes interact with the application program at the same node using *call* and *response* events. The processes P_1, P_2,… P_n interact through a finite set of $x \in X$ shared objects via *message-send* and *message-receive* events. The process P_i can be also modeled as an automaton with states and a transition function that takes as input the current state and a call or message-receive event, and produces a new state, a set of response events and a set of message-send events.

A *history* of a process describes what steps the process takes and times they occur; it must satisfy certain "consistency" conditions. An execution of a set of processes is a set of histories, one for each process.

An *execution* of a set of processes is a set of histories, one for each process, together with a one-to-one correspondence between the messages sent by P_i to P_j and the messages received by P_j from process P_i. We use the message correspondence to define the *delay* of any message in an execution to be the real time of receipt minus the real time of sending. The execution is admissible if the delay of every message is less than d, for fixed $d \geq 0$, and for every P_i, at any time at most one call at P_i is *pending*.

Every object is assumed to have a *serial specification*. The specification defines a set of *operations*, which are ordered pairs of call and response events, and a set of *operations sequences*, which are the allowable sequences of operations on that object. As an example, in the case of a read/write object, the ordered pair of events *[Read$_i$ (x), Return$_i$ (x,v)]* forms an *operation* for any process P_i, object x, and value v, i.e. (v, $(r(x,v))$) as does *[Write$_i$ (x,v), Ack$_i$ (x)] (w(x,v))*.

A. Legal Operations in Distributed Consistent Systems

An *execution history* of a DCS is a partial order $\widehat{H} = (H, \rightarrow_H)$, formally:

$$H = \bigcup_i h_i$$

$o_1 \rightarrow_H o_2$ if:

1) $\exists P_i : o_1 \rightarrow_i o_2$ (in that case $\rightarrow_H$ is called a *process-order* relation
2) $\exists w(x,v), r(x,v)$ such that $w(x,v) \in o_1$ and $r(x,v) \in o_2$ (in that case $\rightarrow_H$ is called a *read-from* relation)
3) $\exists o_3 : o_1 \rightarrow_H o_3$ and $o_3 \rightarrow_H o_2$ (transitivity)

Let's now consider a history $\hat{H}$. Informally, an operation $o \in H$ is legal if it does not read overwritten values, i.e. the legality of an operation (causal dependency) is defined as follows:

Definition 1. An operation o is legal if $\forall r(x,v) \in o : \exists o'$ such that:

$o' \rightarrow_H o$ (o' precedes o)

$w(x,v) \in o'$ (o' is the operation that wrote v into x)

$\forall o''$ such that $o' \rightarrow_H o'' \rightarrow_H o : w(x) \notin o''$ (there is no overwriting operation)

Definition 2 A history $\widehat{H} = (H, \rightarrow_H)$ is *causally consistent* if, for each process P_i there exists a linear extension of $\widehat{H}$ in which all operations issued by P_i are legal. In other words, the order of all operations of P_i maintains causal dependency of the operations .

As an example let see Figure 2, where appears the model of an execution that is only possible in a causally consistent system. This shows processes P_i, P_j and P_k modifying concurrently different objects. The operation $o_{i,1}$ updates object y at the same time that $o_{j,1}$ updates object x. The second concurrent update occurs when $o_{j,3}$ writes to object x and $o_{k,4}$ writes to object y. P_k is able to read the update of P_j in $o_{k,1}$ but the update $w(y,1)$ from P_i is not seen until $o_{k,3}$. These executions are acceptable because the two objects are written concurrently and hence P_k makes no assumptions about which object will be updates first. The model in Figure 2 shows that the execution $\widehat{H}2$ is not serializable since there does not exist a linear extension of $\widehat{H}2$ in which all operations are legal. However, $\widehat{H}2$ is causally consistent as there exists, for each process P_i, a linear extension including all write operations plus all read operations issued by P_i , in which all operations are legal.

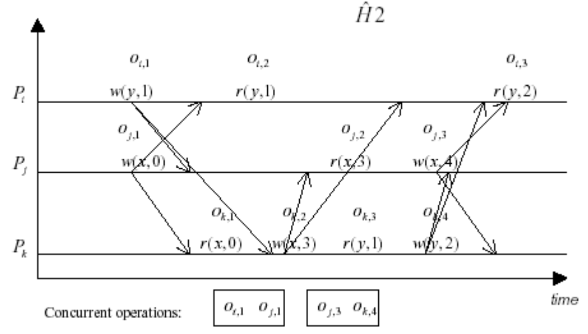


Figure 2. Causal consistency executions

The causal ordering of messages deals with the notion of maintaining the same causal relationship that holds among “message send” events with the corresponding “message receive” events. Events that occur at a single site are ordered in time in the normal way. Informally, an event *a* at a site *s* is ordered temporally after an event *b* at site *t* if, and only if, there is a sequence of messages, the first one originating from site *t* after the event *b*, the next message being sent from the destination site of the first message after the first message is received there, and so on, with the last message being received at site *s* before the event *a*. The following execution examples show how inconsistencies can appear if the system does not ensure causal synchronization.

A conflict-free run is depicted in Figure 3. This is normally the case, where due to the relatively low network roundtrip times are small compared to user interaction intervals. In this example, Process 1 modified and unselected the object (released the lock over the object) before Process 2 had sent a select message: Process 2 started without waiting for any synchronization or acknowledgement messages.

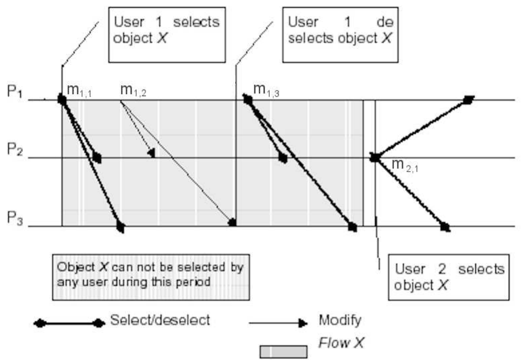


Figure 3. Normal execution with no conflict.

In Figure 4 it is presented a case when the system does not provide any causal consistency mechanism. P_2 received the deselect message from P_1 and immediately selected the same object (message $m_{2,1}$) before P_3 received the previous deselect message from P_1. This case may occur if packets travel between sites through different paths, and their roundtrip times vary noticeably. If P_2 modifies its local copy before $m_{1,3}$ arrives to P_3, the database becomes inconsistent. The last occurs because there is no causal synchronization.

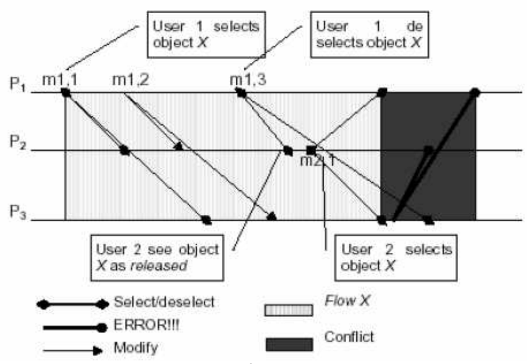


Figure 4. Execution with conflict and no causal synchronization.

The execution diagram depicted in Figure 5 shows the result of applying hybrid synchronization to the previous example. P_3 does not start a flow, it does not send any update message, until it receives the message sent by $m_{1,3}$. Therefore, P_2 cannot start any object processing until the select strong select operation is globally ordered at every site.

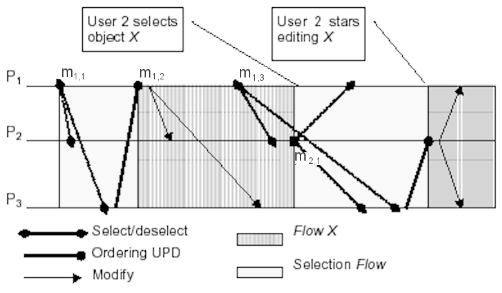


Figure 5. Causal synchronization

Causal consistency is attractive because not only it can meet the sharing needs of many applications but it can also be implemented efficiently. It is possible to complete send and receive accesses to causally consistent objects without synchronisation among processes (or sites) that store copies of the objects. This can lead to a scalable architecture because coordination among a large number of nodes is not necessary with causally consistent shared objects. Among these, service-oriented architectures (SOA) are typical for the necessity to assure a dependable global ordering.

III. AN ALGORITHM FOR DEPENDABLE GLOBAL ORDERING OPERATIONS IN SOA

This algorithm is an improvement of the classical ordering algorithms based on timestamps. As framework We considered a service-oriented architecture (SOA), which actually is a collection of services. A service is a function that is well-defined and does not depend on the context or state of other services. These services communicate with each other in the same way as interact processes in a distributed system. Services is becoming a platform for information interaction between applications.

Our approach can maintain the data consistency among multiple service replicas while we still guarantee the loose coupling and location transparency characteristics among the service replicas. In the informational flow, consistent operations are classified as either strong or weak. Informally, flows consistency guarantees two properties:

1) Strong operations appear to be executed in some sequential order.

2) If two operations are invoked by the same process and one of them is strong, then they appear to be executed in the order they were invoked.

Each replica of the editor holds a local copy of the entire memory, a local timestamp counter and an array that keeps conservative about the values of all other timestamp counters in the system. A weak operation is executed instantly on the local copy of the object. In case of writes, update messages are sent to all other processes, which update their local copies of the memory upon receiving these messages. Timestamps are used to enforce global ordering on the strong operations. Strong operations are timestamped with the local timestamp counter, and a message is sent to all processes; the initiating process then increments its local timestamp counter by 1. The execution of any strong operation is postponed until the timestamp of that operation is smaller than all the estimated timestamp counters of the system. If more than one strong operation can be executed together, they are executed according to their timestamps in increasing order.

The algorithm guarantees that if process P_i estimates P_j counter as x, then the local timestamp of P_j is at least x (that is, the estimate is conservative). This implies that all strong operations ever invoked by P_j bearing timestamp smaller than x have arrived at P_i, and ensures that all strong operations are executed in the same order and that weak operations that were invoked later are also executed later.

We assume a system of n processes, connected by an interconnection network, each maintaining a local copy of the entire database. Each process P_i has a local timestamp counter, lts_i, initially 0, and an array ts_i such that ts_i[j] contains P_i's estimate of lts_i. Weak operations are executed locally and instantly. If a weak operation is a write of v to object x, then *update* messages are broadcast to all processes (an *update* message includes the new value v to object x to be updated). A process that receives an update message of v to object x, updates its copy of object x with v. For any strong operation (*select* or *deselect* messages), a strong-op message is sent to all other process; this message not only contains update information (path of the object to be selected) but also a timestamp *lts*. Process P_i suspends the execution of a strong operation with timestamp *ts*, until it knows that the counters are at least ts+1. When several pending strong operations may be executed, they are executed according to their timestamps and *ids* in increasing order.

Executing a strong *select* operation at process P_i is done by updating the list of selected objects in the local copy. If object x specified in the select message is marked as already selected by another operation, the operation is ignored and no action is taken. Otherwise, object x is added to the local selection list. Executing a strong *deselect* operation at process P_i is done by deleting the object x specified in the message from the selection list.

Process P_i increases its timestamp in each of the following cases:

1) After P_i sends a strong-op message to all processes.
2) After P_i receives a strong-op message with timestamp equal to lts_i and for all j, $ts_i[j] \geq lts_i$.
3) A strong operation *with* $ts=lts_i$*-1* was executed in P_i, and there exists k such that $ts_i[k] \geq lts_i$.

In the last two cases, a *ts-update* message is sent to all other processes.

Let's now discuss how the proposed algorithm offers dependable solutions. A crucial issue encountered in distributed systems is the way each process perceives the state of the other processes. To that end, the proposed model provides each process p_i with three sets denoted $idle_i$, $active_i$ and $uncertain_i$. The only thing a process p_i can do with respect to these sets is to read the sets it is provided with; it cannot write them and has no access to the sets of the other processes. These sets, that can evolve dynamically, are made up of process identities. Intuitively, the fact that a given process p_j belongs to one of the three sets provides p_i with some hint on the current status of p_j . More operationally, if $p_j \in idle_i$, p_i can safely consider p_j as being crashed. If $p_j \notin idle_i$, the state of p_j is not known by p_i with certainty: more precisely, if $p_j \in active_i$, p_i is given a hint that it can currently consider p_j as not crashed; when $p_j \in uncertain_i$, p_i has no information on the current state (crashed or active) of p_j. The specification of the sets $idle_i$, $active_i$ and *uncertaini*, $1 \leq i \leq n$, is the following:

S1 - Initial global consistency. Initially, the sets $active_i$, $idle_i$ and $uncertain_i$ of all the processes p_i are identical. Namely, for $t = 0$, $\forall\ i, j$: $state_i(t) = state_j(t)$, where *state* is *active*, *idle* and *uncertain* respectively.

S2 - Internal consistency. The sets of each p_i define a partition $idle_i(t) \cup active_i(t) \cup uncertain_i(t) = \Pi$, $\forall\ i,t$. and any two sets in $idle_i(t)$, $active_i(t)$ and $uncertain_i(t)$ have an empty intersection.

S3 Consistency of the $idle_i$ sets: an $idle_i$ set is never decreasing, i.e. $idle_i(t)\ \forall\ idle_i(t + 1)$, $\forall\ i,t$

S4 Consistent global transitions. The sets $idle_i$ and $uncertain_j$ of any pair of processes p_i and p_j evolve consistently. More precisely, $\forall\ i, j, k, t_0$ we have $(p_k \in active_i\ (t_0)) \cap (p_k \subseteq idle_i\ (t_0 + 1)) \Rightarrow \forall\ t_1 > t_0 : p_k \notin uncertain_j\ (t_1)$.

As we can see from these specifications, at any time t and for any pair of processes p_i and p_j, it is possible to have $active_i(t) = active_j(t)$ (and similarly for the other sets). Operationally, this means that distinct processes can have different views of the current state of each other process. The rules [S1-S4] define a distributed computing model that satisfies the strong consistency property. That property provides the processes with a mutually consistent view on the possibility to detect the crash of following a given process. More specifically, if the crash of a process p_k is never known by p_i (because p_k continuously belongs to $uncertain_i$), then no process p_j will detect the crash of p_k (because $p_k \in idle_j$). Conversely, if the crash of p_i is known by p_j, the other processes will also know it.

IV. The implementation of the application

We will present an application that uses the proposed ordering algorithm in a distributed system for emergency management. The main objective is the consistent synchronization of alerts. That implies to have complete information about the temporal dimension of alerts, compatibility with the alert standards and with the software and hardware resources running the application. The participants in the alert process are computers acting as nodes in a network which communicate using standard ISO-OSI protocols. The application is realized in Java, in order to be supported on a large set of hardware platforms.

The application is composed from several classes, as follows:

AlertNode – is the class for instantiation of the matricial logical clock of the node that contains the function *main()* which launch the client and server execution threads. In the initial state one must specify the node ID, the number of active nodes in the whole network and the port of the server which take the alert.

AlertServerThread – is the class that implements the several able to receive alerts. For each connection a dedicated thread is created, so many clients can be simultaneously serviced.

AlertProtocol – is the class that implements the communication protocol between the server and the alert client. This class contains the function *readAlert*, that initiate the class *CAPHandler*, which parses the client alert in the Common Alerting Protocol (CAP) XML format. When the alert is received, one launch the method *receiveAction* of the matricial clock that implements the clock logic.

MatrixClock – is the class that implements the matricial logical clock.

AlertClientThread – is the class which allows to transmit alerts from client to server, only in CAP format.

As an example, let now consider the following scenario, as shown in figure 6:

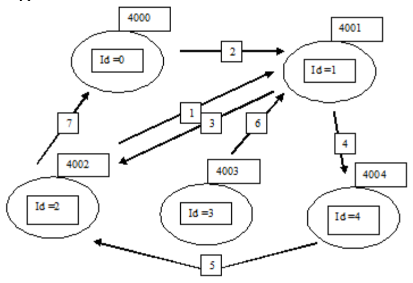


Figure 6. An alert secenario

The ellipses represent the network nodes, each node having an unique identifier. In the rectangle above the node appears the number of the listening port. The arrows represent the direction of the transmitted alert, and the associated numbers represent the sequential order for alerts transmission. On each node is running a software agent with double functionality (client and alert server). The alert is connection oriented, using TCP stream sockets. A socket is unique identified by an IP (node address) and a port (which directs the data to destination).

When a node has to transmit an alert to other node, the server try to connect the destination node through a separate execution thread), but it maintains the idle state in order to accept other connections also. The client is addressed by a command line, on the associated port. But it is noticeable that the client can interrogate periodically a data base were are registered the out of limits parameters, without a special command of the server, and can decide himself is another node must be alerted. Figure 7 shows the values of the timestamps at the matricial clocks, for the first steps of the scenario depicted in figure 6. At the end of the process the clocks have the value of the arrows end.

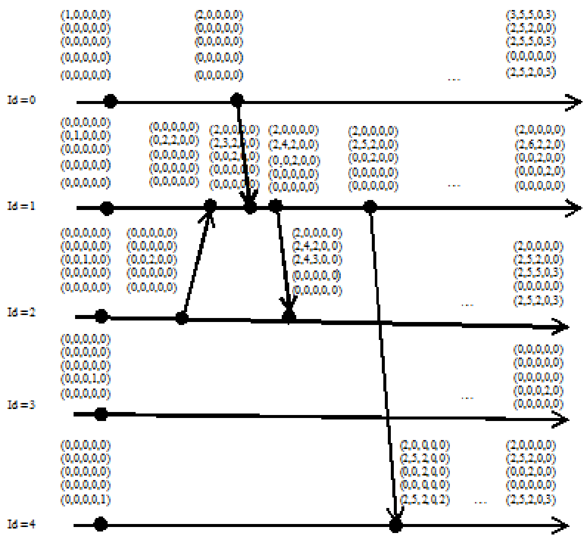


Figure 7. Alerts flow and the matricial clocks of the nodes

The main contribution of the proposed scheme is the correlation of alerts in emergency systems, introducing as a new element in the classical Lamport algorithm a matriceal clock which acts as a component of the advertising protocols structure. The efficiency of this mechanism is improved by adding a fault detection component of the timestamp assignment that verifies if each secondary vector of the matrix is smaller than the principal vector of the current node.

The algorithm imposes to send dedicated messages for the matriceal clock refresh, at the same frequency as that of the information messages, if in a specified interval a process does not succed to perform a send-receive operation.

V. Conclusions

This paper proposed an improved global ordering algorithm for dependable distributed computing, that encompasses both the synchronous model and the asynchronous model. The algorithm guarantees the order of messages delivery to the application and respect temporal and causal relationships. In this aim the strong operations are timestamped with a local timestamp counter, and a message is sent to all processes. If more than one strong operation can be executed together, they are executed according to their timestamps and in increasing order. We have chose to focus on the distinction between performing a data operation locally at a process, based on its local state, and performing an operation that requires communication between processes before the control can be returned to the application. When collaboration involves communicating via a single or multiple flows, causal relationships among messages sent over the flows must be maintained to preserve the context in which a message is sent.

Other contributions which derive from the conceptual framework can be summarized as it follows: the implementation and testing of a general protocol for data replication in a distributed architecture; a scheduler for operations of a collaborative process; the definition of a formal consistency criteria of the flows framework; the classification of strong and weak operations that allows the implementation of this consistency criteria; the definition of the form that a process state perceives each other's states by accessing the contents of three local non-intersecting sets, (*uncertain*, *active*, and *idle*). The proposed system has been implemented in JAVA and tested over a set networked LINUX workstations, equipped with QoS capabilities

Future work will be oriented on: strong operations' generalization for different type of operations, specially those operations that modify the topology of a scene tree, i.e. addition or deletion of nodes; the implementation of a policy that allows to support latecomers and early leaving in to the distributed system; the implementation of a multicast protocol for supporting many users simultaneously; the evaluation of the benefits of the admission control policies with respect to the media quality of the serviced clients, the average latency time, and the throughput of the system.

REFERENCES

[1] C. J. Fidge, „Timestamps in Message-Passing Systems that Preserve Partial Ordering". In *Proceedings of 11th Australian Computer Science Conference*, pp. 56-66, 1988

[2] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Comm. of the ACM*, 21(7), pp. 558-565, 1978.

[3] K. M. Chandy and L. Lamport, „Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Transactions on Computer Systems*, 3(1), pp.63-75,1985.

[4] H. Kopetz, A. Ademaj and A. Hanzlik, „Combination of clock-state and clock-rate correction in fault tolerant distributed systems", *Real-Time Systems*, Vol. 33, pp.139-173, 2006

[5] Yang, J., Q. Zhang and N. Gu (2006) A Consistency Maintenance Approach in Replicated Services, *Proc. of the Sixth IEEE Int. Conf. on Computer and Information Technology*, pp. 248 – 258

[6] A. Hanzlik, „SIDERA - A Simulation Model for Time-Triggered Distributed Real-Time Systems", *Int. Review on Computers and Software* (IRECOS), Vol. 1, N. 3, pp. 181-193, 2006

[7] R. Dobrescu and M. Dobrescu, A "flows consistency" model for message ordering in collaborative distributed systems, *13th IFAC Symposium on Information Control Problems in Manufacturing*, 2009

[8] V. Cholvi, A. Fernández Anta, E. Jimenez, P. Manzano, M. Raynal. "A Methodological Construction of an Efficient Sequentially Consistent Distributed Shared Memory". *The Computer Journal, 53(9),* pp.1523-1534, 2010

[9] R. Jimenez-Peris, M. Patiño-Martinez, D. Serrano, J. Milán and B. Kemme, „Leveraging the Scalability and Availability of Replicated Databases with Autonomic Capabilities", *3rd Int. Conf. on Autonomic Computing and Communication Systems*, 2009.

[10] F. Mattern, " Virtual Time and Global States of Distributed Systems", *Proceedings of the Parallel and Distributed Algorithms*, pp.215-226, 1989

[11] S. Gorender, R. Macedo and M. Raynal, "A Hybrid and Adaptive Model for Fault-Tolerant Distributed Computing", *Proceedings of the Int. Conf. on Dependable Systems and Networks,* pp.412-421, 2005

Fuzzy Event Assignment for Robust Context-aware Computing

Hannes Wolf, Jonas Palauro, Klaus Herrmann
Institute of Parallel and Distributed Systems, Universität Stuttgart, Germany
Email: {*hannes.wolf*|*klaus.herrmann*}@ipvs.uni-stuttgart.de, jonas@palauro.de

***Abstract*—User acceptance of context-aware applications relies on unobtrusive interaction and perceived dependability of the application. The accurate recognition and handling of high-level context information is a key factor , to achieve both. Currently, applications mostly work as isolated pieces of software and have to deal individually with the high uncertainties when recognizing and ambiguities when consuming high level context information. We use Adaptable Pervasive Flows (APF), to overcome these limitations and present our Fuzzy Event Assignment (FEvA) algorithm to resolve the ambiguities when assigning context information to the applications. Our simulation results show assignment accuracies between 83% to 97% and an improved performance when dealing with false positive, out-of-order and missed context information.**

I. Introduction

Context-aware applications provide users support for a broad range of activities in everyday life. But unobtrusive application, demand for little explicit user interaction and context information as the main source of input, driving the execution [1]. Throughout the paper we consider an application that automatically documents a nurses tasks in daily patient care without the necessity of explicit user interaction as an example for this kind of unobtrusive support. However, the nurse will only accept this kind application if it deals with the uncertainties of context recognition in a robust way.

Recognizing the actions and other high-level context information of the nurse from the environment involves reading data from (uncertain) sensors, processing the data and composing the context information from different sources. The sensor readings can be quantified by accuracy and precision, but the processing amplifies the degree of uncertainty. A context management system (CMS) [2], [3] provides the context information to the application via a query interface or in an event-based fashion. If the CMS also supports uncertainty handling, it further supplies the application with the degree of uncertainty for the requested information. The application decides if the received context information is consumed from the CMS so that it will no longer be available for other applications. This is necessary because the uncertain information could be interpreted in multiple ways, leading to inconsistent behavior of the informed applications. If the uncertainty is too high, the application discards the context. Similarly, the application could provide some policy, which allows the CMS to make those decision instead, but either approach leads to ambiguities when consumerist context.

We claim, that the *perceived dependability*, i.e. dependability from a user's point of view, of a context-aware application is conditioned by two factors: 1) the handling of uncertainties in context information and 2) the resolution of ambiguities when actually consuming context.

CMS provide sophisticated methods dealing with uncertainty of primary context, like location information [4] and for high-level context information – like needed to document the activities of the nurse – uncertain context reasoning or event correlation can be applied [5], [6]. However there is no system that takes the structural or contextual relation between the different applications into account to resolve the ambiguities, when assigning context information to the right application.

The algorithm we propose to robustly solve the assignment builds on Adaptable Pervasive Flows (APF) or flows for short. Flows originate from classical workflows, and were recently proposed as a programming paradigm for pervasive applications [7]. A flow basically consists of a number of activities $a \in A$ that with directed transitions $t \in T$, which define a partial execution order for the activities. An activity in a flow either represents some computational task, e.g. writing a database record or invoking a Web Service, or it specifies a task that a human has to perform in the real world, such as our nurse administering medicine to a patient.

Our newly developed *Fuzzy Event Assignment* (FEvA) algorithm supports the execution of flows, providing a robust yet flexible dynamic assignment of *context events* to single *activities* in the flow. FEvA determines all the activities, that could be interested in the available context information, based on the flows structural information and its current execution state. FEvA coordinates a competition between the activities, that is based on fuzzy logic, weighting the events and selecting the most appropriate candidates. Finally, the candidates are assigned, in such a way that a successful execution becomes more probable. We have implemented a simulation and tested our algorithm against false positive context information, context that occurs out-of-order, and missing context. The results show an avg. assignment accuracy of 91%.

The rest of this paper is structured as follows: In the next Section we introduce the assumptions we make in our system including the context model. After that, we present FEvA in Section III. Then, we show and discuss the results of our evaluation in Section IV. Finally we put our approach in perspective to related work in Section V and conclude the paper in Section VI.

II. Basic Assumptions and Models

First, we introduce some basic concepts of flow execution and then introduce our context model. Following that, we define the failure model on the context information the application has to deal with.

Applications that run in our system are flows. At development time, a programmer creates a *flow model* $\mathcal{F}$ of the application (documenting the daily routine of a nurse) that acts as a template. An *instance* of the flow is created at runtime, e.g., for a specific nurse, specific day, and executed on a flow engine. Most of the activities in this flow map to real-world activities of the nurse. Therefore we use a CMS that provides the flow engine with *context events*.

***Definition** 2.1 (Context Event):* A situation that can be recognized in the real world is referred to as event $e \in U$, where U denotes the universe of all events that the CMS can measure.

As the recognition relies on (uncertain) sensor readings, processing and composition of low-level context, an event is always uncertain. Currently, we assume that this uncertainty solely arises from the recognition process, i.e. the nurse in the real world always behaves correctly wrt the flow.

Events that represent semantically similar context can belong to a common *event type*, where each event belongs to at least one.

***Definition** 2.2 (Event Type):* An event type $E \subset U$ contains a number of individual events $E := \{e_1, \ldots, e_n\}$. A single event can be a member of different event types.

The purpose of an event type is twofold: First, it allows the programmer to simply select the most appropriate context the activity should respond to. A flow $\mathcal{F}$ defines a function $\epsilon : A \times \mathbb{N} \rightarrow \mathcal{P}(U)$ that maps activities to a number of distinct event types. $\mathcal{P}(U)$ denotes power set over the universe of events and $epsilon(a, i)$ yields the ith event type of activity a and $\emptyset$ otherwise. We write $\epsilon(a)$ for short, when referring to all event types of an activity. Second, the related semantics of the grouped events allow a more accurate recognition and classification. Events that are not contained in the expected event type are likely out of scope. The flow engine registers the event types $\epsilon(a)$ of a running activity at the CMS and receives an *event instance*, that indicates which event actually has been detected and also provides the degree of uncertainty.

***Definition** 2.3 (Event Instance):* Let E be an event type. An event instance $I_E : E \rightarrow [0, 1]$ defines a probability distribution function, where $\sum_{e \in E} I_E(e) = 1$.
We use probability theory, but the definition could be adjusted to incorporate other measures of uncertainty as well. This concludes the context model when considering single activities in the flow or single situations in the real-world. However, for successful flow execution we have to consider a number of event instances and their order, uncertainty and type. Therefore we define an *event sequence*.

***Definition** 2.4 (Event Sequence):* Let $\mathcal{E} := \{E_1, \ldots, E_j\}$ be the image set of ϵ for a given flow $\mathcal{F}$, i.e. $\mathcal{E}$ contains all event types used in $\mathcal{F}$. An Event Sequence $S := (I_E^1, \ldots, I_E^k)$ is an ordered list of k event instances, where $E \in \mathcal{E}$ can be of any type.

Given $\mathcal{F}$, we call S a valid sequence if it leads to a successful execution of $\mathcal{F}$. While event types are easily checked, we already showed in previous work how the uncertainty could be decreased using the flow structure [8]. However, as we motivated in the introduction, it remains a challenge to assign the event instances to the right activities. Therefore, ordering and actual occurrence of the events in the sequence are crucial. In the following, our we present our failure model covering three failure types.

The first failure type are false positives. The context system sometimes notifies the application of an event that did not occur in the real world. We define α as the percentage of added false positive events in a sequence S. For example, let S be a valid sequence and $\alpha = 0.05$, then the size of modified sequence S_α would be $|S_\alpha| \approx |S| * 1.05$. We assume that added event instances are uniformly distributed over the sequence, their type is randomly picked from $\mathcal{E}$ and the probability distribution of the instance is similar to the others in the sequence i.e. they can not be distinguished from the correct events in the sequence when inspecting the distribution.

Second, there are out-of-order events. Due to network transmission delay in a distributed CMS, temporary sensor failures or delay when running the situation detection operators on low-level context, the events in a valid sequence may be shifted. We define γ as the percentage of events that have not been affected by a sequence shift. For example, given $\gamma = 85$, 85% of the events have not been affected, but the remaining 15% are shifted in time (either way) according a normal distribution $\mathcal{N}(0, \omega)$. The ω is chosen such that 15% of the samples are larger than ± 1. The integer part of the sample indicates the shift and direction in the event sequence relative to its original position in the sequence.

Finally, there are events that happened in the real world, but the CMS simply missed them. This might be due to sensor unavailability, a bad reading, or the removal of a reading due to high uncertainty during the situation detection. Let δ denote the percentage of events from a valid event sequence that have been missed. So a value of $\delta = 10$ results in $|S_\delta| \approx |S| * 0.9 = |S| * (1 - \delta \backslash 100)$. The three failure models we presented can be combined and applied to a single event sequence, written as $S_{\alpha,\gamma,\delta}$.

III. Fuzzy Event Assignment

The goal of FEvA is to interpret a given a recognized event sequence $S_{\alpha,\gamma,\delta}$ for a flow $\mathcal{F}$, so that it is a valid sequence and leads to the same execution path as the original sequence S. FEvA tries to "find" the original events and map them to the right activities and, given enough evidence from existing events, also tolerate that some events are missing. We first introduce more details on *activity states*, the activity life cycle, flow execution semantics and our extensions to the activity state space. Then, we describe how FEvA fuzzifies incoming event instances and provides them as possible candidates for the competing activities using the *candidate selection* algorithm. As the execution of an activity progresses

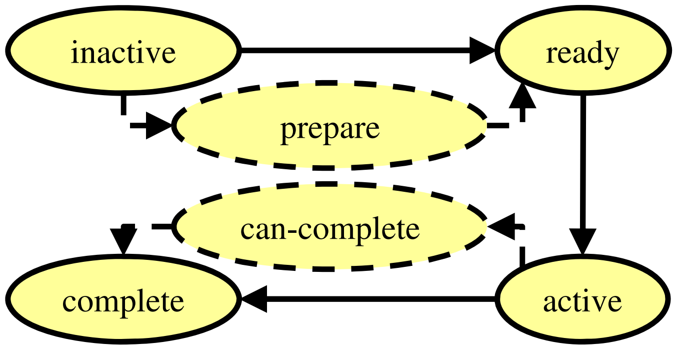


Fig. 1. Activity State Machine

it will eventually complete and the *event assignment* algorithm finalizes the event assignment and resolves possible conflicts.

A. Flow Model Semantics and Activity State Extension

As mentioned in the introduction, a flow basically consists of a directed acyclic graph $G = (A, T)$ with activities $a \in A$ as nodes and directed transitions between activities $t \in T \subset A \times A$ as edges. Each transition can be annotated with a logic condition, which depends on the received context information of the originating activity. Furthermore, some of the activities are mandatory, and must be completed successfully for a successful flow execution.

While being executed, an activity a assumes four distinct states that indicate its completion progress. These states are in order of execution $Z = \{inactive, ready, active, complete\}$. When an instance of $\mathcal{F}$ is created, all activities are in the inactive state. Meeting all prerequisites for execution, an activity a switches to the ready state. The flow engine then registers its event types at the CMS. Having received the first event, a assumes the active state. When a has received the last event, the conditions of the outgoing transitions are evaluated and a reaches the *complete* state. Following activities may be set to the ready state, depending on the condition evaluation results. The execution of the whole flow instance is considered successful if no activity is currently running i.e. in the active state and all activities that are mandatory for the successful execution are completed. The state machine for an activity is also depicted in Figure 1, considering only states and transitions with continuous lines. For more details on the formal flow model and the execution semantics refer to our previous work [8].

We extend the activity state space for FEvA with two additional states: $Z' := Z \cup \{prepare, can\text{-}complete\}$. Further, let $\omega : A \rightarrow Z'$ be the function that retrieves the current state of an activity a. If the state of the preceding activities a_x with $(a_x, a) \in T$ is $\omega(a_x) \geq ready$, then a switches from *inactive* to *prepare*. The event types of a prepared activity are also registered at the CMS. Therefore the number of registered event types increases and the chances of missing an out-of-order event are reduced. If an event is recognized out-of-order in current systems, the flow engine has not yet registered the event types at the CMS and the event is dropped.

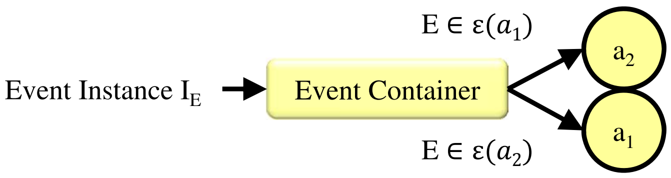


Fig. 2. Event Container Principle

Instead, the early events can now be cached by the flow engine until the prepared activity actually becomes *ready*. Before switching from *active* to *complete*, an activity first assumes the *can-complete* state. The state indicates that a has selected candidates for all its event types but the preceding activities have not yet reached the *complete* state. Waiting for their completion, we avoid that a consumes events that are possibly more suitable candidates for the predecessors while a better fitting event for a might still arrive. However, the conflict resolution mechanisms, which we will introduce later, will occasionally bypass this rule. The extensions of the state machine are also depicted in Figure 1 as states with dashed borders and dashed state transitions.

B. Fuzzy Event Assignment

As mentioned earlier FEvA, consists of two algorithms, one for event candidate selection and one for event assignment. Both algorithms are plugged into the *event container*, the component of the flow engine responsible for event caching and dispatching. Activities that have registered their event types at the CMS are known to the event container and stored separately for each event type as set of competing activities $C_E = \{a \in A | (\omega(a) \neq inactive \wedge \omega(a) \neq complete \wedge E \in \epsilon(a)\}$. The event container also stores a list of event instance candidates for each activity denoted as $candidates[a]$. Whenever the flow engine is notified about a new event instance I_E, it is stored in the event container.

The candidate selection algorithm, depicted in Algorithm 1, computes, which event instances are added to the list of candidates of an activity. First the algorithm computes a fuzzified representation of I_E. We utilize fuzzy sets (cf. [9]), each representing a linguistic value, defining the fitting quality of I_E for a single event from E. The individual fuzzy membership functions are defined as $\mu_x : [0,1] \rightarrow [0,1]$ where $x \in \{VL, L, M, H, VH\}$ is one of the linguistic variables ”very low”, ”low”, ”medium”, ”high”, ”very high”. The functions map the probability - or more generally the degree of uncertainty - of a single event given by I_E to a fuzzy membership value for the respective linguistic value. We used the same membership functions based on the standard triangular fuzzy functions for all combinations of activities and event types. For example, $e_1 \in I_E$ is the event where the nurse measures the pulse of the patient and $I_E(e_1) = 0.3$ then $\mu_M(I_E(e_1)) = 0.75$ and $\mu_H(I_E(e_1)) = 0.25$. As each event is weighted by every membership function, we further introduce the *fuzzy event weighting* function $\lambda : [0,1] \rightarrow [0,1]^5$ as concise version including all the membership function results. Given $e_1 \in I_E$, $\lambda(I_E(e_1))$ yields $(\mu_{VL}(I_E(e_1)), \mu_L(I_E(e_1)), \mu_M(I_E(e_1)), \mu_H(I_E(e_1)), \mu_{VH}(I_E(e_1)))$, i.e. the mapping of the individual probability of the event to the fuzzified membership in all five fuzzy sets. The candidate selection algorithm computes the weighted event $\lambda(I_E(e))$ for all $e \in E$ and sends it to each activity, that is subscribed to E i.e. the activities in the set C_E. Each activity checks if any, and which, conditions have an at least "high" matching with I_E. They compute this matching using their own *fuzzy activity weighting* function $\kappa : [0,1]^5 \rightarrow [true, false]$, deciding if I_E is a suitable candidate event. We consider the mentioned example of pulse measuring $e_1 \in E$ again. An activity will chose I_E as possible candidate if and only if the lowest nonzero linguistic membership in $\lambda(I_E(e_1))$ is "high" or "very high" thus, $\mu_H((I_E(e_1) \geq 0.5 \vee \mu_{VH}(I_E(e_1)) \geq 0.0)$. Given this equation is fulfilled, the result of $\kappa(\lambda(I_E(e_1)))$ yields true and I_E is stored as a possible $candidate[a, E]$ for the activity a and the event type E. If I_E becomes a new candidate for an activity it further checks if it has the best overall fitting of the candidates available in $candidates[a, E]$. We denote the best fitting event instance as I_E^{max} where $\forall I_E \in candidates[a, E] : \mu_x(I_E^{max}(e)) \geq \mu_x(I_E(e))$ for the the highest non-zero linguistic membership value of I_E^{max}. Given that the new event instance $I_E = I_E^{max}(e)$ is the new best fitting one, the algorithm issues an assignment request for I_E that is later handled by the event assignment algorithm.

Algorithm 1 Candidate Selection Algorithm

```
   Input: $C_E, I_E$
   for $e \in E$ do
     $fuzzyWeights[e] \leftarrow \lambda(I_E(e))$
   end for
5: for $a \in C_E$ do
     for $e \in E$ do
       if $\kappa(fuzzyWeights[e])$ then
         $candidates[a, E] \leftarrow candidates[a, E] \cup \{I_E\}$
       end if
10:  end for
     $I_E^{max} \leftarrow max_{I_E}(candidates[a, E])$
     $issue\_assignment\_request(I_E^{max})$
   end for
```

Having eventually received incoming events for all event types, an activity a changes its state to *can-complete*. But before it can commit its execution and reach the final *complete* state it must consume a single candidate event for each type from the event container. However, this might lead to conflicts, because a requested event instance I_E could also have been requested by another activity a_o. The event assignment algorithm is responsible for the final consumption of I_E and tries to resolve the conflicts. We omit a listing of the algorithm due to space reasons.

First the algorithm checks, if some $a_o \in C_E$ has also issued an assignment request. In case,this other activity has alternative candidates available , i.e $(candidates[a_o, E] \setminus I_E) \neq \emptyset$, we just select another candidate for a_o and assign I_E to a for consumption. When there is no other candidate available, we check if only one of the activities a or a_o is mandatory, preferring the mandatory one for the event assignment. Given that a now still lacks a non-requested candidate, we try to reevaluate the rejected candidates. If some of the other requested events for a have a fuzzy value that is very high, we relax the candidate criteria for E and check if more candidates become available. If this still does not yield a suitable candidate, the activity may be completed without having assigned an event to E. In order to do this a number of strict criteria have to be fulfilled. First, there is a fixed maximum number of allowed missing events per activity; in our case only one missing event is accepted. Second, the preceding activities of a must be *complete* and the succeeding activities must be in *can-complete*. Third the succeeding activities must have at least one event instance assigned with a very high fitting value. These rules ensure that there is enough evidence available, so we can assume that the necessary event has been missed and complete the activity nonetheless. However, most of the conflicts we mentioned will be resolved without using the conflict resolution mechanisms, when more of the correct events arrive. In the next section we assess the performance of FEvA wrt. the failure model we introduced in Section II.

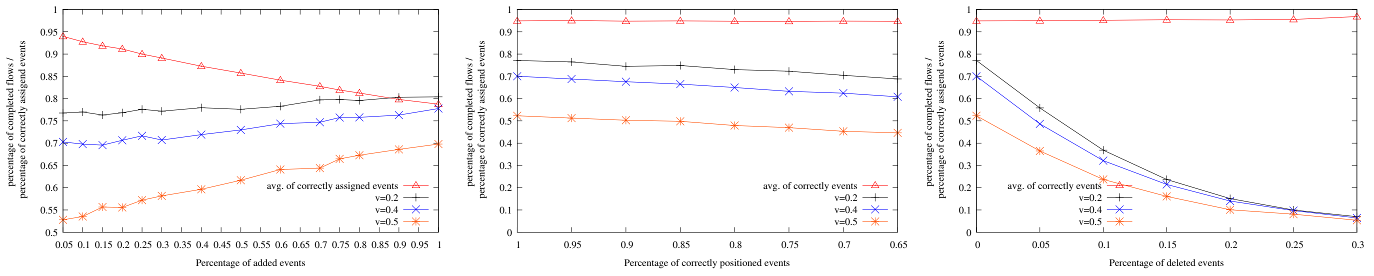
IV. Evaluation

To evaluate FEvA, we extended the existing simulation environment, developed in previous work [8]. In the following we present the setup of the individual experiments and the simulation parameters and then discuss the results.

A. Simulation Setup

In order to evaluate FEvA we need a suitable set of realistic flow models to test the algorithms in a wide range of cases. We created the flows from so-called workflow patterns. These patterns are common building blocks that have been identified in a number of real-world workflows, which include a high number of human tasks [10]. Furthermore, it has been shown that these patterns adhere to a co-occurrence distribution, indicating that some patterns follow others more regularly [11]. We used this co-occurrence distribution to generate the structure of the workflows used in our evaluations. The flows had sizes of 20, 30, 40 and 50 activities. For each size, we simulated 25 different flows and fed 100 different event sequences into each flow. Therefore every data point in the evaluation results is created from 10,000 flows executions.

The simulation is informed by two sets of parameters. The first set consists of the two values, *ground truth GT* and *variance V*, which are used to generate the probability distribution for the individual event instances. The GT is the probability that the CMS detects the correct situation that happened in the real world. The remaining probability $1-GT$ is geometrically distributed to the other events of the event type. The V adds noise to the resulting distribution, i.e. the probability of each event of the pdf is altered by V and the final distribution again normalized. We simulated GT for values from 0.4 to 1.0 in steps of 0.1 and V from 0.05 to 1.0. To show



(a) added events α (b) moved events γ (c) deleted events δ

Fig. 3. Simulation Results - FEvA performance

the results clearly, the figures depicted only contain the graphs for $GT = 0.5$ which is FEvA's threshold for accepting an event (cf. Section III-B) and variances of $0.2, 0.4, 0.5$ representing a rather low, medium high and high amount of noise.

The second set defines the parameters α, γ and δ according to our definition in Section II. For α we chose values between 5% and 100%, for γ between 100% and 65% and for δ between 0% and 30%.

B. Results

The main results of the simulation for each of the parameters α, γ, and δ are depicted in Figure 3. The other values are set to their default value, i.e. 100 for γ and 0 for α and δ.

The results for the added false positives are the most surprising (c.f Figure 3(a)). While the number of correct events assigned to the activities decreases slowly from about 94% to 78%, the actual number of successfully completed flows increases. This is due to the fact, that the added false positives also take part in the assignment process and, especially in conflict situations, may be mapped to an activity too early and falsely, too. On the one hand this effect becomes stronger the higher the variance is; On the other hand the decrease in correctly assigned events also holds for the low variance but the effect cannot be identified right away. If we recognize the events with a high accuracy, the flow is able to deal with more false positives, but also assigns some of these false positives to the activities. However, for the worse context readings the result is counter-intuitive, as more flows complete because it is more likely that a fitting false positive exists.

When we confront FEvA with out-of-order events (c.f Figure 3(b)), the algorithm performs very well and tolerates the deviation. Most of the events are correctly assigned, i.e. well over 97% and the impact of the few falsely assigned events on the flow completion is low compared to the variance.

Considering the missed events (c.f Figure 3(c)), FEvA is still able to assign the remaining events accurately, again with well over 97%, but the missing events have a very strong impact on the flow execution. While a low number of missing events is somewhat tolerable, the amount of correctly completed flows drops rapidly to a mere 7% when more than a quarter of all events is missing. The difference between the graphs also becomes smaller. This shows that the deletion of events has a more severe impact on the correct execution than the variance. The mechanism we introduced to tolerate missed events somewhat helps, but there is a lot of room for improvement.

V. Related Work

We have investigated two different areas on work that is related to FEvA. We begin discussing the handling of fuzzy or uncertain (context) information in other workflow management systems, and continue with activity recognition systems, especially with a background in the health-care domain considering our example application.

The integration of context information into classic workflows used in enterprises has first been suggested by Wieland et al. [12] Their original approach does not consider uncertainty in context information, but in the meantime the authors provided a basic solution based on policies [13], which allows a workflow to specify a well defined behavior when dealing with uncertain context information and sensor failures. But their work actually lacks an algorithm, such as FEvA is, for matchmaking between uncertain context information and the workflow activities. Also from the area of business process management, Adam et al. [14], [15] proposed to use fuzzy logic to enable soft decisions in workflows based on the input provided to the workflow. However they did not consider uncertainties or ambiguities in the input information.

There are plenty of workflow models based on petri-nets (e.g. [16]), and also fuzzy petri net variants have been proposed [17] and applied to workflows [18]. Basically all elements of a petri net – places, markers and transitions – can be fuzzified and integrated into a fuzzy reasoning process. On the one hand, if we interpret an event instance as a fuzzified marker, our approach would be somewhat similar to the fuzzy petri nets. On the other hand the events represent external input, which has not been considered yet.

There have been numerous studies on activity recognition in the health-care domain. The major factors for decreasing the uncertainty and ambiguity in the recognition results are the selection of appropriate sensors, the available application model as well as the ease of sensor deployment and cost. For example, Barger et al. [19] studied a health status monitoring application and learned behavioral patterns of the user observing his daily activities using a number of motion sensors. But their system lacks an application model, leading to missed events and false positives and a rather low recognition accuracy for uncommon situations. Najafi et al. [20] have built a monitoring system for elderly people using one acceleration sensor, and detecting position transitions and mode of locomotion. While performing very well for single transitions in a specific test scenario, the authors admit that for extended periods of time the sensing quality decreases. Finally, Biswas et al. [21] investigated different scenarios for elderly monitoring in home and professional-care scenarios. They used a complex and most likely expensive sensor setup for activity recognition, tailored for a specific scenario, state of the art recognition methods and very promising results. However their application model informing the recognition is basic and has been created manually. The authors specifically remark that knowledge from domain experts should be encoded in the recognition process. A flow is a very detailed representation of expert application knowledge, that we used to resolve the ambiguities when mapping the events to the activities.The presented approaches all use sophisticated activity recognition techniques, but do not consider the kind of application knowledge, that a flow could provide. In summary, the FEvA approach is a unique algorithm bridging the gap between activity recognition and context aware applications, dealing with ambiguities when consuming the recognized events.

VI. Conclusions and Future Work

FEvA, our new algorithm to resolve ambiguities when consuming uncertain context information, has demonstrated its effectiveness under the provided failure models. It achieves a reasonable assignment when facing a large number of false positive events and works very well when facing out-of-order events. We were able to limit the impact of a small number of missing events. We conclude that FEvA would be a very useful supplement for systems and environments where a lot of context information drives structured applications, such as the health-care documentation scenario we mentioned. In this scenario, FEvA significantly improves the perceived dependability of context-aware applications, advancing their user acceptance.

However, there is room for improving FEvA. Currently, we aim for a better mechanism to deal with the missed events. Furthermore, investigating the effects of different weighting functions per activity could lead to interesting results. Finally, it would be interesting to extend the approach not taking only one but multiple flow into account.

VII. Acknowledgments

The work described in this paper is partially funded by the 7th Framework EU-FET Project 213339 ALLOW

References

[1] Weiser, M.: The computer for the 21st century. Scientific American **265**(3) (September 1991) 94–104

[2] Kjaer, K.E.: A survey of context-aware middleware. In: SE'07: Proceedings of the 25th conference on IASTED International Multi-Conference, Anaheim, CA, USA, ACTA Press (aug 2007) 148–155

[3] Baldauf, M., Dustdar, S., Rosenberg, F.: A survey on context-aware systems. International Journal of Ad Hoc and Ubiquitous Computing **2**(4) (2007) 263–277

[4] Lange, R., Weinschrott, H., Geiger, L., Blessing, A., Dürr, F., Rothermel, K., Schütze, H.: On a generic uncertainty model for position information. In Rothermel, K., Fritsch, D., Blochinger, W., Dürr, F., eds.: First Internationa Workshop on Quality of Context, QuaCon 2009. Number 5786 in LNCS, Stuttgart, Springer (June 2009) 76–87

[5] Koch, G.G., Koldehofe, B., Rothermel, K.: Cordies: expressive event correlation in distributed systems. In: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems. DEBS '10, New York, NY, USA, ACM (2010) 26–37

[6] Choudhury, T., Philipose, M., Wyatt, D., Lester, J.: Towards activity databases: Using sensors and statistical models to summarize people's lives. IEEE Data Eng. Bull. **29**(1) (2006) 49–58

[7] Herrmann, K., Rothermel, K., Kortuem, G., Dulay, N.: Adaptable Pervasive Flows–An Emerging Technology for Pervasive Adaptation. In: Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, IEEE Computer Society (2008) 108–113

[8] Wolf, H., Herrmann, K., Rothermel, K.: Robustness in Context-Aware mobile computing. In: IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob'2010), Niagara Falls, Canada (10 2010)

[9] Zadeh, L.: Fuzzy sets. Information and Control **8**(3) (1965) 338–353

[10] Chiao, C., Iochpe, C., Thom, L.H., Reichert, M.: Verifying existence, completeness and sequences of semantic process patterns in real workflow processes. In: Proc. of the Simpsio Brasileiro de Sistemas de Informao. Rio de Janeiro: UNIRIO, Brazil (2008) p. 164–175.

[11] Lau, J.M., Iochpe, C., Thom, L.H., Reichert, M.: Discovery and analysis of activity pattern co-occurrences in business process models. In: ICEIS (3). (2009) 83–88

[12] Wieland, M., Kopp, O., Nicklas, D., Leymann, F.: Towards context-aware workflows. In Pernici, B., Gulla, J.A., eds.: CAiSE07 Proceedings of the Workshops and Doctoral Consortium. Volume 2., Trondheim Norway, Tapir Acasemic Press (Juni 2007)

[13] Wieland, M., Käppeler, U.P., Levi, P., Leymann, F., Nicklas, D.: Towards Integration of Uncertain Sensor Data into Context-aware Workflows. In in Informatics (LNI), G.E.L.N., ed.: Tagungsband INFORMATIK 2009 Im Focus das Leben, 39. Jahrestagung der Gesellschaft für Informatik e.V. (GI), Lübeck, Lecture Notes in Informatics (LNI) (September 2009)

[14] Adam, O., Thomas, O., Martin, G.: Fuzzy WorkflowsEnhancing Workflow Management with Vagueness. In: EURO/INFORMS Istanbul 2003 Joint International Meeting. (2003) 6–10

[15] Adam, O., Thomas, O., Vanderhaeghen, D.: Fuzzy-set-based modeling of business process cases. In: ICCBR Workshops. (August 2005) 251–260

[16] van der Aalst, W.M., van Hee, K., Houben, G.: Modelling and analysing workflow using a petri-net based approach. In: Proc. 2nd Workshop on Computer-Supported Cooperative Work Petri nets and related formalisms. (1994) pp 31–50

[17] Pedrycz, W., Gomide, F.: A generalized fuzzy petri net model. Fuzzy Systems, IEEE Transactions on **2**(4) (November 1994) 295 –301

[18] Raposo, A., Coelho, A., Magalhaes, L., Ricarte, I.: Using fuzzy petri nets to coordinate collaborative activities. In: IFSA World Congress and 20th NAFIPS International Conference, 2001. Joint 9th. Volume 3. (2001) 1494 –1499 vol.3

[19] Barger, T., Brown, D., Alwan, M.: Health-status monitoring through analysis of behavioral patterns. Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on **35**(1) (2005) 22 – 27

[20] Najafi, B., Aminian, K., Paraschiv-Ionescu, A., Loew, F., Bula, C., Robert, P.: Ambulatory system for human motion analysis using a kinematic sensor: monitoring of daily physical activity in the elderly. Biomedical Engineering, IEEE Transactions on **50**(6) (2003) 711 –723

[21] Biswas, J., Tolstikov, A., Jayachandran, M., Fook, V.F.S., Wai, A.A.P., Phua, C., Huang, W., Shue, L., Gopalakrishnan, K., Lee, J.E.: Health and wellness monitoring through wearable and ambient sensors: exemplars from home-based care of elderly with mild dementia. Annales des Télécommunications **65**(9-10) (2010) 505–521

An Automated Wrapper-based Approach to the Design of Dependable Software

Matthew Leeke
Department of Computer Science
University of Warwick
Coventry, UK, CV4 7AL
matt@dcs.warwick.ac.uk

Arshad Jhumka
Department of Computer Science
University of Warwick
Coventry, UK, CV4 7AL
arshad@dcs.warwick.ac.uk

***Abstract*—The design of dependable software systems invariably comprises two main activities: (i) the design of dependability mechanisms, and (ii) the location of dependability mechanisms. It has been shown that these activities are intrinsically difficult. In this paper we propose an automated wrapper-based methodology to circumvent the problems associated with the design and location of dependability mechanisms. To achieve this we replicate important variables so that they can be used as part of standard, efficient dependability mechanisms. These well-understood mechanisms are then deployed in all relevant locations. To validate the proposed methodology we apply it to three complex software systems, evaluating the dependability enhancement and execution overhead in each case. The results generated demonstrate that the system failure rate of a wrapped software system can be several orders of magnitude lower than that of an unwrapped equivalent.**

***Keywords*-Importance, Metric, Replication, Variable, Wrappers**

I. Introduction

As computer systems become pervasive, our reliance on computer software to provide correct and timely services is ever-increasing. To meet these demands it is important that software be dependable [1]. It has been shown that a dependable software must contain two types of artefact; (i) error detection mechanisms (EDMs) and (ii) error recovery mechanisms (ERMs) [2], where EDMs are commonly known as detectors and ERMs as correctors. A detector is a component that asserts the validity of a predicate during execution, whilst a corrector is a component that enforces a predicate. Examples of detectors include runtime checks and error detection codes. Examples of correctors include exception handlers and retry [3]. During the execution of a dependable software, an EDM at a given location evaluates whether the corresponding predicate holds at that location, i.e., it attempts to detect an erroneous state. When an erroneous state is detected, an ERM will attempt to restore a suitable state by enforcing a predicate, i.e., it attempts to recover from an erroneous state. Using EDMs and ERMs it is possible to address the error propagation problem. A failure to contain the propagation of erroneous state across a software is known to make recovery more difficult [4].

The design of efficient EDMs [5] [6] [7] and ERMs [8] is notoriously difficult. Three key factors associated with this difficulty are (i) the design of the required predicate [6] [9], (ii) the location of that predicate [10], and (iii) bugs introduced by EDMs and ERMs. The problem of EDM and ERM design is exacerbated when software engineers are lacking in experience in software development or dependability mechanisms [7]. One approach to overcoming this difficulty is to reuse standard, efficient mechanisms, such as majority voting [11], in the design dependable software. However, techniques such as replication or N-version programming (NVP) [11] [12] are expensive, as they work at the software level, i.e., the whole software is replicated in some way. It would be ideal to adapt standard, efficient mechanisms to operate at a finer granularity in order to lessen overheads.

In this paper, we propose an automated methodology for the design of dependable software. Our approach is based on variable replication. This contrasts with current state-of-the-art approaches, which operate at a software level. The replication of software can be viewed as the replication of every variable and code component in a software. However, our approach focuses on replicating *important* variables. The proposed methodology works as follows: A lookup table in which variables are ranked according to their importance is generated. Once this is obtained, we duplicate or triplicate a subset of variables based on their importance using software wrappers, i.e., creating *shadow* variables. When an important variable is written to, the value held by the all relevant shadow variables is updated. When an important variable is read, its value is compared to those of its shadow variables, with any discrepancy indicating an erroneous state. Our approach induces a execution overhead ranging from 20%–35%, and a memory overhead ranging from 0.5%–20%.

The advantages of our approach over current state-of-the-art techniques are: (i) we circumvent the need to obtain non-trivial predicates by using standard efficient predicates, viz. majority voting (ii) we circumvent the need to know the optimal location of a given predicate by comparing values on all important variable reads, (iii) the efficiency of the standard mechanisms is known a priori, obviating the need for validation of the dependability mechanisms using fault injection [13], (iv) the overhead is significantly less than would be incurred by a complete software replication, and (v) we reduce the risk of inserting new bugs through detectors and correctors [8] [14].

A. Contributions

In this paper we make the following specific contributions:

- We describe an automated wrapper-based methodology for the design of dependable software, outlining the steps required for its application and providing insight into the use of the metrics on which it is based.
- We experimentally evaluate the effectiveness of the described methodology in the context of three complex software systems, obtaining results which serve to validate the usefulness of the metrics proposed in [15].
- We evaluate the execution overheads of the described methodology, offering insights into the relevant trade-offs between dependability and execution overheads.

II. Related Work

Software wrapper technology has been investigated in many fields, including computer security, software engineering, database systems and software dependability. In the context of computer security, software wrappers have been used to enforce a specific security policies [16] and protect vulnerable assets [17]. It has also been shown that security wrappers deployed within an operating system kernel can be used to meet application specific security requirements [18].

Software wrappers have been widely applied in the integration of legacy systems [19], where they act as connectors which allow independent systems to interact and the reconciliation of functionality [20] [21]. Examples of this can be found in the field of database systems, where software wrappers are used to encapsulate legacy databases [22] [23].

Software wrappers have been extensively investigated in the context of operating system dependability [24] [25], where emphasis is placed on wrapping device drivers and shared libraries [26] [27]. Software wrappers have also been used to address the more general problem of improving dependability in commercial-off-the-shelf software [28], as well as several more specific software dependability issues, such as the problem of non-atomic exception handling [14].

The proposed methodology is related to [29], where wrappers were used to detect contract violations in component-based systems. In contrast, the proposed methodology combines software wrappers that implement standard predicates and variable replication to enhance dependability. The variable-centric approach, facilitated by the metrics developed in [15], also differentiates the proposed methodology.

III. Models

In this section we present the models assumed in this paper.

A. Software model

A software system S is considered to be a set of interconnected modules $M_1 \dots M_n$. A module M_k contains a set of non-composite variables V_k, which have a domain of values, and a sequence of actions $A_{k1} \dots A_{ki}$. Each action in $A_{k1} \dots A_{ki}$ may read or write to a subset of V_k. Software is assumed to be grey-box, permitting source code access, but assuming no knowledge of functionality or structure.

B. Fault model

A fault model has been shown to contain two parts: (i) a local part, and (ii) a global part [30]. The local fault model, called the *impact model*, states the type of faults likely to occur in the system, while the global model, called the *rely specification*, states the extent to which the local fault model can occur. The rely specification constrains the occurrence of the local model so that dependability can be imparted. For example, a rely specification will state that “at most f of n nodes can crash”, or “faults can occur only finitely often”. Infinite fault occurrences can only be tolerated by infinite redundancy, which is impossible.

In this paper we assume a *transient* data value fault model [31]. Here, the local fault model is the transient data value failure, i.e., a variable whose value is corrupted, and that this corruption may ever occur again. The global fault model is that we assume that any variable can be affected by transient faults. The transient fault model is used to model hardware faults in which bit flips occur in memory areas that causes instantaneous changes to values held in memory. A transient data value fault model is often assumed during dependability analysis because it can be used to mimic more severe fault models [31], thus making it a good base fault model.

IV. Methodology

The proposed methodology is based on the premise that the replication of important variables can yield significant reduction in failures without incurring significant execution overheads. The importance of variables is based on their implication in error propagation. The methodology is a three step process. First, a table ranking variables according to their importance for a given module is generated. Next, all read and write operations to important variables, as defined by a threshold value, are identified. Finally, such operations are wrapped using specifically designed software wrappers. An overview of the methodology is shown in Figure 1. Sections IV-A-IV-C provide a description of these steps.

A. Step 1: Establishing Variable Importance

The first step is to evaluate the relative importance of each variable within the software module to be wrapped. To achieve this we use the metric suite in [15] to measure importance. The importance metric is a function of two sub-metrics, spatial and temporal impact, and system failure rate.
Spatial Impact: Given a software whose functionality is logically distributed over a set of modules, the spatial impact of variable v in module M for a run r, denoted as $\sigma^r_{v,M}$, is the number of modules corrupted in r. Thus, the spatial impact of a variable v of module M, denoted $\sigma_{v,M}$, is:

$$\sigma_{v,M} = max\{\sigma^r_{v,M}\}, \forall r \tag{1}$$

Thus, $\sigma_{v,M}$ captures the diameter of the affected area when variable v in module M is corrupted. The higher $\sigma_{v,M}$ is, the more difficult it is to recover from the corruption. As the metric captures the diameter of the area affected by the propagation of errors, low values are desirable.

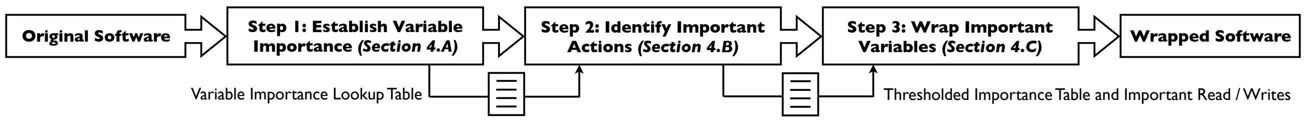

Original Software
Step 1: Establish Variable Importance (Section 4.A)
Step 2: Identify Important Actions (Section 4.B)
Step 3: Wrap Important Variables (Section 4.C)
Wrapped Software
Variable Importance Lookup Table
Thresholded Importance Table and Important Read / Writes

Figure 1. Methodology overview

Temporal Impact: Given a software whose functionality is logically distributed over a set of modules, the temporal impact of variable v in module M for a run r, denoted as $\tau^r_{v,M}$, is the number of time units over which at least one module remains corrupted in r. Thus, the temporal impact of a variable v of module M, denoted $\tau_{v,M}$ is:

$$\tau_{v,M} = max\{\tau^r_{v,M}\}, \forall r \qquad (2)$$

Thus, $\tau_{v,M}$ captures the period that program state remains affected when variable v in module M is corrupted. The higher $\tau_{v,M}$ is, the likelier a failure is to occur. As the metric captures the persistence of errors, low values are desirable.
Importance Metric: The importance metric, as instantiated in [15], is defined for variable v in module M with variable specific system failure rate f as:

$$I_{v,M} = \frac{1}{(1-f)^2}\Big(\frac{\sigma_{v,M}}{\sigma_{max}} + \frac{\tau_{v,M}}{\tau_{max}}\Big)^1 \qquad (3)$$

The importance of all variables in a module can be evaluated using Equations 1-3. In [15] fault injection was used to estimate variable importance, though the metric can be evaluated using alternative approaches. Note that the variable ranking generated for each module is relative to that module, which means that these values should not be compared on an inter-module basis. Once this first step of the proposed methodology has been completed, a lookup table relating any given variable to its importance value can be generated.

B. Step 2: Identifying Important Actions

The next step is to identify all read and write actions on important variables. As the replication of a whole software, or indeed every variable, incurs a large overhead, we select a subset of the most important variables for replication using thresholds. We set two thresholds to govern the number of duplicated and triplicated variables; λ_d and λ_t respectively. Thresholds may be defined with respect to importance values, though in many situations, it is reasonable to define thresholds as a proportion of the variables in a module. For example, to triplicate the top 10% and duplicate the top 15% of variables, we would set $\lambda_t = 0.10$ and $\lambda_d = 0.15$. The use of two thresholds is motivated by the desire to reduce replication overhead and provide flexibility in the application of the proposed methodology.

Once threshold values have been set, the variables to be wrapped can be identified. However, before wrapping, each possible read and write location on an important variable must be identified. This can be achieved by several means, including system call monitoring and memory management techniques. The only requirement is that all possible read and write actions on important variables must be identified. In this paper, source code analysis is used to identify important read and write actions, as detailed in Section VI. Completing this step will mean that variables to be wrapped have been identified and a mechanism has been used, or is in place, to identify read and write actions on those variables.

Algorithm 1 Write-Wrapper: Writing a variable v

```
$v := f(\ldots)$
if (rank($v$) $\geq \lambda_t$) then
    create($v'$);
    create($v''$);
    $v, v', v'' := f(\ldots)$;
else if (rank($v$) $\geq \lambda_d$) then
    create($v'$);
    $v, v' := f(\ldots)$;
end if
```

Algorithm 2 ReadWrapper: Reading a variable v

```
$y := g(v, \ldots)$;
if (rank($v$) $\geq \lambda_t$) then
    $y :=$ g(majority($v, v', v''$), ...);
else if (rank($v$) $\geq \lambda_d$) then
    $y :=$ g(random($v, v'$), ...);
end if
```

C. Step 3: Wrapping Important Variables

Two types of software wrapper are employed by the methodology; *write*-wrappers and *read*-wrappers. Pseudocode for these wrappers is shown in Algorithms 1 and 2.
Write-Wrapper: This software wrapper is invoked when an important variable is written. When a variable v is assigned a value $f(\ldots)$, where f is some function, in the unwrapped module, the ranking of the variable is checked. If the rank of v is in the top λ_t, then two shadow variables, v' and v'', are created. Alternatively, if the rank of v is between λ_t and λ_d, then a shadow variable v' is created. Then, v and all of its shadow variables are updated with $f(\ldots)$.
Read-Wrapper: This software wrapper is invoked when an important variable is read. When a variable y is updated with a function $g(v, \ldots)$ in the unwrapped module, where g is a function and variable v is to be read, the rank of v is checked against λ_t. If the rank of v is greater than λ_t then function g uses the majority of the v, v', v''. If the rank of v is between λ_t and λ_d, then g uses v or v'.

V. Experimental Setup

In this section we detail the experimental setup used in the estimation of the importance metric for each target system.

A. Target Systems

7-Zip (7Z): The 7-Zip utility is a high-compression archiver supporting a variety of encryption formats [32]. The system is widely-used and has been developed by many different software engineers. Most project source code is available under the GNU Lesser General Public License.

Flightgear (FG): The FlightGear Flight Simulator project is an open-source project that aims to develop an extensible yet highly sophisticated flight simulator [33]. The system is modular, contains over 220,000 lines of code and simulates a situation where dependability is critical. All project source code are available under the GNU General Public License.

Mp3Gain (MG): The Mp3Gain analyser is an open-source volume normalisation software for mp3 files [34]. The system is modular, widely-used and has been predominantly developed by a single software engineer. All project source code are available under the GNU General Public License.

B. Test Cases

7Z: An archiving procedure was executed in all test cases. A set of 25 files were input to the procedure, each of which was compressed to form an archive and then decompressed in order to recover the original content. The temporal impact of faults was measured with respect to the number of files processed. For example, if a fault were injected during the processing of file 15 and persisted until the end of a test case, its temporal impact would be 10. To create a varied system load, the experiments associated with each instrumented variable were repeated for 25 distinct test cases, where each test case involved a distinct set of 25 input files.

FG: A takeoff procedure was executed in all test cases. The procedure executed for 2700 iterations of the main simulation loop, where the first 500 iterations correspond to an initialisation period and the remaining 2200 iterations correspond to pre-injection and post-injection periods. A control module was used to provide a consistent input vector at each iteration of the simulation. To create a varied and representative system load, the experiments associated with each instrumented variable were repeated for 9 distinct test cases; 3 aircraft masses and 3 wind speeds uniformly distributed across 1300-2100lbs and 0-60kph respectively.

MG: A volume-level normalisation procedure was executed in all test cases. The procedure took a set of 25 mp3 files of varying sizes as input and normalised the volume across each file. The temporal impact of injected faults was measured with respect to the number of files processed. To create a varied system load, the experiments associated with each instrumented variable were repeated for 3 distinct test cases, where each test case used a distinct set of 25 input files.

C. System Instrumentation, Fault Injection and Logging

Instrumented modules in each target system were chosen randomly from modules used in the execution of the aforementioned test cases. A summary of instrumented modules is given in Table I. The number of variables instrumented for each module accounted for no less than 90% of the total number of variables in that module. All code locations where an instrumented variable could be read were instrumented for fault injection. Those variables and locations not instrumented were associated with execution paths that would not be executed under normal circumstances, e.g., test routines.

Table I
Summary of Instrumented Software Modules

Module	Target System	Module Name
7Z1	7-Zip	LZMADecode
7Z2	7-Zip	7zInput
7Z3	7-Zip	7zFileHandle
FG1	FightGear	FGInter
FG2	FightGear	FGPropulsion
FG3	FightGear	FGLGear
MG1	Mp3Gain	NLaunch
MG2	Mp3Gain	GainAnalysis
MG3	Mp3Gain	Decode

Fault injection was used to determine the spatial and temporal impact associated with each software module [15]. The Propagation Analysis Environment was used for fault injection and logging [35]. A *golden run* was created for each test case, where a golden run is a reproducible fault-free run of the system for a given test case, capturing information about the state of the system during execution. Bit flip faults were injected at each bit-position for all instrumented variables. Each injected run entailed a single bit-flip in a variable at one of these positions, i.e. no multiple injections. For FG each single bit-flip experiment was performed at 3 injection times uniformly distributed across the 2200 simulation loop iterations that followed system initialisation, i.e. 600, 1200 and 1800 control loop iterations after initialisation. For 7Z and M3, each single bit-flip experiment was performed at 25 distinct injection times uniformly distributed across the 25 time units of each test case. The state of all modules used in the execution of all test cases was monitored during each fault injection experiment. The data logged during fault injection experiments was then compared with the corresponding golden run, with any deviations being deemed erroneous and thus contributing to variable importance.

D. Failure Specification

7Z: A test case execution was considered a failure if the set of archive files and recovered content files were different from those generated by the corresponding golden run.

FG: A failure specification was established using of golden run observation and relevant aviation information. A failed execution was considered to fall into at least one of three categories; speed failure, distance failure and angle failure. A run was considered a speed failure if the aircraft failed to reach a safe takeoff speed after first passing through critical speed and velocity of rotation. A run was considered a distance failure if the takeoff distance exceeds that specified by the aircraft manufacturer, where the distance is increased by 10 meters for every additional 200lbs over the aircraft

Table II
IMPORTANCE RANKING FOR 7Z1 VARIABLES

Rank	Variable	Importance	Failure Rate
1	processedPos	0.012869318	0.009893411
2	remainLen	0.010028409	0.009865020
3	distance	0.010085227	0.004867079
4	posState	0.008380682	0.004858712
5	ttt	0.006903409	0.004851485

base-weight. A run was considered an angle failure if a pitch rate of 4.5 degrees is exceeded before the aircraft is clear of the runway or the aircraft stalls during climb out.
MG: A test case execution was considered a failure if the set of normalised output files were different from those generated by the corresponding golden run.

VI. RESULTS

The importance metric can be evaluated using many different approaches, including static analysis and the evaluation of data-flow. In this paper, as in [15], importance is measured using fault injection. Fault injection is a dependability validation approach, whereby the response of a system to the insertion of faults is analysed with respect to a given oracle. Fault injection is typically used to assess the coverage and latency of error detection and correction mechanisms.

Using the approach outlined in Section V, the spatial and temporal impact of each variable was estimated. This information, and the failure rate for fault injections on each variable, was used to evaluate the importance of all variables according to Equation 3. Tables II-X show the importance ranking of all, subsequently identified, important variables for each target modules. For each variable, the *Importance* column gives the value of the importance metric, whilst *Failure Rate* is the proportion of fault injected execution that caused a system failure. Note that failure rate is assessed on a per variable basis. For example, if a variable is subject to 100 fault injected executions and 25 of these result in a system failure, then the it has a failure rate of 0.25.

The entries in Tables II-X give importance values for variable identifies as important in each module. To perform the thresholding required for this identification, we set $\lambda_d = 0.15$ and $\lambda_t = 0.10$. This meant that, for each software module, the top 15% of variables were to be wrapped, with the top 10% being triplicated and the next 5% being duplicated. For example, Table II shows 15% of the 36 variables in module 7Z, where the top three variables were triplicated and the rest were duplicated. The chosen threshold values were selected in order ensure that at least one variable in each module was wrapped, though the choice of λ_d and λ_t will typically be situation specific.

Once the importance table for each module had been thresholded, source code analysis was used to identify read and write actions on important variables. The implementation of our source code analyser was based on the premise that writes and reads to important variable are the only operation types that are deemed to be important actions. When adopting a source code analysis approach it must be

Table III
IMPORTANCE RANKING FOR 7Z2 VARIABLES

Rank	Variable	Importance	Failure Rate
1	numberStreams	0.757575163	0.013881579
2	highPart	0.699089580	0.015526316
3	unpack	0.453218118	0.010994318
4	sizeIndex	0.379870331	0.002755682
5	i_unpack	0.248907060	0.002698864
6	attribute	0.141369792	0.018011364
7	numInStreams	0.099065565	0.002443182
8	numSubstream	0.099059922	0.002386364

Table IV
IMPORTANCE RANKING FOR 7Z3 VARIABLES

Rank	Variable	Importance	Failure Rate
1	seekInStreamS	0.009250000	0.382360363

Table V
IMPORTANCE RANKING FOR FG1 VARIABLES

Rank	Variable	Importance	Failure Rate
1	vTrueKts	0.056881	0.003472
2	runAltitude	0.039179	0.002778
3	vGroundSpeed	0.035471	0.208333
4	alpha	0.033359	0.004629

Table VI
IMPORTANCE RANKING FOR FG2 VARIABLES

Rank	Variable	Importance	Failure Rate
1	currentThrust	1.047348000	0.010417000
2	hasInitEngines	1.016663000	0.003472000
3	numTanks	1.012560000	0.004630000
4	totalQuanFuel	1.011618000	0.004167000
5	firsttime	1.009914000	0.001736000
6	dt	0.506376000	0.005208000

Table VII
IMPORTANCE RANKING FOR FG3 VARIABLES

Rank	Variable	Importance	Failure Rate
1	compressLen	0.730128000	0.013889000
2	groundSpeed	0.433243000	0.001984000
3	steerAngle	0.053254000	0.011111000

Table VIII
IMPORTANCE RANKING FOR MG1 VARIABLES

Rank	Variable	Importance	Failure Rate
1	selfWrite	0.283413927	0.028650000
2	bitridx	0.278821206	0.012650000
3	whiChannel	0.277626178	0.008400000
4	gainA	0.160324478	0.016700000
5	curFrame	0.160096536	0.015300000
6	inf	0.160035590	0.014925000
7	cuFile	0.099405049	0.005850000

recognised that any analysis tool must work under the assumption that any unidentifiable operation type could be an action relating to any important variable. This conservative stance ensures that the coverage of the process is maximised, though unnecessary overheads may be incurred.

Following the identification of read and write actions on

Table IX
IMPORTANCE RANKING FOR MG2 VARIABLES

Rank	Variable	Importance	Failure Rate
1	sampleWin	1.337694959	0.194400000
2	batchSample	0.988385859	0.031100000
3	curSamples	0.925373931	0.008350000
4	first	0.923418424	0.006250000

Table X
IMPORTANCE RANKING FOR MG3 VARIABLES

Rank	Variable	Importance	Failure Rate
1	maxAmpOnly	1.131021387	0.011825000
2	dSmp	0.683939300	0.009200000
3	winCont	0.678189611	0.000800000

important variables, the software wrappers described in Section IV-C were deployed. As the locations for read-wrapper and write-wrapper deployment were necessarily consistent with the code locations of important read and write actions respectively, information generated during source code analysis was used to drive wrapper deployment.

Figures 2 and 3 show examples of read-wrapper and write-wrapper deployments. The first line in each figure shows the original program statement before wrapping. The second line in each figure illustrates the use of wrappers. In Figure 2 the *dt* variable is being read-wrapped, whilst Figure 3 shows the *currentThrust* variable being write-wrapped. Observe that, in both cases, it is necessary to provide the wrapping functions with identifiers for the variable and location. This information is generated, maintained and known only to the wrapping software following the identification of important read and write actions, which means that it has no discernible impact on the execution of the target system.

To validate the effectiveness of the proposed methodology, the fault injection experiments described in Section V were repeated on wrapped target systems. Only one module in any target system has its important variables wrapped at any time. Table XI summarises the impact that the proposed methodology had on the dependability of all target modules. The *Unwrapped Failure Rate* column gives the original system failure rate with respect to all fault injection experiments, i.e., the proportion of failures of the unwrapped system when fault injection across all variables in the given module are considered. The *Wrapped Failure Rate* column then gives the same statistic for wrapped modules.

Observe from Table XI that the system failure rate of each module decreased in all cases, thus demonstrating the effectiveness the methodology. Further, the decrease in system failure rate of many modules is greater than combined failure rates of the wrapped variables in those modules. For example, module MG3 had an unwrapped failure rate of 0.002780830, which corresponded to 39361 failures. The same module had a wrapped failure rate of 0.000006105, corresponding to 86 failures. This improvement can not be accounted for by the 1142 failures incurred by the three wrapped variables, thus there is evidence that the error propagation problem has been addressed. This observation is particularly relevant to limiting the propagation of erroneous states that originate "upstream" of a target module.

```
/* tankUPD = calc + (dt * rate); */
tankUPD = calc + (readWrapper(VARID_12, LOCID_4, dt) * rate);
```

Figure 2. Read wrapper example deployment

```
/* currentThrust = Engines[i]->GetThrust();*/
currentThrust = writeWrapper(VARID_17, LOCID_8, Engines[i]->GetThrust());
```

Figure 3. Write wrapper example deployment

Table XI
SYSTEM FAILURE RATES ASSOCIATED WITH ALL FAULT INJECTED EXECUTIONS OF INSTRUMENTED MODULES

Module	Unwrapped Failure Rate	Wrapped Failure Rate
7Z1	0.002407940	0.000017397
7Z2	0.007082023	0.000141946
7Z3	0.000856604	0.000030189
FG1	0.004582688	0.000045475
FG2	0.002481621	0.000002047
FG3	0.001471873	0.000135395
MG1	0.004983750	0.000012083
MG2	0.007888044	0.000013426
MG3	0.002780792	0.000006076

Table XII
PEAK INCREASE IN EXECUTION TIME AND MEMORY USAGE INCURRED BY WRAPPERS (SHOWN AS % INCREASES FOR EACH MODULE)

Module	Execution Time (Peak % Increase)	Memory Usage (Peak % Increase)
7Z1	26.048%	07.55%
7Z2	31.470%	18.16%
7Z3	20.359%	00.94%
FG1	30.660%	20.63%
FG2	35.829%	03.32%
FG3	23.529%	02.03%
MG1	25.983%	05.22%
MG2	28.090%	04.93%
MG3	23.174%	00.58%

Table XII summarises the overhead of the proposed methodology on all target modules. The *Execution Time* column gives the peak percentage increase in runtime when comparing executions of the wrapped and unwrapped target modules. The *Memory Usage* column gives the peak percentage increase in memory consumption when comparing executions of the wrapped and unwrapped target modules. All overheads were measured by monitoring target modules in isolation using the Microsoft Visual Studio Profiler.

Observe from Table XII that the execution overhead of wrapped modules varies between 20% and 35%. The worst case absolute increase in the execution time of a module was observed for module 7Z2, which increased by 31.470% to approximately 28μs. There is a coarse correlation between the increase in execution time and the number of variables in each module, though the frequency with which each variable is used is likely to impact this overhead more directly. The increases in memory consumption are more varied than increases in execution time, with the maximum and minimum increases being 20.63% and 0.94% respectively. Note that the memory usage increases shown are the peak observed increases for each module. This means that the increase is unlikely to be sustained beyond the execution of a module and the relative scale of an increase may be small. For example, the 20.63% increase in memory consumption shown for FG1 module corresponds to an additional overhead of less than 4KB.

VII. Discussion

Inserting detection and correction mechanisms directly into a software system is likely to result in a low overhead, due to the fact that only a small number of variables and code segments must be added or replicated. However, as argued earlier, this approach necessitates the design of non-trivial predicates, which is known to be difficult [7]. Also, it is known that the design of correctors often introduces additional bugs into software [14]. The proposed methodology circumvents these problems by (re)using standard efficient detectors and correctors, though this comes at the cost of greater overheads. We see this problem as a tradeoff; inserting mechanisms directly is more difficult and error prone but imposes less overhead, whilst our approach can reuse simpler mechanisms at the expense of greater overheads.

The performance overheads of the proposed methodology will vary according to the extent of variable wrapping performed, i.e., according to λ_d and λ_t. Overhead comparison with similar approaches are desirable, but generally invalid due to difference in the extent, intention and focus of the wrapping mechanisms. For example, the results presented in this paper demonstrate that with $\lambda_d = 0.15$ and $\lambda_t = 0.10$, for a single module measured in isolation, our approach introduces a additional runtime overhead ranging from 20%-35% and a memory overhead ranging from 0.5%-20%. In contrast, the approach developed in [14] had a memory overhead for the masking of a fixed-duration function (5μs) of over 1200%. However, the component / object focus of this approach, as opposed to the novel variable-centric focus developed in this paper, invalidates this comparison.

Given that the software wrappers operate by updating replicated variables during writes and choosing a majority value during reads, our approach will work with variables of different types whenever the notion of equality exists or can be defined for that type. This is well-defined for integer, real and boolean types, which were the ones mostly encountered in the case studies presented, but there is no reason why the notion of equality can not be defined for composite types.

To prevent bias, the target modules in this paper were selected randomly. In reality, module selection could be based on expert knowledge, an understanding of system structure and dependability properties. For example, in systems where a given module is known to act as a "hub", it would be come a candidate for wrapping. Dependability frameworks can also be used to inform module selection [36].

The main limitation of the proposed methodology, as it has been applied in this paper, is the need for source code access. Although no attempt has been made to constrain the means by which methodology steps can be met, it may be difficult to devise an appropriate combination of means when source code is not available. For example, the identification of read and write actions on important variables was performed using source code analysis. In situations where source code is not available this is not possible, meaning that an alternative approach must be employed. However, it should be remembered that the intention of the methodology is to aid in the design of dependable software during its development, when source code is normally available.

VIII. Conclusion

In this paper we developed an automated wrapped-based approach for the design of dependable software. The novelty of the approach is in the reuse of standard efficient dependability mechanisms at the level of variables, which has been enabled by the use of software wrappers that have been built on a dynamic error propagation metric. The use of wrappers is justified by the fact that we do not require knowledge of system implementation in order to apply the methodology. The propose methodology was validated through in-depth studies of several complex software systems, each drawn from a different application domain. This application of the methodology yielded promising results, with all treated modules exhibiting significant dependability improvements.

References

[1] J.-C. Laprie, *Dependability: Basic Concepts and Terminology*. Springer-Verlag, December 1992.

[2] A. Arora and S. S. Kulkarni, "Detectors and correctors: A theory of fault-tolerance components," in *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, May 1998, pp. 436–443.

[3] Y. M. Wang, Y. Huang, and W. K. Fuchs, "Progressive retry for software error recovery in distributed systems," in *Proceedings of the 23rd International Symposium on Fault Tolerant Computing*, June 1993, pp. 138–144.

[4] A. Arora and M. Gouda, "Distributed reset," *IEEE Transactions on Computers*, vol. 43, no. 9, pp. 1026–1038, September 1994.

[5] A. Arora and S. Kulkarni, "Designing masking fault-tolerance via nonmasking fault-tolerance," in *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems*, June 1995, pp. 435–450.

[6] A. Jhumka, F. Freiling, C. Fetzer, and N. Suri, "An approach to synthesise safe systems," *International Journal of Security and Networks*, vol. 1, no. 1, pp. 62–74, September 2006.

[7] N. G. Leveson, S. S. Cha, J. C. Knight, and T. J. Shimeall, "The use of self checks and voting in software error detection: An empirical study," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 432–443, April 1990.

[8] H. Shah, C. Gorg, and M. J. Harrold, “Understanding exception handling: Viewpoints of novices and experts,” *IEEE Transaction on Software Engineering*, vol. 36, no. 2, pp. 150–161, March 2010.

[9] A. Arora and S. S. Kulkarni, “Component based design of multitolerant systems,” *IEEE Transactions on Software Engineering*, vol. 24, no. 1, pp. 63–78, January 1998.

[10] M. Hiller, A. Jhumka, and N. Suri, “An approach for analysing the propagation of data errors in software,” in *Proceedings of the 31st IEEE/IFIP International Conference on Dependable Systems and Networks*, July 2001, pp. 161–172.

[11] A. Avizienis, “The n-version approach to fault-tolerant software,” *IEEE Transaction on Software Engineering*, vol. 11, no. 12, pp. 1491–1501, December 1985.

[12] A. Avizienis and L. Chen, “On the implementation of n-version programming for software fault tolerance during execution,” in *Proceedings of the 1st IEEE-CS International Computer Software Applications Conference*, November 1977, pp. 149–155.

[13] M. Hsueh, T. K. Tsai, and R. K. Iyer, “Fault injection techniques and tools,” *IEEE Computer*, vol. 30, no. 4, pp. 75–82, April 1997.

[14] C. Fetzer, P. Felber, and K. Hogstedt, “Automatic detection and masking of nonatomic exception handling,” *IEEE Transactions on Software Engineering*, vol. 30, no. 8, pp. 547–560, August 2004.

[15] M. Leeke and A. Jhumka, “Towards understanding the importance of variables in dependable software,” in *Proceedings of the 8th European Dependable Computing Conference*, April 2010, pp. 85–94.

[16] P. Sewell and J. Vitek, “Secure composition of untrusted code: Wrappers and causality types,” in *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, July 2000, pp. 269–284.

[17] S. Cheung and K. N. Levitt, “A formal-specification based approch for protecting the domain name system,” in *Proceedings of the 30th IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2000, pp. 641–651.

[18] T. Mitchem, R. Lu, and R. O’Brien, “Using kernel hypervisors to secure applications,” in *Proceedings of the 13th Annual Confernece on Computer Security Applications*, December 1997, pp. 175–181.

[19] E. Wohlstadter, S. Jackson, and P. Devanbu, “Generating wrappers for command line programs: The cal-aggie wrap-o-matic project,” in *Proceedings of the 23rd ACM/IEEE International Conference on Software Engineering*, May 2001, pp. 243–252.

[20] A. C. Marosi, Z. Balaton, and P. Kacsuk, “Genwrapper: A generic wrapper for running legacy applications on desktop grids,” in *Proceedings of the 23rd International Symposium on Parallel and Distributed Computing*, May 2009, pp. 1–6.

[21] B. Spitznagel and D. Garlan, “A compositional formalization of connector wrappers,” in *Proceedings of the 25th ACM/IEEE International Conference on Software Engineering*, May 2003, pp. 374–384.

[22] A. Cleve, “Automating program conversion in database reengineering: A wrapper-based approach,” in *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, March 2006, pp. 323–326.

[23] P. Thiran, J. Hainaut, and G. Houben, “Database wrappers development: Towards automatic generation,” in *Proceedings of the 9th European Conference on Software Maintenance and Reengineering*, March 2005, pp. 207–216.

[24] A. K. Ghosh, M. Schmid, and F. Hill, “Wrapping windows nt software for robustness,” in *Proceedings of the 29th Annual Symposium on Fault Tolerant Computing*, June 1999, pp. 344–347.

[25] A. S. Tanenbaum, J. N. Herder, and H. Bos, “Can we make operating systems reliable and secure?” *Computer*, vol. 39, no. 5, pp. 44–51, May 2006.

[26] C. Fetzer and Z. Xiao, “An automated approach to increasing the robustness of c libraries,” in *Proceedings of the 32nd IEEE/IFIP International Conference on Dependable Systems and Networks*, December 2002, pp. 155–164.

[27] M. Susskraut and C. Fetzer, “Robustness and security hardening of costs software libraries,” in *Proceedings of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2007, pp. 61–71.

[28] F. Salles, M. Rodriguez, J.-C. Fabre, and J. Arlat, “Metakernels and fault containment wrappers,” in *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, November 1999, pp. 22–29.

[29] S. H. Edwards, M. Sitaraman, and B. W. Weide, “Contract-checking wrappers for c++ classes,” *IEEE Transactions on Software Engineering*, vol. 30, no. 11, pp. 794–810, November 2004.

[30] H. Volzer, “Verifying fault tolerance of distributed algorithms formally - an example,” in *Proceedings of the 1st International Conference on the Application of Concurreny to System Design*, March 1998, pp. 187–197.

[31] D. Powell, “Failure model assumptions and assumption coverage,” in *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, July 1992, pp. 386–395.

[32] 7-Zip, “http://www.7-zip.org/,” February 2011.

[33] FlightGear, “http://www.flightgear.org/,” February 2011.

[34] MP3Gain, “http://mp3gain.sourceforge.net/,” February 2011.

[35] M. Hiller, A. Jhumka, and N. Suri, “Propane: An environment for examining the propagation of errors in software,” in *Proceedings of the 11th ACM SIGSOFT International Symposium on Software Testing and Analysis*, July 2002, pp. 81–85.

[36] ——, “Epic: profiling the propagation and effect of data errors in software,” *IEEE Transactions on Computers*, vol. 53, no. 3, pp. 512–530, May 2004.

An Approach for the Reliability Analysis of Automotive Control Systems

Mikhail Glukhikh, Mikhail Moiseev
St. Petersburg State Polytechnical University
St. Petersburg, Russia
glukhikh@mail.ru, mikhail.moiseev@gmail.com

Harald Richter
Clausthal University of Technology
Clausthal, Germany
hri@tu-clausthal.de

***Abstract*—In this paper, we present an approach and a tool that automates and thereby accelerates the most time-consuming phases of reliability engineering. In this approach, an operational function is computed automatically from a high-level system description by using of system components' properties, fault types propagation rules and other auxiliary information. The tool allows arbitrary component types, any component couplings and failure types and covers thus mandatory features for a profound reliability analysis. It calculates the mean time to failure, the mean fault number and the components' influence on the overall reliability as system reliability characteristics. This tool was tested by a major car manufacturer in an embedded electronic system of a car. The main advantage of the developed tool is that it simplifies reliability analysis of complex-structured systems using a novel method for system operational state description.**

Keywords-reliability analysis; automotive control system.

I. Introduction

In reliability engineering, there are two different tasks to accomplish: the analysis of the reliability of a given system, and the synthesis of a successor system that is more reliable than the first one, under the boundary conditions of the costs needed to produce it. Reliability analysis is used iteratively many times in the process of system design. It is thus important to evaluate the reliability parameters quickly and accurately with minimal effort [1][2].

This paper presents a new approach and a tool for automating the reliability analysis of complex-structured control systems. Our approach is based on a system meta-model, which allows to represent many classes of control systems. As an example, the application of this approach for automotive control systems is given in this paper.

Analyzing the reliability of car electronics becomes more and more important because of two reasons. First, the number of Electronic Controller Units (ECUs) that are built into contemporary cars has already reached the amount of 100, thus reducing the mean time to failure (MTTF) by the pure quantity of components. Second, the passengers' safety depends more and more exclusively on the reliability of ECUs' hardware and software, together with other components such as sensors, actuators, cable trees and connectors, power supplies, generator and battery [3][4].

Because of the used general methodology our approach is not restricted to cars only but can also be applied to other technical systems, such as in medicine, aerospace and nuclear power plants, where harsh environment conditions are prevailing or where system breakdown is unacceptable.

The rest of the paper is organized as follows: Section 2 shows state-of-the-art, Section 3 summarizes related work in this field, Section 4 describes the main idea of our approach, Section 5 presents its specialization for automotive applications, Section 6 describes the used reliability analisys methods, the tool itself is explained in Section 7, the paper ends in Section 8 with a conclusion and an outlook to future work.

II. State-of-the-Art

The goal of reliability engineering is to achieve either a prescribed reliability for a planned system or a higher reliability at lower costs. State-of-the-art to achieve this is to change the system structure, to add more reliable components, or to add redundancy such as standby reserve, hot reserve, standby containers and load sharing containers.

Reliability analysis of a technical system normally needs many iteration cycles in which the so-called survival or reliability function is calculated multiple times, together with other important parameters such as the MTTF, the mean fault number and the components' influence on the overall reliability. These parameters allow to identify weaknesses in the system design. The reliability function is a probabilistic function over time that is based on the individual failure rates of the system components. For the calculation of the reliability function, the system has to be modelled first. Standard models are reliability block diagrams (RBDs), as well as fault trees (with or without Markov chains) and event trees as alternatives [5]. Another common model is the operational function [6] which is used by our approach.

In practice, automotive control systems are complex with respect to their interconnect topology between components and because of the sheer amount of parts. High-end cars have already several thousands of cables and connectors. Furthermore, there are different types of data paths and power lines that couple the components together, and many component types that have to be differentiated as well.

Typical automotive control systems can not be decomposed into a set of elementary serial or parallel circuits of components. Instead, loops and crossings of links exist in the interconnect topology. The system topology is thereby considered to be a general graph.

State-of-the art in current tools is that the constructing of reliability models is hardly automated yet. However, the manual constructing of the models mentioned is too time-consuming in practice because of the large size and the high complexity of real-world system topology. We propose an automated approach based on a configurable high-level system description and component parametrization.

III. Related Work

Beside research projects, there are several commercially avaliable tools for the reliability analysis of complex systems such as [11][12][13][14][15].

Usually, these tools support a comprehensive range of analysis methods. According to [5], the most common methods are fault tree analysis (FTA), which may be preceded by an RBD system model, and event tree analysis (ETA). FTA can be combined with Markov chains (MC) too. The most important features of these commercial tools are depicted in Table 1.

Table I
Reliability analysis tools

Tool	RBD	ETA	FTA	MC
ITEM ToolKit	+	+	+	+
RAM Commander	+	+	+	+
Isograph FaultTree+	−	+	+	+
PTC Relex	−	+	+	+
ReliaSoft BlockSim 7	+	−	+	−

All tools suffer from the manual definition of the fault tree for FTA. Already from 10 components on, the effort for fault tree construction by hand is high because its time-complexity grows exponentially. In some cases known from practice, a preceded RBD with subsequent conversion into a fault tree may simplify this task [9]. However, in complex cases, the RBD construction complexity is already comparable with fault tree construction complexity.

The other model used for fault tree synthesis automation is Fault Tolerant Data Flow (FTDF) [10]. Fault tree synthesis algorithm traverses FTDF graph finding all event combinations which lead to system failure. However, this algorithm does not support cyclic dependencies between elements and uses exhaustive search for synthesis.

Reliability parameters are usually calculated by using the system reliability function. There are several possibilites to obtain the reliability function out of a previously established fault tree or out of an operational function. These possibilites are:

1) Selection of a minimal-cut set in the disjunctive normal form of the boolean description of the fault tree and subsequent use of the inclusion/exclusion principle as described by [5][7].
2) Establishing a binary decision tree as an intermediate data structure of the fault tree as proposed by [7][8].
3) Using a substitution form of the operational function, together with boolean-probabilistic transformation rules that are described in [6][1].

All three ways exhibit in the general case an exponential complexity as soon as the system size increases. However, we found out that in practice the boolean-probabilistic method has good scalability which is why we used it in our tool.

IV. Semi-Automated Analysis

We suggest a new approach for the semi-automatic reliability analysis that consists of three major steps:

1) An abstract meta-model.
2) An application-specific configuration of the meta-model by a so-called pattern.
3) A set of boolean-probabilistic transformations.

Our first step employs an abstract meta-model instead of a concrete RBD, fault tree or structure graph. This meta-model is defined by the tuple $M = \langle T, G, I, F, O \rangle$, with T is the set of component types, G is the graph of the system topology, I is the set of failure types each component type can have, F is the set of failure propagation rules for every failure type, and O is a set of rules for constructing the operational function.

Step 2 in our approach yields a concrete description of a given system from the meta-model, after having configured the model with an application-specific pattern. The pattern has to be created by hand, together with the system's topology. The pattern we used for automotive control systems is given in detail in Section 5. Other control systems need different patterns but the meta-model can remain the same.

Because of the time-consuming manual definition of a fault tree its generation should be automated. We decided to use an operational function for the system operability description (see Section 6). From the operational function, a reliability function can be derived automatically. To achieve this, boolean-probabilistic transformations are employed.

The approach requires in detail the following 6 phases:

1) Definition of all component types, failure types, failure propagation rules and operational function constructing rules as the application-specific pattern.
2) Construction of the graph of the system topology.
3) Definition of all failure rates.
4) Derivation of the operational function.
5) Conversion of the operational function into the reliability function.
6) Computation of the MTTF, the mean fault number and the components influence on the overall reliability.

The first phase must be performed once by hand for each application-specific pattern. Furtherfore, for any given system, phases 2-3 also have to be performed manually once.

Phases 4-6 are computed automatically. These are the steps that are repeated multiple times for reliability engineering which is why we automated their execution.

V. Pattern for Automotive Applications

Modern cars may contain the following assistance systems for driver and infotainment: motor management, electronic stabilization (ESP) with or without active steering, adaptive cruise control (ACC), speed control, distance control, rear vision, night vision, lane keeping, lane changing, parking, navigation etc. These systems can be analysed by the subsequently described pattern.

A. System elements

There were the following element types defined by us:

1) Electronic Controller Units (ECUs).
2) Active and passive gateways and connection lines that provide for data propagation.
3) Power supply generators, batteries, power lines, connectors and fuses that provide for power propagation.
4) Sensors and actuators.

B. System Topology

In this pattern, the system topology is defined via two subgraphs, one for the data paths and one for the power lines of the system. Both graphs have the same instances of component types.

C. Failure Types

The automotive pattern defined by us contains the following failure types:

1) Internal Error. This type is valid for ECUs, active and passive gateways and data paths. In case of an internal error, the component does not perform its function. It outputs therefore no data or even wrong data.
2) Silence Error. This type is again valid for ECUs, active and passive gateways and data paths. In case of silence error, the component outputs no data although it should. However, it does not output wrong data.
3) Babbling Error. This type is valid for ECUs and active gateways. In case of babbling error, the component continuously outputs data although it should not. Most or all data are wrong.
4) Short Circuit Error. This type is valid for power supplies, power lines, generator, battery, fuses and data paths.

D. Failure Propagation Rules

Some failure types can propagate through the system. Their propagation characteristics are predefined in the automotive pattern as:

1) The babbling error propagates through data paths and passive gateways. It can not propagate through ECUs and active gateways.
2) The short circuit error propagates through power lines and data paths. It does not propagate through fuses.
3) The internal error and the silence error can not propagate.

E. System Operational Function

In the following, it is assumed that all tasks of the control system can represented by a set of functions. Then, the system operational function is the boolean AND of all these functions. This means that the system function F_{sys} is defined by the functions of its components, according to $F_{sys} = \bigwedge F_i$. Every function F_i in turn relies on the well-functioning of one or several ECUs. Finally, each ECU needs a power supply and may need input data from other ECUs.

VI. Reliability analysis methods

A. Operational Function

The system operational function is a boolean expression, that represents the operational states of its individual components. In this function a "1" denotes an operable state. For the calculation of the operational function, let us consider a system comprising of n elements – $C_j, j \in 1, 2, \ldots, n$.

Let X_j be the boolean variable that indicates whether the element C_j of the system is operational. Then, the system operational function is constructed by the means of the basic rules O, the component types T ($\forall C_j : \exists! T_i \in T : C_j \in T_i$ – C_j of type T_i) and the component connection graphs G for power and data lines. Also the operational functions F_i of the individual components are constructed by G, as well as by the failure types I and the failure propagation rules F. Additionally, it has to be taken into account that there are dependencies between components, and as a consequence, a linear system of equation has to be set-up and solved (the unknown variables in this system are the X_j). Every F_i in turn is determined by its corresponding ECU type T_i. F_i is "1" if there is at least one operable component C_j of type T_i: $F_i = \bigvee_{\forall j : C_j \in T_i} X_j$, $X_j = x_j X_j^{power} X_j^{data}$. In this equation, the used variables have the following meaning:

- x_j – is operable state of element C_j.
- X_j^{power} depends on the power supplies that are connected to C_j by power lines and fuses. The analysis of the power supplies must take into account that short circuits can propagate.
- X_j^{data} is "1" if input is available from all necessary providers. Let C_k be a data provider for component C_j then X_j^{data} contains the component $X_k P_{k,j}$, where $P_{k,j}$ is the operable function of the data paths. In addition, $P_{k,j}$ must consider babbling errors that propagate through data lines and passive gateways.

Subsequently, the equation system of all X_j has to be set-up. In the left part of each equation is a X_j according to $X_j = \mathcal{F}(X_1, ..., X_n)$. However, X_j can also be an independent variable of $\mathcal{F}$ on the right side of the equation.

The system of equations is therefore solved by exclusion-of-unknowns method which means that X_j in $\mathcal{F}$ is replaced by "1" (correctness of this action is proven). An example of this process is given in Section 7.

B. Reliability Function

The reliability function has independent stochastic variables that denote probabilistic failure events of components. For the computation of the reliability function out of the operational function, a boolean-probabilistic method [6][1] is used. This method includes the following two steps:

1) Conversion of the operational function into an equivalent form called "substitution form".
2) Conversion of the substitution form to the reliability function by applying boolean to probabilistic transformation rules according to [6][1].

1) Substitution Form: The substitution form of the operational function F_{sys} is the boolean sum of the orthogonal and repetition-free summands that are concatenated with AND/NOT only. To explain that, let us consider the conjunctive form (which is not normalized) of the operational function $F_{sys} = \bigwedge F_i$, where $F_i = \bigvee_{\forall j: C_j \in T_i} X_j$. This form is converted into the disjunctive normal form by applying the distributivity law. After that, each summand is made orthogonal to every other such that the boolean multiplication of any summand pair yields "0". After that, the orthogonal disjunctive normal form is augmented by the property that every x_k occurs only once in a summand. This is achieved by successively factoring out all x_k in each summand that occurs more than once. Finally, all x_k have to be concatenated with AND and NOT operators only. This is accomplished by applying de Morgan rules. For example, $F_{sys} = \overline{\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3}}$ is a repetition-free form with AND/NOT. This example is also considered to have orthogonal summands because it consists only of one summand.

For the computation of the reliability function, that substitution form of F_{sys} is used because it allows boolean to probabilistic transformations. However, the described procedure to obtain the substitution form is too time-consuming in practice which is why we use a shortcut to obtain it. This shortcut is called Cutting Algorithm [6].

2) Cutting Algorithm: The goal of the Cutting Algorithm is to reduce in every step the number of arguments x_k of the operational function F_{sys} by one. The general form of the cutting algorithm is:

$$\begin{aligned} f(x_1, x_2, ..., x_i, ..., x_n) = \\ = \overline{x_i} f(x_1, x_2, ..., x_{i-1}, 0, x_{i+1}..., x_n) + \\ + x_i f(x_1, x_2, ..., x_{i-1}, 1, x_{i+1}..., x_n), \end{aligned} \quad (1)$$

Every reduction step is applied to that argument which occurs most often in F_{sys}. This argument is factored out according to (1).

For example, $F_{sys} = x_1x_4 + x_2x_5 + x_1x_3x_5$ is a boolean function of 5 independent variables x_1 to x_5 comprising of 3 summands from which x_1 and x_5 are the most frequent ones. The algorithm factors out x_1 in the first step according to: $F_{sys} = \overline{x_1}x_2x_5 + x_1(x_4 + x_3x_5 + x_2x_5)$. Now, x_5 is the most frequent variable, etc. The algorithm stops after one more step in which x_5 have also been factored out, yielding the result: $F_{sys} = \overline{x_1}x_2x_5 + x_1\overline{x_5}x_4 + x_1x_5(x_2 + x_3 + x_4)$.

Finally, AND/OR is replaced by AND and NOT giving: $F_{sys} = \overline{x_1}x_2x_5 + x_1\overline{x_5}x_4 + x_1x_5\overline{\overline{x_2} \cdot \overline{x_3} \cdot \overline{x_4}}$. This is the desired substitution form of F_{sys}, i.e. of the operational function. Each summand is orthogonal to every other, and each x_k occurs only once in every summand.

As one can see, factoring out is achieved at the expense of doubling the number of summands. If N is the number of arguments in F_{sys}, then the algorithm terminates at most after N steps. Therefore, after N steps, the number of summands can be grown by a factor of 2^N. In order to make the cutting algorithm usable, some optimizations have to be introduced additionally.

3) Optimization: Two kinds of optimizations are used to reduce the execution time and memory requirements of the cutting algorithm. These two optimization types are:

1) Simplification of the operational function by boolean algebra rules.
2) Idenfifying independent segments in the system model.

The boolean simplifications have to be employed in every step of the algorithm, while the identifying of independent segments happens only once.

Independent segments are parts of the system which have an own power supply or parts of the system who have regions with local data traffic only. Each segment corresponds to independent part of the operational function. Independent parts can be obtained formally by the following procedure:

1) Find in F_{sys} boolean expressions that occur more than once.
2) Exclude from them those expressions who have the same arguments x_i also in other expressions of F_{sys}.
3) Replace the remaining expressions by substituting them with new functions y_i.
4) Treat y_i as new independent argument in F_{sys}.

For example: $f = (x_1 + x_2x_3 + x_4x_5x_6)(x_1x_2 + x_3 + x_4x_5x_6)$ can be converted into $f = (x_1 + x_2x_3 + y)(x_1x_2 + x_3 + y)$, with $y = x_4x_5x_6$ as new function.

The result of the optimization is that in a system of 100 components, for example, segment splitting reduces time and space complexity by a factor of 5-10, and function simplifying accelerates by an other factor of 3-5, resulting in a cycle time that is 15-50 times quicker than before.

4) Boolean to Probabilistic Transformation: For the subsequent computation of the reliability function, two more phases have to be accomplished. These phases are:

1) Replacement of all variables x_k and boolean operators by stochastic variables p_k and probabilistic operators.
2) Replacement of all p_k with $e^{-\lambda_k t}$, where λ_k is the failure rate of component C_k and t is the independent variable, i.e. argument of the reliability function.

For the replacement of boolean variables and boolean operators, all ANDs in the substitution form of the operational function are replaced by the multiplication of probabilities, all boolean ORs are replaced by the addition of probabilities, and all boolean NOTs are replaced by the $1 - P$ operator, where P is the probability that an event occurs.

These rules are based on the addition and multiplication rules for independent events. Finally, every stochastic variable p_k is replaced by $e^{-\lambda_k t}$ thus obtaining the reliability function $R(t)$. For example, $f = \overline{x_1}x_2 + x_1$ is transformed first into $P = (1 - p_1)p_2 + p_1$, and then into $R(t) = (1 - e^{-\lambda_1 t})e^{-\lambda_2 t} + e^{-\lambda_1 t}$.

Out of the reliability function, the MTTF, the mean fault number MFN in the interval $[T_{min}, T_{max}]$ and the components influence CI_k on the system reliability are automatically computed by the following formulas:

$$MTTF = \int_0^\infty tR(t)dt \quad (2)$$

$$MFN = \frac{(R(T_{min}) - R(T_{max}))(T_{max} - T_{min})}{\int_{T_{min}}^{T_{max}} R(t)dt} \quad (3)$$

$$CI_k(t) = R_k^*(t) - R(t) \quad (4)$$

$R_k^*(t)$ is the reliability function of the system under the assumption that C_k is 100% reliable, i.e. $x_k = 1$. So, for the computation of $CI_k(t)$, $R(t)$ has to be computed twice, one time with C_k in the system, the other time without. Furthermore, all $CI_k(t)$ are calculated at the time point T of MTTF and are subsequently normalized to $[0, 1]$.

VII. Reliability Analyzer Tool

The reliability analyzer tool implements the meta-model with the automotive pattern. All pattern data can be conveniently entered, extended or modified. This is accomplished by means of several menus based on tabs for defining pattern values, as well as by a graph editor, by a component type editor and by other features. All patterns are stored in XML format. Analysis results can be displayed by a chart viewer. The components influence CI_k on the system reliability is color-coded in the viewer in order to inform the users eyes quickly.

Let us consider as an example Figure 1 which contains 4 ECUs (E_1 - E_4), 2 batteries (A and B), one active gateway (G), one passive gateway (P) and 2 fuses (U_1 and U_2).

The elements E_1 and E_2 are of type T_1, E_3 and E_4 are of type T_2. The system operational function is $F_{sys} = F_1$, $F_1 = X_1 + X_2$, where X_i is the operational function of element E_i. The elements E_1 and E_3 have no input data from other

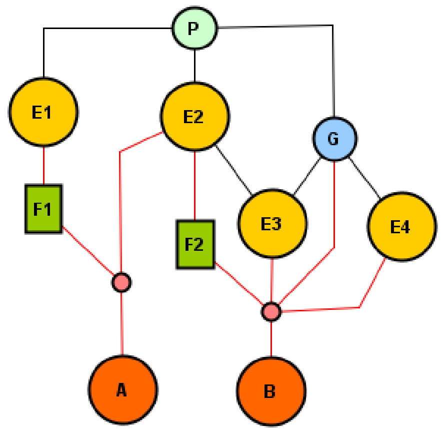


Figure 1. Example of an automotive control system

elements. Element E_2, however, uses data from elements of type T_2, Element E_4 uses data from other elements of type T_1. Under the assumption that all data paths are absolutely reliable, the operational functions of these elements are:

$$X_1 = E_1 \cdot (A \cdot U_1),$$
$$X_2 = E_2 \cdot (A + B \cdot U_2)(X_3 + X_4 \cdot G \cdot P),$$
$$X_3 = E_3 \cdot B,$$
$$X_4 = E_4 \cdot B \cdot (X_1 + X_2) \cdot G \cdot P.$$

This equation system is solved first for the unknowns X_2 and X_3 by applying the mentioned exclusion-of-unknowns method:

$$X_2 = E_2 \cdot (A + U_2) \cdot B \cdot (E_3 + E_4 \cdot P \cdot G),$$
$$X_4 = E_4 \cdot B(E_1 \cdot A \cdot U_1 + E_2(A + U_2)B(E_3 + E_4 \cdot P \cdot G)).$$

From that, the system operational function is achieved which is afterwards converted into substitution form, and then into the reliability function. All other reliability parameters are computed out of the latter. The result of the example is shown in Figure 2. The chosen example was simple but it showed the main points. The analysis of larger systems is accomplished accordingly. Our tool was tested with real-world automotive control systems, that were given to us by one of the leading car manufacturers. It has helped to compare various alternative control system architectures and to select the one with the needed reliability.

The current implementation of the tool can analyze systems of up to several hundreds of components on a PC with AMD Athlon CPU with 2,4 GHz and 256 MB of RAM. We measured the following computation times for systems of different sizes (Table II).

The tool has some overhead compared to commercial tools because of the automated construction of the system

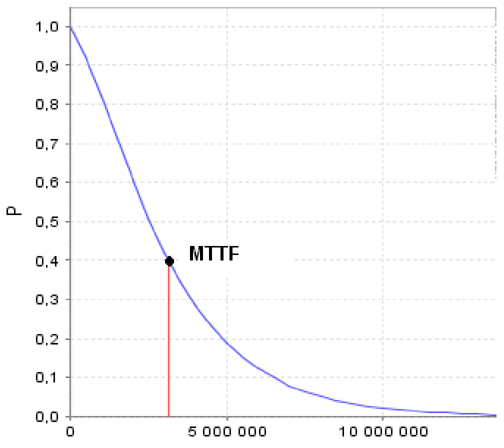


Figure 2. MTTF and the reliability function

Table II
ANALYSIS TIME

Number of components	10	20	30	50	100
Analysis time, sec	1	3	11	57	292

operational function and the automatic conversion of the operational function into the reliability function. However, this overhead can be neglected.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we proposed a new approach and tool for reliability engineering. Our approach is based on a meta-model that allows to many classes class of systems by configuring the model with so-called patterns. The pattern for automotive applications are given by us. It was implemented in our reliability analyzer tool. The tool was used by a large car manufacturer to improve the control system of a automobile. The approach constructs semi-automatically the operational function of the system under test by using information about the system's structure. Our tool computes the operational function, performs an automatic conversion to the reliability function and determines the mean time to failure, the mean fault number and the components' influence on the overall reliability as reliability parameters.

In the future, we will extend the tool to further accelerate the design cycle for reliability engineering by automating the synthesis of redundant systems. Furthermore, we will extend our approach and tool to other application areas outside of automotive control systems.

REFERENCES

[1] G.N. Tcherkesov, *Hardware-software systems reliability.* St. Petersburg: Piter, 2005.

[2] M. Rausand and A. Hoyland, *System Reliability Theory. Models, Statistical Methods and Applications.* Hoboken, NJ: John Wiley & Sons, Inc., 2004.

[3] R. Bosch, *Automotive Electrics and Automotive Electronics, Completely Revised and Extended.* Hoboken, NJ: John Wiley & Sons, Inc., 2007.

[4] T. Denton, *Automobile Electrical and Electronic Systems.* Burlington, MA: Elsevier, 2004.

[5] C. Ericson, *Fault Tree Analysis – a History.* Proceedings of the 17th International Systems Safety Conferencem, 1999, pp. 1-9. http://www.fault-tree.net/papers/ericson-fta-history.pdf. – Retrieved 2011-06-04.

[6] I.F. Ryabinin, *Reliability and safety of structural-complex systems.* St. Petersburg: Polytechnika, 2000.

[7] J.D. Andrews, *An Analysis Strategy for Large Fault Trees.* Proceedings of the 21st International System Safety Conference, August 2003, pp. 375-386.

[8] L.M. Bartlett, *Progression of the binary decision diagram conversion methods.* Proceedings of 21st International System Safety Conference, August 2003, pp. 116-125.

[9] ReliaSoft Corporation. *Comparison of RBD and Fault Tree Simulation.* HotWire Issue 44 (October 2004). http://www.weibull.com/hotwire/issue44/ relbasics44.html. – Retrieved 2011-06-04.

[10] M. McKelvin, G. Eirea and C. Pinello et al., *A Formal Approach to Fault Tree Synthesis for the Analysis of Distributed Fault Tolerant Systems.* Proceedings of the 5th ACM international conference on Embedded software, ACM Press., 2005, pp. 237-246.

[11] Isograph. *Isograph Reliability Analysis Software.* http://www.isograph-software.com. – Retrieved 2011-06-04.

[12] ITEM. *Reliability Software from ITEM.* http://www.itemsoftware.com. – Retrieved 2011-06-04.

[13] ALD. *Advanced Logistics Development.* http://www.aldservice.com. – Retrieved 2011-06-04.

[14] PTC. *The Product Development Company.* http://www.ptc.com. – Retrieved 2011-06-04.

[15] Reliasoft. *Reliability Software.* http://www.reliasoft.com. – Retrieved 2011-06-04.

Methodology and Experience for Designing Safety-Related Systems in IEC 61508

Zhe Chen
College of Computer Science and Technology
Nanjing University of Aeronautics and Astronautics
29 Yudao Street, 210016 Nanjing, China
Email: zhechen@nuaa.edu.cn

Gilles Motet
LAAS-CNRS, INSA
Université de Toulouse
135 Avenue de Rangueil, 31077 Toulouse, France
Email: gilles.motet@insa-toulouse.fr

***Abstract*—The international standard IEC 61508 provides a generic process for electrical, electronic, or programmable electronic (E/E/PE) safety-related systems (SRS) to achieve an acceptable level of functional safety. This paper first proposes the concept of *functional validity* of SRS, based on our observation on two important problems that occur in industrial practice, i.e., the rightness of overall and allocated safety requirements and the lack of technical methodologies for validating SRS. *Functional validity* means whether the safety functions realized by SRS can really prevent accidents and recover the system from hazardous states, provided the expected safety integrity level is reached. Then this paper proposes a generic technical methodology to achieve the functional validity of SRS, and summarizes industrial experiences in designing functionally valid SRS. A concrete example is used to illustrate the proposed methodology.**

***Keywords*-safety-related system; IEC 61508; functional validity; verification; model checking; formal method; SPIN**

I. Safety-Related Systems and Functional Validity

The international standard IEC 61508 [1][2][3] provides a generic process for electrical, electronic, or programmable electronic (E/E/PE) safety-related systems to achieve an acceptable level of functional safety. The principles of IEC 61508 have been recognized as fundamental to modern safety management [2], thus have gained a widespread acceptance and been used in practice in many countries and industry sectors [4].

Like other safety standards (e.g., DO-178B [5]), IEC 61508 gives recommendations on best practices such as planning, documentation, verification, safety assessment, rather than concrete technical solutions. Thus, it is a generic standard for the safety management throughout the life-cycle, rather than a system development standard. More sector-specific and application-specific standards can be derived based on the standard, such as IEC 61511 for process industry [6], IEC 62061 for machinery industry, IEC 61513 for nuclear plants, EN 50126 for European railway, and ISO 26262 for automotive safety.

As shown in Fig. 1, the first premise of the standard is that there is an equipment intended to provide a function, and a system which controls it. The equipment is called an *Equipment Under Control* (EUC). The *Control System* (CS) may be integrated with or remote from the EUC. A fundamental tenet of the standard is that, even if the EUC and the CS are reliable, they are not necessarily safe. This is true for numerous systems implementing hazardous specification. They may pose risks of misdirected energy which result in accidents.

The second premise is that *Safety-Related Systems* (SRS) are provided to implement the expected *safety requirements*, which are specified to reduce the risks and achieve functional safety for the EUC. The SRS may be placed within or separated from the CS. In principle, their separation is preferred.

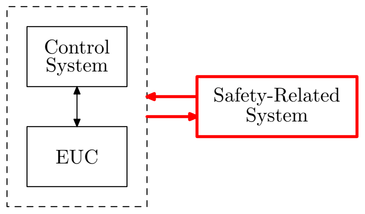


Figure 1. The Architecture of Systems with SRS

An SRS may comprise subsystems such as sensors, logic controllers, communication channels connected to the CS, and actuators. Usually, an SRS may receive two types of input: the values of safety-related variables monitored by its sensors, and the messages sent by the CS. Then the SRS executes computation to decide whether the system is in a safe state. According to the result, it may actualize safety functions through two types of output: directly sending commands to its actuators, or sending a message to the CS.

Let us consider two important problems that occur in industrial practice.

The first one questions the *rightness* of overall and allocated safety requirements. According to IEC 61508, safety requirements consist of two parts, *safety functions* and associated *safety integrity levels* (SIL). The two elements are of the same importance in practice, because the safety functions determine the maximum theoretically possible risk reduction [7]. However, the standard focuses more on the realization of integrity requirements rather than function requirements. As a result, the standard indicates only that the product is of a given reliable integrity, but not that it implements the *right* safety requirements.

Second, the standard does not prescribe exactly *how* the verification of safety functions of an SRS could technically be done. On one hand, the standard calls for avoiding faults in the design phase, since the ALARP principle (As Low As Reasonably Practicable) is adopted for determining tolerability of risk. Indeed, *systematic faults* are often introduced during the specification and design phases. Unlike random hardware failures, the likelihood of systematic failures cannot easily be estimated. On the other hand, the standard only recommends a *process* and a list of techniques and measures during the design phase to avoid the introduction of systematic faults, such as computer-aided design, formal methods (e.g., temporal logic), assertion programming, recovery (see parts 2, 3 of [1]). The detailed use of these techniques is left to the designer.

As a result, the problem of functional validity arises. ***Functional validity* means whether the *safety functions* realized by SRS can really prevent accidents and recover the system from hazardous states**, provided the expected safety integrity level is reached. People are searching for a generic technical methodology to achieve functional validity.

In fact, with the introduction of SRS, it becomes much harder to ensure the safety of the overall system, due to complex interactions and the resulting huge state space. Unlike the case of a single CS, human is no longer capable to analyze manually and clearly the behaviors of the overall system. Therefore, we shall expect a computer-aided method.

This paper proposes such a generic technical methodology (or a framework) for designing *functionally valid* SRS. To the best of our knowledge, we are the first to consider the technical solution to functional validity of SRS in the literature. We focus on the systems that operate on demand (i.e., discrete event). The methodology is based on computer-aided design in association with automated verification tools (e.g., SPIN). In Section II, we present a concrete example illustrating the application of the proposed methodology, which is formally discussed in Section III. We discuss related work in Section IV, then conclude in Section V.

II. Example: Chemical Reactor

As an example, consider an accident occurred in a batch chemical reactor in England [8][9]. Figure 2 shows the design of the system. The computer, which served as a control system, controlled the flow of catalyst into the reactor and the flow of water for cooling off the reaction, by manipulating the valves. Additionally, the computer received sensor inputs indicating the status of the system. The designers were told that if an abnormal signal occurred in the plant, they were to leave all controlled variables as they were and to sound an alarm.

On one occasion, the control system received an abnormal signal indicating a low oil level in a gearbox, and reacted as the functional requirements specified, that is, sounded an alarm and maintained all the variables with their present condition. Unfortunately, a catalyst had just been added into the reactor, but the control system had not yet opened the flow of cooling water. As a result, the reactor overheated, the relief valve lifted and the contents of the reactor were discharged into the atmosphere.

We believe that an SRS could be used to avoid the accident. Figure 3 shows the role of the SRS in the overall system. It receives signals from additional sensors, and communicates with the CS. The key issue is how to specify and design the SRS and prove its *functional validity*, i.e., the SRS is really efficient in the hazardous context. We illustrate a methodology based on computer-aided design in association with the SPIN model checker.

The SPIN (Simple Promela INterpreter) model checker is an automated tool for verifying the correctness of asynchronous distributed software models [10][11]. Systems and correctness properties to be verified are both described in Promela (Process Meta Language). SPIN also supports Linear Temporal Logic (LTL) formulas, which are converted into never claims written in Promela for verification. Besides a checker, SPIN can also operate as a simulator by executing the model following one possible execution trace.

We will illustrate two main steps of the methodology: modeling the CS, and modeling the SRS.

Modeling Control Systems. The first step is to analyze the behaviors of the CS by modeling it using Promela. Listing 1 shows the model. The CS scans the status of the reactor, and then manipulates the valves according to the status.

Listing 1. The Promela Program for Reactor Control System

```
#define sa ((abnorm && !cata)||(abnorm && cata && water))
#define fcon status==nocata || status==encata
mtype = {abnormal, nocata, encata, nowater, enwater};
mtype status = nocata; /* status of the reactor */
bool cata   = false;   /* whether catalyst flow is open */
bool water  = false;   /* whether water flow is open */
bool abnorm = false; /* whether abnormal signal occured */

/* random simulation of scanning the status */
inline scan() {
if
  :: true -> status = abnormal;
  :: cata  == false -> status = nocata;
  :: cata  == true  -> status = encata;
  :: water == false -> status = nowater;
  :: water == true  -> status = enwater;
fi;
}

/* possible actions of the system */
inline opencata()  {cata =true; printf("open cata -> "); }
inline closecata() {cata =false;printf("close cata -> ");}
inline openwater() {water=true; printf("open water -> ");}
inline closewater(){water=false;printf("close water ->");}
inline alarm() { abnorm = true; printf("alarm -> "); }
inline ending(){ printf("ending -> "); }

active proctype controlsystem(){
/*Initial actions omitted*/
```

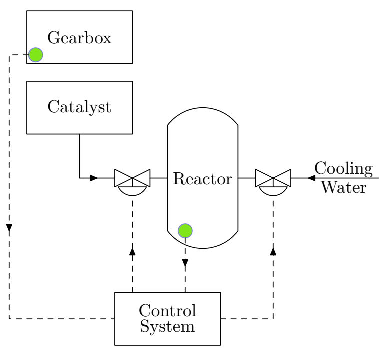


Figure 2. Reactor Control System

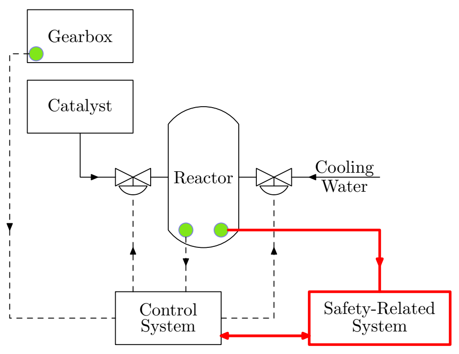


Figure 3. Reactor Control System with SRS

```
30 do
31 :: scan();
32    if
33    :: status == abnormal -> alarm(); goto END;
34    :: else -> if
35       :: status==nocata  && cata ==false->opencata();
36       :: status==encata  && cata ==true ->closecata();
37       :: status==nowater && water==false->openwater();
38       :: status==enwater && water==true ->closewater();
39       :: else -> skip;
40       fi;
41    fi;
42 od;
43 END: ending();
44 assert(sa);
45 }
```

Lines 1-2 define macros for specifying correctness properties.

Lines 3-7 define the variables. Note that `status` denotes the detection of events (e.g., abnormal signal, no catalyst or enough catalyst in the reactor), and controls the flow of the computation, while the other three variables save the state information. Note that abnormal signal may be caused by several events. For example, low oil level is one of these events.

Lines 10-18 simulate the input events to the CS. In the verification mode, SPIN can only treat closed system without users' input. Therefore, the status of reactor is generated in a random manner, i.e., the `if` statement (lines 11-17) chooses randomly one of the alternative statements whose condition holds. Note that this matches exactly what happens in practice, since the status of the reactor is determined nondeterministically by the reaction and the environment, not the user.

Lines 21-26 define the primitive actions of the CS. To obtain an efficient model for verification, the real codes for manipulating physic equipments are replaced by `printf` statements, which could be used for observing the execution traces of the CS.

Lines 28-45 specify the model of the CS. The non-critical codes (irrelevant to the concerned property, e.g., the code for initiation) are omitted or simplified to produce an efficient model. In line 44, we check whether the system satisfies the safety assertion `sa` (cf. line 1), which means when an abnormal signal occurs, either the flow of catalyst must be closed, or the flow of water must be also open if the flow of catalyst is open. The violation of this assertion may result in the mentioned accident.

In order to see whether the model describes exactly the behaviors of the CS, we run the model through the simulation mode of SPIN. One of the outputs is as follows.

```
$ spin reactor.pml       # Linux command
open cata -> alarm -> ending ->
spin: line  44 "reactor.pml", Error: assertion violated
```

The execution trace shows that the alarm is sounded after opening the flow of catalyst, then the safety assertion is violated. This trace characterizes exactly what happened in the accident. It is worth noting that, due to the uncertainty of the value of `status` (cf. lines 11-17), the assertion may be satisfied in another run. This is the reason why the plant had functioned well before the accident. As a result, in the simulation mode, it is very possible that an existing fault will not be detected after numerous runs.

To systematically check all the state space, we use the verification mode, which needs correctness properties specifying hazard-free situation.

One possibility is to use assertions, such as what we did in line 44. Note that an assertion can only check the state at a single position in the execution (i.e., line 44), not the remainder positions. The SPIN checks the assertion in the verification mode as follows.

```
$ spin -a reactor.pml
$ gcc pan.c
$ ./a.out
State-vector 16 byte, depth reached 12, errors: 1
        21 states, stored
         0 states, matched
        21 transitions (= stored+matched)
```

After examing 21 states, SPIN detected the unsafe state. We can simulate the counterexample by tracing simulation.

```
$ spin -t reactor.pml
open cata -> alarm -> ending ->
spin: line  44 "reactor.pml", Error: assertion violated
```

The result shows that the unsafe state can really be reached.

Another alternative for specifying correctness property is LTL formula. For this example, the formula is $\phi \overset{\text{def}}{=} \mathbf{G}(cata \rightarrow \mathbf{F}water)$, where $\mathbf{G}$ means *globally* and $\mathbf{F}$ means *future* [12]. It means whenever the flow of catalyst is opened, the system will have the chance to open the flow of water in the future. Of course, this is based on a reasonable assumption that the status will eventually be `nowater` when there is not enough water in the reactor (i.e., there is a reliable sensor). This assumption is expressed by the fairness condition $\mathbf{GF}!fcon$. The SPIN checks the LTL formula in the verification mode as follows (`[]` denotes $\mathbf{G}$, `<>` denotes $\mathbf{F}$).

```
$ spin -a -f '![](cata -> <>water)&&[]<>!fcon' reactor.pml
$ gcc pan.c
$ ./a.out -A -a #disable assertion, check stutter-
    invariant
State-vector 20 byte, depth reached 29, errors: 1
       23 states, stored
        1 states, matched
       24 transitions (= stored+matched)
```

After examing 23 states, SPIN detected the unsafe state. We can simulate the counterexample by tracing simulation.

```
$ spin -t reactor.pml
open cata -> alarm -> ending ->
spin: trail ends after 30 steps
```

Since we have already known that the CS may cause a hazardous state, we must make sure that the specified properties can detect the hazardous state, or else the properties are not right. Then we introduce an SRS to control potential hazardous executions.

Modeling Safety-Related Systems. The second step is to construct the model of the SRS. We reuse the established model of the CS, and derive an accurate model describing the behaviors of the SRS. Listing 2 shows the models. The model of the SRS receives messages sent by the CS, computes whether the system is safe, and sends a message back to the CS.

Listing 2. Reactor Control System with SRS in Promela

```
#define sa ((abnorm && !cata)||(abnorm && cata && water))
#define fcon status==nocata || status==encata
mtype = {abnormal, nocata, encata, nowater, enwater};
mtype status = nocata; /* status of the reactor */
bool cata   = false;  /* whether catalyst flow is open*/
bool water  = false;  /* whether water flow is open */
bool abnorm = false; /* whether abnormal signal occured*/

/*define safety-related variables, messages structure*/
typedef SRV { bool _water; }
typedef MSG { bool _water; bool _abnorm; bool _res; }
chan ch = [0] of { MSG };       /* message channel */

/*random simulation of scanning the status*/
inline scan() {
if
  :: true -> status = abnormal;
  :: cata  == false -> status = nocata;
  :: cata  == true  -> status = encata;
  :: water == false -> status = nowater;
  :: water == true  -> status = enwater;
fi;
}

/* possible actions of the system */
inline opencata() {cata=true; printf("open cata -> ");}
inline closecata(){cata=false;printf("close cata-> ");}
inline openwater()
  { water = true;  cs_2srs(); printf("open water -> "); }
inline closewater()
  { water = false; cs_2srs(); printf("close water -> ");}
inline alarm(){abnorm=true;cs_2srs();printf("alarm ->");}
inline ending() { printf("ending -> "); }
inline cs_epro()
    { printf("error processing (");
      printf("water opened");
      water = true;
      printf(") -> "); }

/* communication between CS and SRS */
inline cs_2srs() {
if
  :: abnorm == true  -> msg._abnorm = true;
  :: abnorm == false -> msg._abnorm = false;
fi;
if
  :: water == true  -> msg._water = true;
  :: water == false -> msg._water = false;
fi;
msg._res = true;
ch ! msg;
ch ? msg;
if
  :: msg._res == false -> cs_epro();
  :: else -> skip;
fi;
}

active proctype controlsystem(){
/*Initial actions omitted*/
MSG msg;
do
:: scan();
   if
   :: status == abnormal -> alarm(); goto END;
   :: else -> if
      :: status==nocata  && cata ==false ->opencata();
      :: status==encata  && cata ==true  ->closecata();
      :: status==nowater && water==false ->openwater();
      :: status==enwater && water==true  ->closewater();
      :: else -> skip;
      fi;
   fi;
od;
END: ending();
assert(sa);
}

/****** The Safety-related System ******/
/*random simulation of scanning the values of variables*/
inline srs_scan() {
if
  :: srv._water = true;
  :: srv._water = false;
fi;
}

/* compute whether the system is safe */
inline srs_compute() {
if
  :: msg._abnorm == true && msg._water == false ->
                     msg._res = false;
  :: msg._abnorm == true && srv._water == false ->
                     msg._res = false;
  :: else -> msg._res = true;
fi
}

active proctype srs(){
/*Initial actions omitted*/
MSG msg;
SRV srv;
do
  :: true ->
105 endsrs: ch ? msg;
106         srs_scan();
107         srs_compute();
108         ch ! msg;
109 od;
110 }
```

Lines 10-11 define the inputs to the SRS. The type SRV defines a set of safety-related variables, whose values could be obtained from additional sensors outside the CS and managed by the SRS. For this example, SRV monitors only whether the water flow is open. The type MSG defines the structure of the messages communicated between the CS and the SRS. Line 12 defines a rendezvous channel for the message.

In lines 28-32, for each primitive action that modifies the values of the variables monitored by the SRS, the communication between the CS and the SRS is inserted. The communication module (lines 41-57) reads information needed, and sends a message to the SRS, then receives a response. The module analyzes the result in the returned message. If it indicates an unsafe state, the system calls the error processing module to recover from the hazardous state.

The error processing module (lines 34-38) uses the information in the returned message to decide the actions. The process could change the values of certain variables (e.g., line 37) after manipulating physic equipments (e.g., in line 36, `printf` statement abstracts the action of opening the valve). Note that more information could be contained in the returned message in more complex systems (i.e., not only a boolean result), in order to provide sufficient information for the error processing module to analyze the current state.

Lines 99-110 define the model of the SRS. It waits for the message sent by the CS, then scans the values of the safety-related variables (lines 81-86), and computes whether the system is safe using the message and the safety-related variables (lines 89-97). Finally the SRS sends a response to the CS. Note that the computation in `srs_scan` and `srs_compute` could be different in various systems. Embedded C code could be used to implement more complex functions. Anyway, we use the same methodology and framework.

In order to see whether the model characterizes exactly the behaviors of the SRS, we run the model through the simulation mode of SPIN. One of the outputs is as follows.

```
open cata -> error processing ( water opened ) -> alarm ->
       ending ->
```

The execution trace shows that the safety assertion is not violated, which is exactly what we expect to avoid the mentioned accident.

Then we check the assertion in the verification mode, and no error is found.

```
State-vector 40 byte, depth reached 208, errors: 0
      409 states, stored
       57 states, matched
      466 transitions (= stored+matched)
```

We may also check the LTL formula in the verification mode.

```
State-vector 44 byte, depth reached 401, errors: 0
      707 states, stored (979 visited)
      658 states, matched
     1637 transitions (= visited+matched)
```

Zero errors mean that the assertion holds and the LTL property always holds in the execution, i.e., no unsafe state could be reached. Therefore, we conclude that the established model of the SRS can successfully avoid the accident, i.e., a "functionally valid" SRS. Note that, if we use another computation in `srs_compute`, the SRS may be not functionally valid (e.g., always let `msg._res` be `true`). That is the reason why we need such a methodology to ensure functional validity.

It is worth noting that the combination of the CS and the SRS has 707 states and 1637 transitions (more complex the overall system, larger the state space). Human is not able to analyze the correctness of such a complicated system. As a result, computer-aided design may be the only choice for developing functionally valid SRS.

III. Methodology for Designing Functionally Valid SRS

In this section, we propose the generic methodology for designing functionally valid safety-related systems. As we mentioned, the state space of the overall system including the CS and the SRS is much larger than the one of a single CS or a single SRS. As a result, manual analysis and design of the SRS are never trustworthy and always error-prone. This methodology uses computer-aided design in association with automated verification tools, thus can improve our confidence on the functional validity of the design of SRS.

We try to list exhaustively all the key issues that we know in the designing process, in order to guide the practice. Due to the wide application of the standard and SRS, the reader may encounter different situation in various projects and industry sectors. Therefore, some necessary adaptations should be made in detail for a specific project.

Generally, there are three steps for developing a functionally valid SRS: modeling the CS, modeling the SRS and implementing the SRS. This paper focuses on the first two steps (i.e., the design process) which lead to a functionally valid design of the SRS, although it is worth noting that faults may be also introduced in the implementation process.

Modeling Control Systems. The first step is to construct the model of the CS. The results of this step are a Promela program for the CS and the correctness properties. We list some key issues as follows.

(1) The first task is to derive *accurate* and *efficient* abstraction of the behaviors of the CS. The criteria for judging the quality of the model are mainly *accuracy* and *efficiency*. Accuracy means that the model behaves exactly like the real CS, while efficiency means that the model is smart enough, e.g., the program should use as few variables, statements and memory as possible. Some typical issues are the following ones:

(a) Define variables, both for computation and checking. Some variables are used for computation, that is, implementing control flow, behaviors and semantics of the system. Some variables are used for representing the state of the system, so they do not contribute to the functionality of the system. They are only used in the correctness properties to check the property of the indicated state.

It is worth noting that the size of variables must be carefully defined. The principle is "as small as possible". The model checker will produce a huge state space, of which each state contains all the defined variables. As a result, the restriction from `int` to `byte` will save considerable memory when checking.

(b) Simulate the random inputs to the CS. In the verification mode, the Promela model is a closed system. In other words, it cannot receive users' inputs. As a result, we must generate the inputs in the program.

The best way is to generate the inputs randomly, i.e., nondeterministically choose one member from the set of all possible values. The advantage of this method is that it can simulate the uncertainty of the inputs, that is, we do not know when a specific input will occur. For example, consider the sensors' signal which is determined by the environment.

(c) Simplify reasonably the computation of the CS. Due to the size and complexity of the real system, an automated model checker may not even be able to produce the result in an acceptable instant. Obviously, the huge size contributes to a huge number of states, and the complexity contributes to a huge number of transitions. As a result, the size of the state space may be much larger than the memory, then the model checker will fail to accomplish the verification.

One solution is to provide a more coarse abstraction of the CS. That is, we omit some non-critical computations, e.g., the initiation of the system. Furthermore, some manipulations of physical equipments can be expressed with only a `printf` statement, which does not increase the size of the state space.

Another solution is to decompose the system into several parts, and check these parts one by one. When checking one single part, we make the assumption that the other parts are correct. It is worth noting that the decomposition is relevant to the properties to check. That is, we must put all the functions relevant to a certain property into the same part, when checking the property.

(d) Use embedded C codes if necessary. Due to the complexity of embedded C codes and the lack of syntax checking, they are mainly used for automated model extraction in SPIN. However, the strong expressive power of C code is anyway a tempting feature. Thus the recommendation is made only "if necessary".

(e) Simplify or eliminate the codes for controlling equipments. Usually we assume that the codes for implementing primitive manipulations are correct, e.g., opening the flow of water.

(2) The second task is to derive correctness properties, i.e., assertions and LTL formulas.

(a) Assertions are used to check the property at a specific position in the program. Obviously, the expressive power is limited. Thus, if we want to check a property over all the state space, we must use LTL formulas.

(b) LTL formulas are used to check all the states in the system. Obviously, LTL formulas are more powerful. However, it is also worth noting that LTL formulas can considerably largen the state space. Thus we suggest to use them only when assertions are not able to express a property.

(3) Some Experiences.

(a) Check the exactitude of the established model, by using simulation mode. The `printf` statement can be used to output information about the state of the CS. Simulating the model for sufficient many times can show whether the model works as expected. Note that the criteria "sufficient many times" is due to the uncertainty of the inputs.

(b) Check the exactitude of the correctness properties, by using verification mode. If we aim at creating or modifying an SRS for identified risks, we must make sure that the correctness properties can detect the error caused by the identified risks.

Modeling Safety-Related Systems. The second task is to construct the model of the SRS. The results of this step are a Promela program for the SRS and the codes for communication and error processing in the modified model of the CS. We list some key issues as follows.

(1) The premise is that we have the model of the CS and reuse it, e.g., the results of the last step.

(2) The first task is to derive *accurate* and *efficient* abstraction of the behaviors of the SRS. Some typical issues are the following ones:

(a) Define the scope of input and output of the SRS. There are two types of input: the message sent by the CS and the values of safety-related variables sent by the sensors. The SRS may use only one of the two types, or both of them. There are two types of output: the message sent to the CS and the direct manipulation of the equipment. Also, the SRS may use only one of the two types, or both of them.

(b) Define safety-related variables. A *safety-related variable*, which saves a value sent by sensors, is used to collect additional information which is beyond the scope of the CS, or to collect a specific piece of information in the scope of the CS for increasing the reliability of the information. Note that more safety-related variables means higher cost of implementation.

(c) Choose the type of communication between the CS and the SRS. The SPIN model checker supports two types of communications: rendezvous and buffered communication.

In order to process the demand from the CS to the SRS as soon as possible, and also provide information from the SRS to the CS to decide the next action, we usually choose the rendezvous communication.

(d) Define the structure of message. This depends on the information needed by the CS and the SRS. The values of the variables monitored by the SRS should be sent from the CS to the SRS. The message should also contain all the necessary information needed by the CS to deal with risks. In the simplest case, it is a Boolean result, indicating the system is safe or not. However, generally, the CS need more information to determine why the system is not safe, then it can activate corresponding error processing functions.

(e) Define message channels. Usually, one channel is enough for the communication between the CS and the SRS.

(f) Simulate the scanning of the values of safety-related variables. It is similar to the case "simulate the random inputs to the CS". The random simulation express exactly the fact that the values are nondeterministically decided by the environment.

(g) Simplify reasonably the computation of the SRS. (Similar to the case of the CS.)

(h) Use embedded C code if necessary. (Similar to the case of the CS.)

(3) The second task is to define the position for the communication between the CS and the SRS. Generally, the usual location is between the assignment of key variables monitored by the SRS and the manipulation of physical equipment. Therefore, the SRS can check whether the system will be safe if the next manipulation is executed. If no, the SRS can send a message to activate the error processing functions.

(4) The third task is to define the function of error processing. This step is different for different projects, because it is based on the requirements of a specific system. In fact, the correctness of error processing also plays an important role in the correctness of the overall system and the functional validity of the SRS.

(5) Some Experiences.

(a) Check the exactitude of the established model, by using simulation mode. (Similar to the case of the CS.)

(b) Check the exactitude of the correctness properties, by using verification mode. We must make sure that the overall system (including the CS and the SRS) is safe, i.e., satisfies the specified correctness properties. If we aim at creating or modifying an SRS for identified risks, we must make sure that the overall system can avoid the previously detected errors, since the SRS component and additional error processing functions have been added.

If errors are detected, we must check the design of the SRS, and also the exactitude of the correctness properties (because they may specify a semantics different from what we expect).

Implement the SRS. We implement the SRS using the established model and computation. Note that faults may also occur at this step. Since numerous guidelines exist in industrial sectors to handle this issue, the discussion on reliable implementation is beyond the scope of this paper.

IV. Related Work

The use of formal methods and model checking for ensuring safety or proving the absence of certain hazards is not new in the literature.

For example, Eriksson [13] showed how formal verification can be used in a retrospective safety case through an example taken from railway signalling. This approach can be used to demonstrate the safety of the safety-critical systems in operation which were developed according to old practises that would be regarded as unacceptable today. In this application of formal methods, several particular problems were discussed, such as uncertainty about the original requirements and the required safety level of the various system functions.

The ForMoSA project [14] developed an integrated approach for safety analysis of critical embedded systems. The approach brings together the best of engineering practice, formal methods and mathematics: traditional safety analysis, temporal logics and verification, and statistics and optimization. These three orthogonal techniques cover three different aspects of safety: fault tolerance, functional correctness and quantitative analysis.

However, these works in the literature only focus on how to ensuring safety, rather than the functional validity of SRS in IEC 61508. Note that one key idea of the standard is the separation of the CS and the SRS. Thus ensuring the functional validity of SRS is different from the safety issues of a single control system which are discussed in the literature. Therefore, we are the first to consider the technical solution to functional validity of SRS in the literature.

Furthermore, our related papers [15][16] proposed the theory of formal control systems, based on the traditional automata theory. A formal control system consists of two automata. The controlled automaton is monitored by the controlling automaton to satisfy the given specification. The theory can be considered as the theoretical foundation of the SRS. The interested reader is referred to [15][16], since the theoretical aspect is beyond the scope of this industrial practice-oriented paper.

V. Conclusion

This paper first proposed the concept of *functional validity* of SRS, based on our observation on two important problems that occur in industrial practice, i.e., the rightness of overall and allocated safety requirements and the lack of technical methodologies for validating SRS.

Then this paper proposed a generic technical methodology (or a framework) which is based on computer-aided design in association with automated verification tools, for designing *functionally valid* SRS. To the best of our knowledge, we are the first to consider the technical solution to functional validity of SRS in the literature.

The case study provided a customized demonstration of applying the methodology. The same methodology may be used for other situations by taking into account only some necessary adaptations, since it is a generic modeling approach for designing functionally valid SRS.

There are also some limitations of the proposed approach. First, the approach cannot completely eliminate design defects. This is due to the fact that it is not possible to identify all the safety requirements and correctness properties before verification. As a solution, we may use some sophisticated methodologies discussed in the literature to analyze hazards and their probabilities, e.g. fault tree analysis. Second, the modeling process is generally manual. We may consider how to automate the modeling process to decrease the time required for validation. Third, like other model checking techniques, the state explosion problem is also a challenge for scaling up the approach. One possible solution is to improve the model checking algorithm to reduce the state space [17]. Another possible solution is to develop optimized and customized model checker for verifying SRS.

Acknowledgment

This work was supported by the China Postdoctoral Science Foundation and the National Natural Science Foundation of China (60703033 and 61033002). The authors are grateful to the anonymous referees for their detailed comments and helpful suggestions.

References

[1] IEC, *IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*. International Electrotechnical Commission, 1999.

[2] F. Redmill, "IEC 61508 - principles and use in the management of safety," *Computing & Control Engineering Journal*, vol. 9, no. 5, pp. 205–213, 1998.

[3] S. Brown, "Overview of IEC 61508 - design of electrical/-electronic/programmable electronic safety-related systems," *Computing & Control Engineering Journal*, vol. 11, no. 1, pp. 6–12, 2000.

[4] R. Faller, "Project experience with IEC 61508 and its consequences," *Safety Science*, vol. 42, no. 5, pp. 405–422, 2004.

[5] D. S. Herrmann, *Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors*. IEEE Computer Society, 2000.

[6] H. Gall, "Functional safety IEC 61508 / IEC 61511 the impact to certification and the user," in *Proceedings of the 6th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2008)*. IEEE, 2008, pp. 1027–1031.

[7] D. Fowler and P. Bennett, "IEC 61508 - a suitable bases for the certification of safety-critical transport-infrastructure systems??" in *Proceedings of the 19th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2000)*, ser. Lecture Notes in Computer Science, F. Koornneef and M. van der Meulen, Eds., vol. 1943. Springer, 2000, pp. 250–263.

[8] T. Kletz, "Human problems with computer control," *Plant/-Operations Progress*, vol. 1, no. 4, 1982.

[9] N. Leveson, *Safeware: System Safety and Computers*. Addison-Wesley, Reading, MA, 1995.

[10] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[11] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[12] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 2000.

[13] L. henrik Eriksson, "Using formal methods in a retrospective safety case," in *Proceedings of the 23rd International Conference on Computer Safety, Reliability and Security (SAFECOMP 2004)*, ser. Lecture Notes in Computer Science, vol. 3219. Springer, 2004, pp. 31–44.

[14] F. Ortmeier, A. Thums, G. Schellhorn, and W. Reif, "Combining formal methods and safety analysis - the formosa approach," in *Integration of Software Specification Techniques for Applications in Engineering*, ser. Lecture Notes in Computer Science, vol. 3147. Springer, 2004, pp. 474–493.

[15] Z. Chen and G. Motet, "Towards better support for the evolution of safety requirements via the model monitoring approach," in *Proceedings of the 32nd International Conference on Software Engineering (ICSE 2010)*. ACM, 2010, pp. 219–222.

[16] Z. Chen and G. Motet, "System safety requirements as control structures," in *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009)*. IEEE Computer Society, 2009, pp. 324–331.

[17] Z. Chen and G. Motet, "Nevertrace claims for model checking," in *Proceedings of the 17th International SPIN Workshop on Model Checking of Software (SPIN 2010)*, ser. Lecture Notes in Computer Science, vol. 6349. Springer, 2010, pp. 162–179.

Supporting Synthetic Data-Driven Diagnosis through Automated Fault-Injection

Patrick E. Lanigan, Priya Narasimhan
Carnegie Mellon University
Electrical & Computer Engineering
planigan@ece.cmu.edu, priya@cs.cmu.edu

Thomas E. Fuhrman
General Motors Research & Development
Electrical & Controls Integration Lab
thomas.e.fuhrman@gm.com

Abstract

Given the lack of empirical data available from automotive serial-communication networks, an automated fault-injection environment can be used to create synthetic datasets for training and testing data-driven diagnosis algorithms. We use commercial fault-injection hardware with custom software to implement such an environment. A small pilot study using injected physical-layer faults shows promise in producing identifiable error-patterns.

1 Introduction

The automotive industry has become steadily more reliant on software-intensive distributed systems to implement advanced vehicle features. In fact, it has been estimated [7] that "up to 90% of all innovations are driven by electronics and software". It is further estimated [7] that 50-70% of an ECU's development costs come from software, with some vehicles having up to 70 ECUs. Overall, electronics and software can account for up to 40% of a vehicle's cost [7]. When problems in software-based systems are uncovered after a vehicle has gone to production, recall costs can rival development costs. For example, 2004 saw the recall of 680,000 Mercedes-Benz E-Class vehicles due to issues with the electronic brake-by-wire system [23].

A growing trend is toward features that assist the driver in maintaining safe control of the vehicle under a variety of conditions. Previously, such assistance has been provided *passively* in the form of information or warnings. These features are now being given increasing amounts of authority to control the vehicle's motion by *actively* supplementing the driver's inputs. The long term trend is towards fully autonomous operation [16, 21].

Because these systems are critical to ensuring the safe operation of the vehicle, they must be designed to tolerate faults and provide high levels of dependability. Typically, a systematic safety analysis is conducted during the design phase, to evaluate both the severity and likelihood of the consequences of possible faults. Formal verification methods are used to analyze system dependability. Fault-injection can play a complementary role in this analysis by providing an empirical way to study the system's dependability in the presence of faults and to analyze the system's fault-handling capabilities with respect to a particular fault model. This can aid in fault-removal and fault-forecasting [2]. The upcoming ISO 26262 standard for functional safety in automotive electronics highly recommends that fault-injection be included as part of the dependability analysis of critical systems [10].

Despite extensive design processes, emergent behavior will still appear at runtime in dependable automotive systems. Such behavior occurs due to unforeseen interactions and complexity between independently designed components. These interactions are not readily apparent to the system designers, and might not be captured by system models. Therefore, diagnostic approaches that rely solely on system models are unlikely to provide a satisfactory diagnosis when presented with emergent behavior. A data-driven diagnostic approach that analyzes system metrics as well as system models has the potential to provide a more accurate diagnostic output [12].

In order to support the development of data-driven diagnostic approaches, we built an automated fault-injection environment for FlexRay [6]. This environment combines off-the-shelf fault-injection hardware with custom software to allow a large number of *highly repeatable* experiments to be coordinated from a centralized host. It also enables data-logging from *each* node in the cluster, as opposed to relying on a single monitoring point.

We performed a pilot study to validate this fault-injection environment and to determine whether it is feasible to distinguish faults based on their manifestations. The results of this study show that faults do no necessarily manifest symmetrically in a linear bus topology. Furthermore, those manifestations produce identifiable error patterns.

2 Synthetic Data-Driven Diagnosis

The analytic techniques used for data-driven diagnosis vary [3, 4, 8, 11], but commonly require a large dataset for algorithm training and testing. This data typically comes in the form of metrics that are derived from various instrumentation points. These instrumentation points exist at the system-level as well as the component-level. Examples of potential instrumentation points are the status indicators exposed by the FlexRay Controller-Host Interface (CHI) (see section 3.2), hooks inserted into software components, and Operating System (OS) metrics such as context-switch rates. The data can then be analyzed to infer a correlation between the metrics and the system health. Ideally, this correlation leads to some actionable diagnostic output.

Usually, such metrics are gathered from deployed systems. This is problematic in automotive systems, because real-world failure data is scarce. The most advanced dependable automotive systems exist only as research prototypes, and therefore have not seen wide enough deployment to generate useful failure metrics. For the few systems that have been deployed, Original Equipment Manufacturers (OEMs) are understandably reluctant to release failure data for public scrutiny.

Therefore, we propose leveraging fault-injection to build a synthetic data-driven approach. Fault-injection is already recommended to be used in the dependability analysis of critical systems [10], so the impact on existing development processes should be minimal. Even so, there are many research challenges involved in developing such an approach. We discuss these research challenges elsewhere [12], but the entire endeavor rests on a few key propositions.

Proposition 1 *The errors induced by faults form identifiable patterns.*

Proposition 2 *The error patterns corresponding to faults are evident in, and can be derived from, system metrics.*

Proposition 3 *The error patterns derived from system metrics allow faults to be distinguished by type, persistence, etc.*

The remainder of this paper describes a study with dual purposes. The primary purpose is to determine whether our fault-injection environment is suitable for studying Propositions 1–3. The secondary purpose is to determine whether the propositions themselves have any credibility. The experimental apparatus (i.e., fault-injection environment) and process are described in Section 3. We specify the parameters of the pilot study in Section 4. Section 5 discusses the results of the study. A brief overview of related work is contained in Section 7. Section 8 concludes this paper.

3 An Automated Fault-Injection Environment for FlexRay

In order to study Propositions 1–3 — and, indeed, *any* propositions — the fault injection environment should provide *repeatability*, *controllability* and *observability*. We also require *automation* in order to allow unattended operation for extended periods of time.

Controllability. The experimenter should be able to define experimental parameters accurately, with respect to time (e.g., fault activation-trigger and duration), space (e.g., fault location) and value (e.g., fault type).

Repeatability. Experiments run under similar conditions with similar parameters should produce similar results.

Observability. The effect(s) — or lack thereof — of a fault should be readily apparent. *We do not assume that faults manifest symmetrically across nodes*. Therefore, *observations* (i.e., snapshots of instrumented data) must be made at each node and compared with respect to the time, space and value domains. In FlexRay, the *space* domain corresponds to the node identifier (ID); the *time* domain corresponds to the local view of global time (denoted by the current macrotick and cycle counter); and the *value* domain corresponds to the measured data itself.

Automation. While a manual fault-injection process [13] can be useful for rapid-prototyping applications, such a process becomes unwieldy when a large number of experiments are required. In order to develop a robust dataset for training and testing different diagnosis algorithms, we need to experiment with a wide range of faults. Experiments also need to be repeated many times in order to allow for statistically significant analysis.

These goals are achieved through a rigorous process that is enabled by off-the-shelf hardware (see Section 3.1) and implemented by custom software (see Section 3.2).

3.1 Hardware Architecture

The fault-injection environment is based on a cluster of (6) Elektrobit EB 6120[1] prototyping nodes that communicate with each other over a linear FlexRay communication bus. The prototyping nodes feature MFR4310 FlexRay controllers. The $^{\text{TTX}}$Disturbance node, by TTTech, provides fault-injection capabilities at the FlexRay physical-layer and protocol-layer. An Amrel ePower PDS8202 programmable power-supply provides (8) independent DC out-

[1]Formerly known as DECOMSYS NODE MPC5200

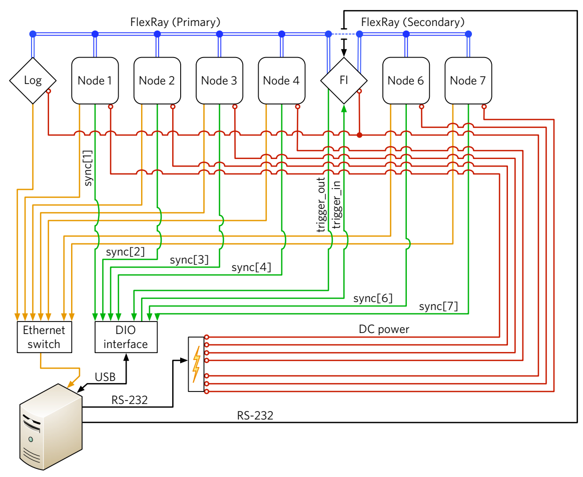


Figure 1. The hardware architecture of the fault-injection environment.

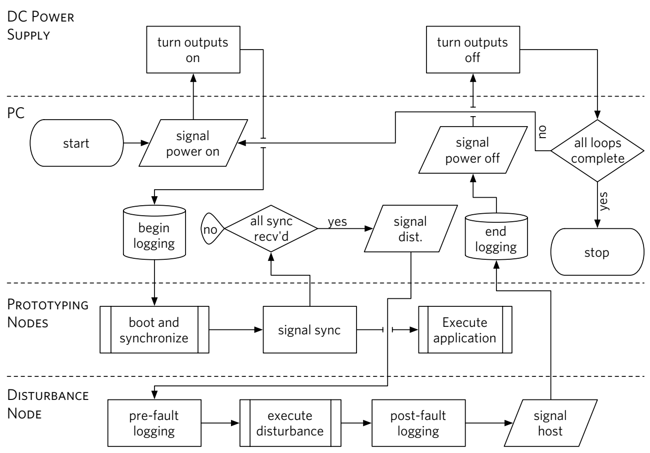


Figure 2. The fault-injection process requires coordinating actions across disparate components.

puts that allow each node to be power-cycled programmatically.

The placement of the ^{TTX}Disturbance node corresponds to the location of injected physical-layer faults. For example, in the current topology, the bus lines can be short circuited or broken between the fourth and fifth prototyping nodes (see Figure 1).

A Windows-based personal computer (PC) communicates directly with the prototyping nodes using an Ethernet backchannel. The PC also controls the power supply and ^{TTX}Disturbance node via RS-232 connections. Finally, a National Instruments USB-6008 Data Acquisition (DAQ) unit connected to the PC provides digital and analog I/O interfaces for triggers and signals. Figure 1 shows the hardware architecture in detail.

3.2 Experimental Process

The high-level experimental process is fairly straightforward.

Step 1 *Power-cycle each prototyping node to reset its internal state.*

Step 2 *As the the prototyping nodes boot and synchronize, begin logging observations.*

Step 3 *Once all of the prototyping nodes have synchronized, delay for a specified time to allow for steady-state observations (i.e., the* pre-fault delay*).*

Step 4 *Wait for a repeatable* trigger *before activating a fault of some* duration *and* type.

Step 5 *When the fault has passed, allow additional time to log observations as any lingering fault-effects pass (i.e., the* post-fault delay*).*

Step 6 *Return to Step 1 and repeat the process, if required.*

A custom control-application that runs on the PC implements the configuration, coordination, and data-collection functionality that this process requires.

3.2.1 Configuration

The *pre-fault delay* (from Step 3); fault *duration*, *type* and *trigger* (from Step 4); and *post-fault delay* (from Step 5) are set in a configuration file that is uploaded to the ^{TTX}Disturbance node by the PC over RS-232. This configuration file is known as a *disturbance scenario*, and specifies the faulty behavior that is applied during the fault-injection process.

The number of times that a particular disturbance scenario is repeated is defined by the *reps* parameter, which is provided as an input to the control application.

3.2.2 Coordination

Implementing this high-level process involves many steps taken by disparate components without any common communication channel. The PC provides the "glue" required to coordinate these steps (see Figure 2).

The PC commands the power supply to turn on its outputs, which causes the prototyping nodes to boot and synchronize. The fault should not be injected until all of the prototyping nodes have synchronized (i.e., reached a steady-state). The ^{TTX}Disturbance node does not provide a way to wait until *all* nodes achieve synchronization, so the prototyping nodes each send a `sync` signal through the DAQ to the PC. Once the PC has received all of the `sync` signals, it sends a signal through the DAQ to trigger the ^{TTX}Disturbance node, which begins running the configured disturbance scenario. The PC then waits until it receives signal from the ^{TTX}Disturbance node that the disturbance scenario has terminated. After receiving the signal, the PC commands the power supply to turn off its outputs. If the configured number of iterations has been completed, then the PC ends the process. Otherwise, the process is repeated.

3.2.3 Data Collection

The PC provides centralized data-collection functionality for observations made by the prototyping nodes. Each prototyping node is assigned to a dedicated port on the PC that listens for incoming data using User Datagram Protocol (UDP). All of the data that the PC receives is tagged with the source node ID and experiment ID and then logged for offline analysis.

4 Pilot Study Specification

For this pilot study, the prototyping nodes were loaded with a simple application. Note that the purpose of this application was not to provide realistic application-level behavior. Rather, it was only used to stimulate bus traffic. Each node transmitted a message counter and a node IDs using two frames in the static segment and a single (arbitrated) frame in the dynamic segment. These frames were received by all controllers but not read by the application. The OS task-schedule and FlexRay communication-schedule were configured to execute synchronously with a 2ms period. The nodes were connected in a linear topology, as shown in Figure 1. Each node in the network was configured as a sync-node with respect to the FlexRay protocol.

4.1 Instrumentation

The FlexRay specification defines various data structures that indicate the status of the communication protocol. Such data structures provide metrics that can be used to detect error patterns. They are accessed though the FlexRay CHI, which is accomplished on the MFR4310 controller by reading and writing 16-bit memory-mapped registers.

A custom instrumentation-component runs on each EB 6120 node and makes observations by reading a subset of the CHI registers at the beginning of each task period (e.g., once every 2ms). Observations are buffered in volatile memory on the node. The instrumentation component periodically empties the buffer by sending all of the stored observations to the PC using the Ethernet backchannel.

For this study, we observed 4 discrete error-indicators during each experiment: Boundary Violations (BVs), Syntax Errors (SERRs), Content Errors (CERRs) and Valid Frames (VFs). The indicators themselves were aggregated over the entire communication cycle. Observations were made at the beginning of each communication cycle by reading error indicators from the FlexRay CHI.

4.2 Disturbance Scenarios

The consisted of 5 disturbance scenarios, each of which was repeated 100 times. For this study, we choose to focus on a small set of physical-layer faults. Each scenario was associated with a different fault-type, which was activated for 500ms. No fault was activated during the **`none`** scenario, which provided a *baseline case*. The **`break`** scenario caused a physical separation of the Bus Plus (BP) and Bus Minus (BM) lines. The **`noise`** scenario injected differential white noise onto the bus. The **`short_vcc`** and **`short_gnd`** scenarios short-circuited the BP line to ground and supply voltage, respectively. Observations from each node were logged for 1s prior to activation and 5s following deactivation of the disturbance.

5 Results

Recall that the instrumentation component records an observation once every 2ms. Therefore, 500 observations are expected during the pre-fault period (1000 ms); 250 observations are expected while the fault is active (500 ms); and 2500 observations are expected during the post-fault period (5000 ms). For this study, we are not interested in observations made during the synchronization phase. In total, each experiment is expected to produce 3250 total observations. Note that because each observation accumulates indicators over an entire cycle, you can have multiple indicators set in a single observation (i.e., the sum of the observed indicators may be greater than the total number of observations).

`none` (baseline) As expected, no error indicators were observed during the baseline scenario.

`break` The `break` scenario resulted in SERRs, CERRs and BVs at each node, with some variation across nodes (see fig. 3). The `break` scenario was further distinguished by being the only scenario to result in CERRs (see fig. 4).

`noise` A roughly equal number of BVs and SERRs were observed during the `noise` scenario, along with a corresponding drop in VFs (see fig. 5).

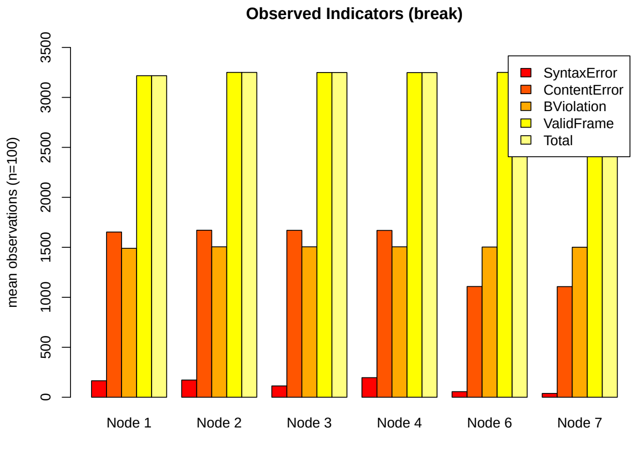


Figure 3. The `break` scenario resulted in SERRs, CERRs and BVs at each node, with some variation across nodes.

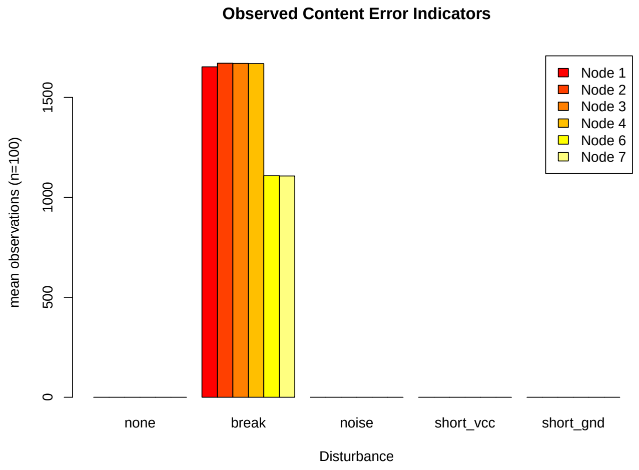


Figure 4. Content Errors (CERRs) were only observed during the `break` scenario.

`short` Short circuits resulted in a drop in valid frames, compared to the total number of observations (see fig. 6). Between three and four syntax errors were also observed consistently during each short circuit. There was no measurable difference in the effects of shorting to ground vs. shorting to supply voltage.

6 Discussion

In order to look for potential error patterns, we simply compared the average number of times a particular indicator is observed for each scenario. Thus, we can deduce the following preliminary error patterns:

`break:` many content errors and boundary violations with comparatively few syntax errors

`noise:` equal number of syntax errors and boundary violations with a corresponding drop in valid frames

`short_vcc:` lack of valid frames with comparatively few syntax errors

`short_gnd:` lack of valid frames with comparatively few syntax errors

The fault-injection process itself showed good controllability and repeatability. It was trivial to specify the experimental parameters accurately. Given consistent parameters, the process produced consistent results.

7 Related Work

A useful overview of general fault injection techniques is available in [9]. Here, we focus on fault injection techniques that specifically target the automotive domain, with an emphasis on diagnosis.

Fault-injection experiments using heavy-ion fault injection have shown fail-silence violations [20] and error propagation [1] in Time-Triggered Protocol/Class-C (TTP/C). Fault-injection experiments using hardware models have show that transient faults in the CAN Communication Controller (CC) and Communication Network Interface (CNI) can result in masquerade failures, where a faulty node "impersonates" a non-faulty node [17]. Within the context of diagnosis, this result can be viewed as a false-positive on the non-faulty node as well as a false-negative on the true faulty node. Experiments performed using modeled FlexRay controllers have highlighted instances of error propagation in FlexRay bus and star topologies [5].

The online diagnosis algorithms developed by [22] were extended to discriminate healthy nodes from unhealthy nodes in time-triggered automotive systems [18]. A prototype implementation of the protocol was built using TTP/C controllers and analyzed via physical fault injection [19].

Out-of-Norm Assertions (ONAs) are introduced as a way to correlate fault effects in the three dimensions of value, time and space . The ONA mechanism underlies a framework for diagnosing failures in time-triggered networks [14]. A prototype implementation of the framework using TTP/C controllers instruments the frame status field of the controller, which is similar to the information available from the FlexRay CHI. The $^{\mathrm{TTP}}$Disturbance node was used

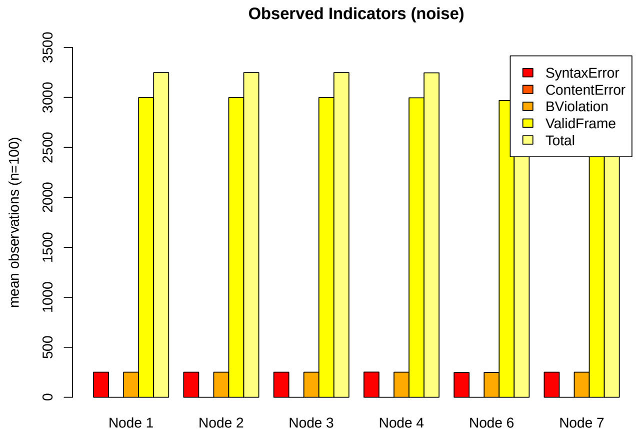


Figure 5. The `noise` scenario produced a roughly equal number of BVs and SERRs with a corresponding drop in VFs.

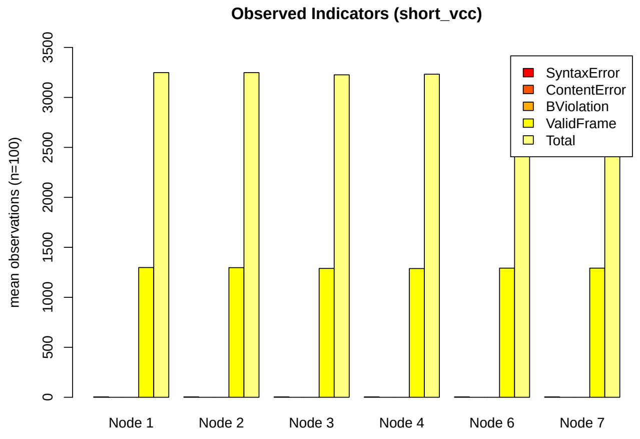


Figure 6. Both short-circuit scenarios produced similar drops in valid frames.

along with Electromagnetic Interference (EMI) injection to evaluate the effectiveness of this diagnostic framework as applied to connector faults TTP/C [15].

8 Conclusion

These preliminary results suggest that (1) *errors do not manifest uniformly across nodes* and (2) *identifiable error patterns may exist*. However, these results cannot be generalized beyond this small study. These patterns are likely to change, perhaps drastically, depending on many factors such as network topology, communication schedule, node configuration, etc. Furthermore, other disturbances that we did not include in this pilot might show similar patterns, making them difficult to distinguish from each other. Clearly, more advanced analysis will be required for more robust fault models.

References

[1] A. Ademaj, H. Sivencrona, G. Bauer, and J. Torin. Evaluation of fault handling of the time-triggered architecture with bus and star topology. In *Proceedings, 2003 International Conference on Dependable Systems and Networks*, DSN '03, pages 123–132, Los Alamitos, CA, USA, June 2003. IEEE Computer Society.

[2] J. Arlat, M. Aquera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, February 1990.

[3] S. Bhatia, A. Kumar, M. E. Fiuczynski, and L. L. Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proceedings, 8th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '08, pages 103–116, Berkeley, CA, USA, December 2008. USENIX Association.

[4] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. *ACM SIGOPS Operating Systems Review*, 39(5):105–118, December 2005.

[5] M. Dehbashi, V. Lari, S. G. Miremadi, and M. Shokrollah-Shirazi. Fault effects in flexray-based networks with hybrid topology. In *Proceedings, 3rd International Conference on Availability, Reliability and Security*, ARES '08, pages 491–496, Los Alamitos, CA, USA, March 2008. IEEE Computer Society.

[6] FlexRay Consortium. *FlexRay Communications System Protocol Specification*, December 2005.

[7] H.-G. Frischkorn. Automotive software – the silent revolution. Automotive Software Workshop, San Diego, CA, Jan 2004.

[8] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *Proceedings, 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOSPLA '04, pages 251–269, New York, NY, USA, October 2004. ACM.

[9] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, April 1997.

[10] *ISO/DIS 26262: Road vehicles – Functional safety*, volume 4–6. International Organization for Standardization, Geneva, Switzerland, 2009.

[11] S. Kavulya, R. Gandhi, and P. Narasimhan. Gumshoe: Diagnosing performance problems in replicated file-systems. In *Proceedings, 2008 IEEE Symposium on Reliable Dis-*

tributed Systems, SRDS ’08, pages 137–146, Los Alamitos, CA, USA, October 2008. IEEE Computer Society.

[12] P. E. Lanigan and P. Narasimhan. Holistic data-driven diagnosis for dependable automotive systems. Appeared in *NIST/NSF/USCAR Workshop on Developing Dependable and Secure Automotive Cyber-Physical Systems from Components*, March 2011. Available at http://varma.ece.cmu.edu/Auto-CPS-2011/Papers/index.html.

[13] P. E. Lanigan, P. Narasimhan, and T. E. Fuhrman. Experiences with a CANoe-based fault injection framework for AUTOSAR. In *Proceedings, 2010 IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN ’10, pages 569—574, Los Alamitos, CA, USA, June 2010. IEEE Computer Society.

[14] P. Peti and R. Obermaisser. A diagnostic framework for integrated time-triggered architectures. In *Proceedings, 9th IEEE International Symposium on Object Oriented Real-Time Distributed Computing*, ISORC ’06, page 11pp, Los Alamitos, CA, USA, April 2006. IEEE Computer Society.

[15] P. Peti, R. Obermaisser, and H. Paulitsch. Investigating connector faults in the time-triggered architecture. In *Proceedings, 11th IEEE Conference on Emerging Technologies and Factory Automation*, ETFA ’06, pages 887–896, Piscataway, NJ, USA, September 2006. IEEE.

[16] J. D. Rupp and A. G. King. Autonomous driving – a practical roadmap. SAE Technical Paper Series 2010-01-2335, SAE International, Warrendale, PA, USA, October 2010.

[17] H. Salmani and S. G. Miremadi. Contribution of controller area networks controllers to masquerade failures. In *Proceedings, 11th Pacific Rim International Symposium on Dependable Computing*, PRDC ’05, page 5, Los Alamitos, CA, USA, December 2005. IEEE Computer Society.

[18] M. Serafini, A. Bondavalli, and N. Suri. Online diagnosis and recovery: On the choice and impact of tuning parameters. *IEEE Transactions on Dependable and Secure Computing*, 4(4):295–312, October-November 2007.

[19] M. Serafini, N. Suri, J. Vinter, A. Ademaj, W. Brandstäter, F. Tagliabò, and J. Koch. A tunable add-on diagnostic protocol for time-triggered systems. In *Proceedings, 2007 IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN ’07, pages 164–174, Los Alamitos, CA, USA, June 2007. IEEE Computer Society.

[20] H. Sivencrona, P. Johannessen, and J. Torin. Protocol membership in dependable distributed communication systems – a question of brittleness. SAE Technical Paper Series 2993-01-0108, SAE International, Warrendale, PA, USA, March 2003.

[21] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. N. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, M. Gittleman, S. Harbaugh, M. Hebert, T. M. Howard, S. Kolski, A. Kelly, M. Likhachev, M. McNaughton, N. Miller, K. Peterson, B. Pilnick, R. Rajkumar, P. Rybski, B. Salesky, Y.-W. Seo, S. Singh, J. Snider, A. Stentz, W. R. Whittaker, Z. Wolkowicki, J. Ziglar, H. Bae, T. Brown, D. Demitrish, B. Litkouhi, J. Nickolaou, V. Sadekar, W. Zhang, J. Struble, M. Taylor, M. Darms, and D. Ferguson. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8):425–466, July 2008.

[22] C. J. Walter, P. Lincoln, and N. Suri. Formally verified online diagnosis. *IEEE Transactions on Software Engineering*, 23(11):684–721, November 1997.

[23] D. Wilson. Ray of hope for auto industry. *Electronic Business*, Nov 2006. Available at http://www.edn.com/article/CA6385672.html.

Acronyms

BM	Bus Minus	4
BP	Bus Plus	4
BV	Boundary Violation	4
CC	Communication Controller	5
CERR	Content Error	4
CHI	Controller-Host Interface	2
CNI	Communication Network Interface	5
DAQ	Data Acquisition	3
EMI	Electromagnetic Interference	6
ID	identifier	2
OEM	Original Equipment Manufacturer	2
ONA	Out-of-Norm Assertion	5
OS	Operating System	2
PC	personal computer	3
SERR	Syntax Error	4
TTP/C	Time-Triggered Protocol/Class-C	5
UDP	User Datagram Protocol	4
VF	Valid Frame	4

On Methods for the Formal Specification of Fault Tolerant Systems

Manuel Mazzara
School of Computing Science, Newcastle University, UK
Manuel.Mazzara@newcastle.ac.uk

***Abstract*—This paper introduces different views for understanding problems and faults with the goal of defining a method for the formal specification of systems. The idea of Layered Fault Tolerant Specification (LFTS) is proposed to make the method extensible to fault tolerant systems. The principle is layering the specification in different levels, the first one for the *normal behavior* and the others for the *abnormal*. The abnormal behavior is described in terms of an Error Injector (EI), which represents a model of the erroneous interference coming from the environment. This structure has been inspired by the notion of idealized fault tolerant component but the combination of LFTS and EI using Rely/Guarantee reasoning to describe their interaction can be considered as a novel contribution. The progress toward this method and this way to organize fault tolerant specifications has been made experimenting on case studies and an example is presented.**

***Keywords*-Formal Methods; Layered Fault Tolerant Specification; Problem Frames; Rely/Guarantee.**

I. Introduction

There is a long tradition of approaching Requirements Engineering (RE) by means of formal or semi-formal techniques. Although ”fuzzy” human skills are involved in the process of elicitation, analysis and specification - as in any other human field - still methodology and formalisms can play an important role [19]. However, the main RE problem has always been communication. A definition of communication teaches us that [9]:

> “Human communication is a process during which source individuals initiate messages using conventionalized symbols, nonverbal signs, and contextual cues to express meanings by transmitting information in such a way that the receiving party constructs similar or parallel understanding or parties toward whom the messages are directed.”

The first thing we have realized in building dependable software is that it is necessary to build dependable communication between parties that use different languages and vocabulary. In the above definition you can easily find the words ”*similar or parallel understanding are constructed by the receiving parties*”, but for building dependable systems matching expectations (and specification) it is not enough to build a *similar or parallel understandings* since we want a more precise mapping between intentions and actions. Formal methods in system specification look to be an approachable solution.

Object Oriented Design [6] and Component Computing [26] are just well known examples of how some rigor and discipline can improve the final quality of software artifacts besides the human communication factor. The success of languages like Java or C# could be interpreted in this sense, as natural target languages for this way of structuring thinking and design. It is also true - and it is worth reminding it - that in many cases it has been the language and the available tools on the market that forced designers to adopt object orientation principles, for example, and not vice versa. This is the clear confirmation that it is always a combination of conceptual and software tools together that create the right environment for the success of a discipline.

Semi-formal notations like UML [10] helped in creating a language that can be understood by both specialists and non specialists, providing different views of the system that can be negotiated between different stakeholders with different backgrounds. The power (and thus the limitation of UML) is the absence of a formal semantics (many attempts can be found in the literature anyway) and the strong commitment on a way of reasoning and structuring problems which is clearly the one disciplined by object orientation. Many other formal/mathematical notations existed for a long time for specifying and verifying systems like process algebras (a short history by Jos Baeten in [3]) or specification languages like Z (early description in [2]) and B [1]. The Vienna Development Method (VDM) is maybe one of the first attempts to establish a Formal Method for the development of computer systems [5]. A survey on these (and others) formalisms can be found in [22]. All these notations are very specific and can be understood only by specialists. The point about all these formalisms is that they are indeed notations, formal or semi-formal. Behind each of them there is a way of structuring thinking that does not offer complete freedom and thus forces designers to adhere to some discipline. But still they are not methods in the proper sense, they are indeed languages.

Contributions of the paper

The goal of this paper is providing a different view for interpreting problems and faults. The overall result will be the definition of a method for the specification of systems that do not run in isolation but in the real, physical world. In [20], we mainly defined a draft of this approach contributing with an understanding of what a method is and an analysis of the desiderata. We then presented our method and its application to a Train System example. We realized that few points were still at a draft stage and their explanation still obscure in some paragraphs. In this paper we will provide more details instead and a different example. The main contributions of this work can be considered:

1) A perspective for describing problems in term of static view and dynamic view and and a discussion on how to combine them
2) A perspective to describe faults in terms of an Error Injector representing a model of faults (and consequent introduction of fault tolerant behavior)
3) The organization of the specification in terms of layers of Rely Guarantee conditions (LFTS)
4) The experimentation on a small automotive case study

In particular, in Section 2, problems are described in terms of static view (based on Problem Frames) and dynamic view (built on top of rely/guarantee conditions). Section 3 introduces faults and the idea of Layered Fault Tolerant Specification (LFTS) which is then applied in section 4 on a very simple example. Section 5 draws conclusions on top of what has been shown in the paper.

II. An Angle to See Problems

Our work in this paper focuses especially on [17] where the original idea of a formal method for the specification of systems running in the physical world originated. That paper was full of interesting ideas but still was lacking of a method in the sense we described in [20] and [21]. Few case studies have been analyzed according to this philosophy in [7] but still a complete method has not been reached. For this reason we think now that a more structured approach is urgent in this area. Thus, the goal of the present work is improving our understanding of those ideas and incrementing that contribution putting it in an homogeneous and uniform way and describing a method featuring the properties we introduced in [20], with particular attention to fault tolerance. In Figure 1, we report a graphical synthesis of the Descartes method presented in [24]. This work presents a method as consisting of a partially ordered set of actions which need to be performed and then discharged within a specific causal relationship. The success of one action determines the following ones. Furthermore, the method has to be repeatable, possibly by non experts or specialists.

At the moment we have had some progress in this direction but we still need more work toward a method for the specification of fault tolerant systems. The basic idea behind [17] was to specify a system not in isolation but considering the environment in which it is going to run and

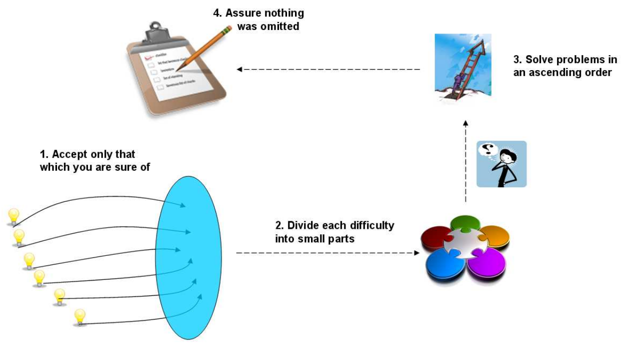

4. Assure nothing was omitted
3. Solve problems in an ascending order
1. Accept only that which you are sure of
2. Divide each difficulty into small parts

Figure 1. The method of science

deriving the final specification from a wider system where assumptions have been understood and formalized as layers of rely conditions. Here the difference between assumptions and requirements is crucial, especially when considering the proper fault tolerance aspects. We could briefly summarize this philosophy as follows:

- Not specifying the digital system in isolation
- Deriving the specification starting from a wider system in which physical phenomena are measurable
- Assumptions about the physical components can be recorded as layers of rely-conditions (starting with stronger assumptions and then weakening when faults are considered)

Sometimes, we have found useful, in the presentation of these concepts, to use Figure 2. This figure allows us to show how a computer system can be seen from a different angle, as not consisting of functions performing tasks in isolation but as relationships (interfaces/contracts) in a wider world including both the machine and the physical (measurable) reality. As we will see later, this philosophy has been inspired by Michael Jackson's approach to software requirements analysis typically called Problem Frames approach [14]. The Silicon Package is the software running on the hosting machine. It should be clear that the machine itself can neither acquire information on the reality around nor modify it. The machine can only operate trough sensors and actuators. To better understand this point, we like to use a similar metaphor about humans where it is easier to realize that our brain/mind system (our Silicon Package?) cannot acquire information about the world but it can only do that through eyes, ears and so on (our sensors). In the same way it cannot modify the world if not through our arms, voice, etc (our actuators). So, as we start describing problems in the real world in terms of what we perceive and what we do (and not about our brain functioning) it makes sense to adopt a similar philosophy for computer systems consisting of sensors and actuators. Around the Silicon Package you can see a red circle representing the problem world and green small spheres representing the assumptions that need to be made regarding it. The arrows and their directions represent the fact that we want to derive the specification of the silicon package starting from the wider system. The way in which we record these assumptions is a topic for the following sections.

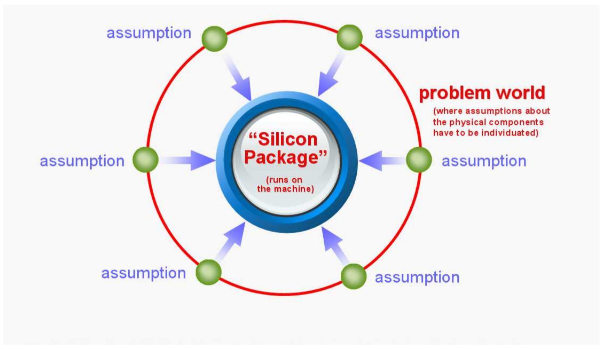


Figure 2. Silicon Package, Problem World and Assumptions

The method, its Steps and its Views

In [20], we analyzed the method introduced in [17] according to the properties described in [24]. To do that, we recognized three macroscopic steps:

1) Define boundaries of the systems
2) Expose and record assumptions
3) Derive the specification

Our idea is not committing to a single language/notation - we want a formal method, not a formal language - so we will define a general high level approach following these guidelines and we will suggest *reference tools* to cope with these steps. It is worth noting that these are only reference tools that are *suggested* to the designers because of a wider experience regarding them from our side. A formal notation can be the final product of the method but it still needs to be not confused with the method itself. In Figure 3, these steps are presented and it is shown how different tools could fit the method at different stages. We call these notations the plug-ins since they can be plugged into the steps.

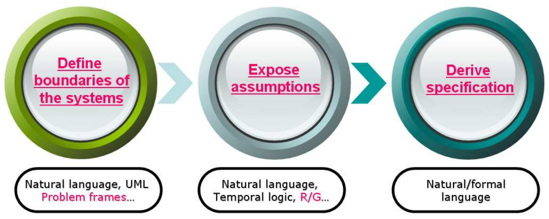

Define boundaries of the systems
Expose assumptions
Derive specification
Natural language, UML Problem frames...
Natural language, Temporal logic, R/G...
Natural/formal language

Figure 3. Steps and Reference Tools

Figure 3 is a generic representation of the method where we want to emphasize the different steps, which were not clearly defined in [17]. The reader will understand that this is still a simplification of the process. We use the word ”steps” instead of ”phases” since we do not want to suggest a sort of linear process, which is not always applicable, especially when coping with fault tolerance (as we will discuss later). We imagine, in the general case, many iterations between the different steps. The idea of the method is to ground the view of the silicon package in the external physical world. This is the problem world where assumptions about the physical components *outside* the computer itself have to be recorded. Only after this can we derive the specification for the software that will run *inside* the computer. A more precise formalization of the method and the features it has to exhibit is one of the main contributions of [20]. The reader is probably realizing that what we are obtaining here is a method exploiting two different perspectives during the three steps.

- a *static view* defining the boundaries of the system and representing the relationships between phenomena and domains in it. Our reference tools here are Problem Diagrams [14].
- a *dynamic view* representing the interactions between different processes in the system and able to record the assumptions. Our mathematical reference tools here are rely/guarantee conditions [16], [15], which regard the execution of concurrently executing (and interfering) processes.

Furthermore we need an approach to consider faulty behavior. This will be described later in the related section. The idea behind having two different views is that different people (or stakeholders) could possibly be interested only in single aspects of the specification and be able to understand only one of the possible projections. In this way you can approach the specification without a full understanding of every single aspect.

Static View

Michael Jackson is well known for having pioneered, in the seventies (with Jean-Dominique Warnier and Ken Orr) the technique for structuring programming basing on correspondences between data stream structure and program structure [12]. Jackson's ideas acquired then the acronym JSP (Jackson Structured Programming). In his following contribution [13], Jackson extended the scope to systems. Jackson System Development (JSD) already contained some of the ideas that made object-oriented program design famous.

In this section, we describe our reference tool for representing the relationships between phenomena and domains of the system we want to specify using Problem Diagrams [14]. Context Diagrams and Problem Diagrams are the graphical notations introduced by Michael Jackson (in the time frame 1995/2001) in his Problem Frames (PF) approach to software requirements analysis. This approach consists of a set of concepts for gathering requirements and creating specifications of software systems. As previously explained, the new philosophy behind PF is that user requirements are here seen as being about relationships in the operational context and not functions that the software system must perform. It is someway a change of perspective with respect to other requirements analysis techniques.

The entire PF software specification goal is modifying the world (the problem environment) through the creation of a dedicated machine, which will be then put into operation in this world. The machine will then operate bringing the desired effects. The overall philosophy is that the problem is located in the world and the solution in the machine. The most important difference with respect to other requirements methodologies is the emphasis on describing the environment and not the machine or its interfaces. Let us consider, for example, the Use Case approach [4]. What is done here is specifying the interface, the focus is on the interaction user-machine. With PF we are pushing our attention beyond the machine interface, we are looking into the real world. The problem is there and it is worth starting there. The first two points of the ideas taken from [17] (not specifying the digital system in isolation and deriving the specification starting from a wider system in which physical phenomena are measurable) can be indeed tracked back, with some further evolution, to [14]. In this work, we are using PF to develop a method for specification of systems, i.e., a description of the machine behavior. But, before doing that, we need to start understanding the problem.

Context Diagrams

The modeling activity of a system should start using this kind of diagram in the PF philosophy. By means of it we are able to identify the boundaries of the system, where a system is intended as the machine to be designed (software + hardware) and its domains with their connections (in terms of shared phenomena). It is part of what we call a static view of the system.

Context Diagrams contain an explicit and graphical representation of:

- the machine to be built
- the problem domains that are relevant to the problem
- the interface (where the Machine and the application domain interact)

A domain here is considered to be a part of the world we are interested in (phenomena, people, events). A domain interface is where domains communicate. It does not represent data flow or messages but shared phenomena (existing in both domains). Figure 4 shows a simple scenario. The lines represent domain interfaces, i.e., where domains overlap and share phenomena.

Problem Diagrams

The basic tool for describing a problem is a Problem Diagram, which can be considered a refinement of a Context Diagrams. This should be the 2nd step of the modeling

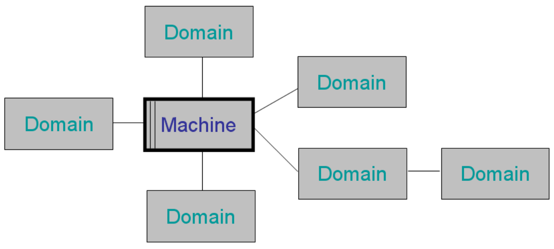


Figure 4. Context Diagram

process. A problem diagram shows the requirements on the system, its domains, and their connections. It is still part of a static view of the system but better represents the assumptions about the system and its environment. They are basic tools to describe problems. To the information contained in context diagrams they add:

- dotted oval for requirements
- dotted lines for requirements references

Figure 5 shows a scenario where the Silicon Package is in charge of monitoring the patients conditions. We believe that the first step of the specification method (define boundaries of the systems) can be accomplished by means of this tools. Thus we use Problem Diagrams as a reference tool for our research but still, as said, not constraining it to a specific notation or language.

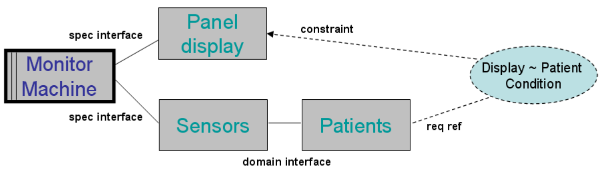


Figure 5. Problem Diagram

Dynamic View

Problem Diagrams taken from the PF approach are a notation that forces us to think about the problem in the physical world instead of focusing immediately on the solution. We believe that they represent an effective tool to define the precise boundaries of the specification we are working on. Summarizing they represent:

1) the machine
2) the problem domains
3) the domain interfaces
4) the requirements to bring about certain effects in the problem domains
5) references in the requirements to phenomena in the problem domains

Once the domains of the context we are working on, their phenomena and the relative overlap have been understood, it will be necessary to focus on the ”border” between the Silicon Package and the real world. It is necessary to distinguish between assumptions and requirements and we need a tool to record assumptions. Our system will be composed of interacting parts and each of these parts will also interact with the world. The world itself has to be understood in term of assumptions about normal/abnormal behavior and a model of fault need to be considered. For all these reason we introduce the concept of *dynamic view*, which represents the interactions between processes in the system and between the system and the world. To record our assumption (as we will see layers of assumption for fault tolerance) we use a mathematical reference tool, i.e., rely/guarantee conditions [16], [15], which regard the execution of concurrently executing processes. R/G conditions are a powerful abstraction for reasoning about interference and they originated in the Hoare logic idea of preconditions and postconditions [11]. The purpose is providing a set of logical rules for reasoning about the correctness of programs. We will explain the idea through examples, for more details please consider the literature. As the reader will realize in this section, rely conditions can be used to record assumptions in the overall context of the proposed method. However, as stated in [23], when they show too much complication this might be a warning indicating a messy interface.

Preconditions and Postcondition

To understand the power of the R/G reasoning it is necessary to realize how preconditions and postconditions can help in specifying a software program when interference does not play its role. What we have to describe (by means of logical formulas) when following this approach is:

1) the input domain and the output range of the program
2) the precondition, i.e., the predicate that we expect to be true at the beginning of the execution
3) the postcondition, i.e., the predicate that will be true at the end of the execution provided that the precondition holds

Preconditions and postconditions represent a sort of contracts between parties: provided that you (the environment, the user, another system) can ensure the validity of a certain condition, the implementation will surely modify the state in such a way that another known condition holds. There is no probability here, it is just logic: if this holds that will hold. And the input-output relation is regulated by a predicate that any implementation has to satisfy.

We show the example of a very simple program, the specification of which in the natural language may be: *“Find the smallest element in a set of natural numbers”*.

This very simple natural language sentence tells us that the smallest element has to be found in *a set of natural numbers*. So the output of our program has necessarily to be a natural number. The input domain and the output range of the program are then easy to describe:

$$I/O : \mathcal{P}(\mathbf{N}) \rightarrow \mathbf{N}$$

Now, you expect your input to be a set of natural numbers, but to be able to compute the min such a set has to be non empty since the min is not defined for empty sets. So the preconditions that has to hold will be:

$$P(S) : S \neq \emptyset$$

Provided that the input is a set of natural numbers *and* it is not empty, the implementation will be able to compute the min element, which is the one satisfying the following:

$$Q(S, r) : r \in S \wedge (\forall e \in S)(r \leq e)$$

Given this set of rules, the input-output relation is given by the following predicate that needs to be satisfied by any implementation f:

$$\forall S \in \mathcal{P}(\mathbf{N})(P(S) \Rightarrow f(S) \in \mathbf{N} \wedge Q(S, f(S)))$$

Interference

The example just shown summarizes the power (and the limitations) of this kind of abstractions. To better understand the limitations consider Figure 6 where interference and global state are depicted. The two processes alternate their execution and access the state. The global state can consist of shared variables or can be a queue of messages if message passing is the paradigm adopted. This figure shows exactly the situations described in [16], quoting precisely that work:

> As soon as the possibility of other programs (processes) running in parallel is admitted, there is a danger of ”interference.” Of more interest are the places where it is required to permit parallel processes to cooperate by changing and referencing the same variables. It is then necessary to show that the interference assumptions of the parallel processes coexist.

Another quote from [8] says:

> The essence of concurrency is interference: shared-variable programs must be designed so as to tolerate state changes; communication-based concurrency shifts the interference to that from messages. One possible way of specifying interference is to use rely/guarantee-conditions.

In case we consider interfering processes, we need to accept that the environment can alter the global state. However,the idea behind R/G is that we impose these changes to be constrained. Any state change made by the environment (other concurrent processes with respect to the one we are considering) can be assumed to satisfy a condition called R (rely) and the process under analysis can change its state only in such a way that observations by other processes will consist of pairs of states satisfying a condition G (guarantee). Thus, the process *relying* on the fact that a given condition holds can *guarantee* another specific condition. An example is now presented.

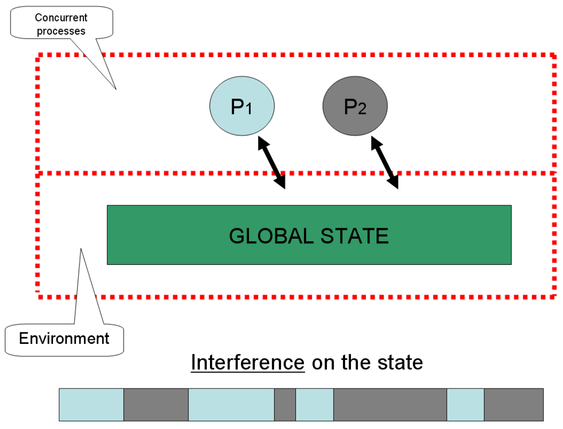


Figure 6. Interference trough global state

Greatest Common Divisor

Consider the two following simple pieces of code, the co-operation of which calculates the Greatest Common Divisor:

```
P1:                          P2:
while(a<>b){                 while(a<>b){
  if(a > b)                    if(b > a)
    a := a-b;                    b := b-a;
}                            }
```

P1 is in charge of decrementing *a* and P2 of decrementing *b*. When $a = b$ will evaluate to true it means that one is the Greatest Common Divisor for *a* and *b*. The specification of the interactions is as follows:

$$R_1 : (a = \overline{a}) \wedge (a \geq b \Rightarrow b = \overline{b}) \wedge (GCD(a,b) = GCD(\overline{a},\overline{b}))$$
$$G_1 : (b = \overline{b}) \wedge (a \leq b \Rightarrow a = \overline{a}) \wedge (GCD(a,b) = GCD(\overline{a},\overline{b}))$$
$$R_2 = G_1$$
$$G_2 = R_1$$

Here the values $\overline{a}$ and $\overline{b}$ are used instead of *a* and *b* when we want to distinguish between the values before the execution and the values after. P1 relies on the fact that P2 is not changing the value of *a* and $a \geq b$ means no decrements for *b* have been performed. Furthermore the CGD did not change. Specular situation is for the guarantee condition. Obviously, what is a guarantee for P1 becomes a rely for P2 and vice versa.

Need for Extension (of Jackson's Diagrams)?

The objective of a PF analysis is the decomposition of a problem into a set of subproblems, where each of these matches a problem frame. A problem frame is a problem pattern, i.e the description of a simple and generic problem for which the solution is already known. There are four main patterns plus some variations:

- required behavior (the behavior of a part of the physical world has to be controlled)
- commanded behavior (the behavior of a part of the physical world has to be controlled in accordance with commands issued by an operator)
- information display (a part of the physical world states and behavior is continuously needed)
- simple workpieces (a tool is needed for a user to create/edit a class of text or graphic objects so that they can be copied, printed...)

Our perception is that, when describing the behavior of interfering processes - especially when faults are considered as a special case of interference (see next section) - the diagrams and the patterns provided are not powerful enough. We need further refinement steps filling the gap between the static and the dynamic view to complete the specification process. Now we briefly describe these ideas that needs further work and can be considered an open issue.

Interface Diagram

In a 3rd step of the modeling process, we want to represent an external, static view of the system. We need a further refinement of the Problem Diagram able to identify the operations of the system and its domains, and the input/output data of these operations (with their types). The relationship of these with the requirements identified in the Problem Diagram has to be represented at this stage.

Process Diagram

In a 4th step of the modeling process, the whole system is represented as a sequential process and each of its domains as a sequential process. Concurrency within the system or within its domains is modeled by representing these as two or more subcomponents plus their rely and guarantee conditions. This is an external, dynamic view of the system and its domains.

III. An Angle to See Faults

Testing can never guarantee that software is correct. Nevertheless, for specific software features - especially the ones involving human actions and interactions - rigorous testing still remains the best choice to build the desired software. We know very little about human behavior, there are few works trying to categorize, for example, human errors in such a way that we can design system that can prevent bad consequences [25] but this goes far beyond the scope of this work. Here we want to focus on the goal of deploying highly reliable software in terms of aspects that can be quantified (measured), for example the functional input/output relation (or input/output plus interference, as we have seen). In this case, formal methods and languages provide some support. The previous sections discussed how to derive a specification of a system looking at the physical world in which it is going to run. No mention has been made of fault tolerance and abnormal situations which deviate from the basic specification. The reader will soon realize that the method we have defined does not directly deal with these issues but it does not prevent fault tolerance from playing a role. The three steps simply represent what you have to follow to specify a system and they do not depend on what you are actually specifying. This allows us to introduce more considerations and to apply the idea to a wider class of systems. Usually, in the formal specification of sequential programs, widening the precondition leads to make a system more robust. The same can be done weakening rely conditions. For example, if eliminating a precondition the system can still satisfy the requirements this means we are in presence of a more robust system. In this paper we will follow this approach presenting the notion of Layered Fault Tolerant Specification (LFTS) and examining the idea of fault as interference [8], i.e., a different angle to perceive system faults. Quoting [8]:

> The essence of this section is to argue that faults can be viewed as interference in the same way that concurrent processes bring about changes beyond the control of the process whose specification and design are being considered.

The idea of Layered Fault Tolerant Specification (LFTS) is now presented in combination with the approach quoted above making use of rely/guarantee reasoning. The principle is layering the specification, for the sake of clarity, in (at least) two different levels, the first one for the *normal behavior* and the others (if more than one) for the *abnormal*. This approach originated from the notion of idealized fault tolerant component [18] but the combination of LFTS and rely guarantee reasoning can be considered one of the main contributions of this work.

Fault Model

First, when specifying concurrent (interfering) processes, we need to define which kind of abnormal situations we are considering. We basically need to define a Fault Model, i.e., what can go wrong and what cannot. Our specification will then take into account that the software will run in an environment when specific things can behave in an ”abnormal” way. There are three main abnormal situations in which we can incur, they can be considered in both the shared variables and message passing paradigm:

- Deleting state update: “lost messages”
- Duplicating state update: “duplicated messages”
- Additional state update (malicious): “fake messages created”

The first one means that a message (or the update of a shared variable) has been lost, i.e., its effect will not be taken into account as if it never happened. The second one regards a situation in which a message has been intentionally sent once (or a variable update has been done once) but the actual result is that it has been sent (or performed) twice because of a faulty interference. The last case is the malicious one, i.e., it has to be done intentionally (by a human, it cannot happen only because of hardware, middleware or software malfunctioning). In this case a fake message (or update) is created from scratch containing unwanted information.

Our model of fault is represented by a so-called *Error Injector* (EI). The way in which we use the word here is different with respect to other literature where Fault Injector or similar are discussed. Here we only mean a model of the erroneous behavior of the environment. This behavior will be limited depending on the number of abnormal cases we intend to consider and the EI will always play its role respecting the RG rules we will provide. In the example we will show in the following we are only considering the first of the three cases, i.e., the Fault Injector is only operating through lost messages.

A contribution of this work is the organization of the specification in terms of layers of Rely/Guarantee conditions. In order to do this we introduce the idea of EI as a model of the environment and we need to describe how the EI will behave and how we can limit it. Here a process will rely on a specific faulty behavior and, given that, will guarantee the ability to handle these situations. More in detail:

- Rely: the Error Injector (environment) interferes with the process (changing the global state) respecting his G (superset of the program’s R) — for example, only “lost messages” can be handled (next example)
- Guarantee: The process provided this kind of (restricted) interference is able to handle exceptional/abnormal (low frequency) situations

All the possibilities of faults in the system are described in these terms and the specification is organized according to the LFTS principle we are going to describe.

LFTS: how to organize a clear specification

The main motto for LFTS is: ”Do not put all in the normal mode”. From the expressiveness point of view, a monolithic specification can include all the aspects, faulty and non faulty of a system in the same way as it is not necessary to organize a program in functions, procedures or classes. The matter here is pragmatics, we believe that following the LFTS principles a specification can be more understandable for all the stakeholders involved.

The specification has to be separated in (at least) two layers, one for the *Normal Mode* and one (or more) for the *Abnormal Mode*. More specifically:

- Normal mode: an operation usually runs in normal mode respecting his “interface” with the world determined by P/Q

- Fault interference: in “low” frequency cases the abnormal mode is “activated” (exception handler, forward recovery)

Figure 7 shows the organization of a process (dashed rectangles) in a main part and a *recovery handler* part where both interact through the global state with other processes and the Error Injector (represented by a devil here).

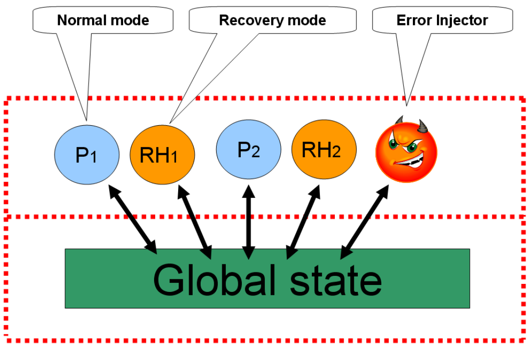


Figure 7. Error Injector

It is worth noting the limitations of this way of operating. Self error detection and self recovery cannot be addressed by this model since EI is a representation of the environment external to the process itself. So faulty behavior due to internal malfunctioning is not what we want to represent here.

Example of Specification of Interference

For a better understanding of how we can exploit this idea of treating faults as extraordinary interference with a low frequency, we introduce a very simple example. First we consider an even simpler example without interference, then we introduce interference to investigate the differences and how we cope with them.

Increments without Interference: Let us consider the following piece of code:

```
C(n):
n’ := n;
while (n’>0){
      n’ := n’-1;
      count ++
}
return count;
```

C is a very simple program, which decrements its input while reaching zero. While decrementing the input it increments a counter with the effect that, at the end, the counter will obviously reach the original value of the input. The specification of C in terms of pre and postconditions is given as follows:

$$I/O : \mathbf{N} \rightarrow \mathbf{N}$$

The input (*n*) and the output (*count*) are natural numbers. The precondition that has to hold is:

$$P(count) : count = 0$$

since we expect the counter to be zero at the beginning. Provided that the input is a natural number *and* the counter is zero, the execution will satisfy the following:

$$Q(n, count) : count = n \wedge \overline{n} = 0$$

Without any interference, the specification of C only requires that the input-output relation satisfy the predicate:

$$\forall a \in \mathbf{N}(P(a) \Rightarrow C(a) \in \mathbf{N} \wedge Q(a, C(a)))$$

Increments with Faulty Interference: Let us consider the same piece of code:

```
C(n):
n’:= n;
while (n’>0){
      n’ := n’-1;
      count ++
}
return count;
```

but running in an environment where the following EI is also running:

```
EI(n’):
if (n’>0){
    n’ := n’+1;
}
```

The role of this EI here is to model the deletion of state updates as in the first of the three cases discussed above. The specification of C as expressed so far is too simple to be able to manage this kind of situations. Even if we are not handling malicious updates, the basic formulation we provided so far needs to be properly incremented because without any changes the desired implication cannot be satisfied:

$$\forall a \in \mathbf{N}(P(a) \not\Rightarrow C(a) \in \mathbf{N} \wedge Q(a, C(a)))$$

What we have to do is restructure the implementation and to pass from pre and postconditions to rely/guarantee in the specification. Let us consider the following modification:

```
C(n):
n’:= n;
while (n’>0){
   if n’+ count = n then {
    n’ := n’-1;
    count ++
  } else {
    n’ := n-count-1
}
}
return count;
```

As the reader will understand what we have done is simply add a recovery handler and a recovery mode based on the evaluation of the condition $n + count = n$ which is able to flag the presence of an unwanted interference (a deletion of an increment). The recovery block is able to cope with abnormal situations provided that faults are restricted in behavior (and that it is known in advance). Thus, provided that a restricted interference happens the program is still able to satisfy the postcondition (and the specification). The normal mode here is the simple code:

```
n' := n'-1;
count ++
```

while the recovery handler is

```
n' := n-count-1
```

and, as represented in Figure 7, C is running in an environment which is shared with EI. The specification we want in this case is different from the previous one and it is expressed, in terms of R/G conditions, as follows:

$R_C : (\overline{n} = n) \wedge (\overline{count} = count) \wedge (\overline{n'} > n')$
$G_C : n' = n - count - 1$
$R_I = true$
$G_I = n' > \overline{n'}$

It is worth noting that there is no rely condition (to be precise there is one always true) for the Error Injector, indeed it would not be reasonable to expect that the processes we are specifying would behave in a way so as to satisfy the needs of a fault model. Instead, EI is guaranteeing that it will only increment n' - it is the case of having only state update deletion (an increment deletes a decrement) as pointed out previously. Decided the EI behavior limitation (and thus decided the fault model) we can design our specification. From the EI specification C can rely on the fact that *n* and *count* will be never modified while *n'* will be only modified in a specific way (incremented). Now, with the addition of a layer in the program and in the specification we are still able to guarantee an (extended) desired behavior by means of the G_C condition, which says that *n'* will always be consistent with the value of *count* preserving the invariant $n' = n - count - 1$, i.e., the summation of *n'* and *count* will always be equal to $n - 1$. This will ensure that the postcondition $count = n \wedge \overline{n} = 0$ will hold at the end like in the case without interference. This simple example shows how the LFTS principles can provide a clear specification (with respect to a monolithic one) ensuring, at the same time, that a desired postcondition holds.

IV. The Automotive Example

The progress toward this way of layering specifications has been made by experimenting few case studies. For example, the one presented in [20] showed the power of the LFTS principle when applied to train systems. Instead, we now consider a simplified automotive case study. The Cruise Control is a system able to automatically control the rate of motion of a motor vehicle. The driver sets the speed and the system will take over the throttle of the car to maintain the same speed. One of the requirements of the cruise control is to be switched off when an error in the engine speed sensor is detected. This has to be taken into account in the specification. We use the CrCt to show how the idea of LFTS can be applied in (semi)realistic systems (simplifications of real system for the sake of experimenting with new ideas but still not mere toy examples). Let us consider the following ideal piece of CrCt code:

```
while (target <> current){
   delta := smooth(target, current);
   result := set_eng(delta);
}
```

The car speed is acquired in `smooth(target, current)` and then a delta is calculated for the car to have a smooth acceleration (smoothness has to be determined by experience). The specification of this code in term of P,Q,R,G is the following (it is expressed in natural language since we are not giving a mathematical model of the car here):

- P: target has to be in a given range
- Q: delta is zero and the driver has been comfortable with the acceleration
- R: the engine is adjusted (smoothly) according to delta
- G: the absolute value of delta is decreasing

The requirement mentioned above is not taken into account in this ideal piece of code, so in case the speed acquisition goes wrong the guarantee will not hold and the absolute value of delta will not be decreased. Indeed, following the LFTS principle we should organize it in two layers: a normal mode and an abnormal one (speed acquisition goes wrong):

```
while (target <> current){
   delta := smooth(target, current);
   result := set_eng(delta);
   if result <> OK then
      switch_off
}
```

This means adding a weaker layer of conditions for the “abnormal case” being still able to guarantee “something”. If speed acquisition goes wrong we do not want to force the engine following the delta since it would imply asking for more power when, for example, the car speed is actually decreasing (maybe an accident is happening or it is just out of fuel). Switching the engine off we avoid an expensive engine damage.

V. Concluding Remarks and Future Work

In this work, we provided a different view for interpreting problems and faults and we worked toward an improvement of the ideas presented in [17]. Our goal was to start an investigation leading to a method for the formal specification of systems that do not run in isolation but in the real, physical world. To accomplish the goal we passed trough a non trivial number of steps including the discussion in [20] of the concept of method itself (computer science has a proliferation of languages but very few methods). Then we presented how we intend to proceed to represent the static and the dynamic view of the problem. A section is dedicated to faults and the following to a case study.

Of course this work is not exhaustive and many aspects need more investigation. Especially the possibility of having Jackson's diagrams extensions working as a bridge between the static and the dynamic view in the way we described them. Although a small example of static and dynamic views is presented in this paper and a way to combine them idealized, more work is needed in combining them in a coherent and readable notation. Jackson' diagrams extensions are only one of the possible solutions anyway. Indeed another point we have just sketched here but that needs more work is about the the plug-ins and how to permit the practical use of different tools/notation. More investigation regarding the case studies is also needed.

Acknowledgments

This work has been funded by the EU FP7 DEPLOY Project (http://www.deploy-project.eu/).

References

[1] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[2] J.-R. Abrial, S. Schuman, and B. Meyer. *A Specification Language*. Cambridge University Press, New York, NY, USA, 1980.

[3] J. C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3):131–146, 2005.

[4] K. Bittner. *Use Case Modeling*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[5] D. Bjorner and C. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer, 1978.

[6] G. Booch. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[7] J. Coleman and C. Jones. Examples of how to determine the specifications of control systems. In A. R. M. Butler, C. Jones and E. Troubitsyna, editors, *Proceedings of the Workshop on Rigorous Engineering of Fault-Tolerant Systems (REFT 2005)*, pages 114–132, 2005.

[8] P. Collette and C. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In *Proof, Language, and Interaction*, pages 277–308, 2000.

[9] M. DeFleur, P. Kearney, and T. Plax. Mastering communication in contemporary america. 1993.

[10] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition*. Addison-Wesley Professional, 2003.

[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Communication of the ACM*, 26(1):53–56, 1983.

[12] M. Jackson. *Principles of Program Design*. Academic Press, Inc., Orlando, FL, USA, 1975.

[13] M. Jackson. *System Development*. Prentice-Hall, 1983.

[14] M. Jackson. *Problem frames: analyzing and structuring software development problems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[15] C. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[16] C. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[17] C. Jones, I. Hayes, and M. Jackson. Deriving specifications for systems that are connected to the physical world. In *Formal Methods and Hybrid Real-Time Systems*, pages 364–390, 2007.

[18] P. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1990.

[19] M. Mannion and B. Keepence. Smart requirements. *SIGSOFT Softw. Eng. Notes*, 1995.

[20] M. Mazzara. Deriving specifications of dependable systems: toward a method. In *Proceedings of the 12th European Workshop on Dependable Computing (EWDC 2009)*, 2009.

[21] M. Mazzara. Different perspectives for reasoning about problems and faults. Technical Report CS-TR No. 1151, School of Computing Science, University of Newcastle, April 2009.

[22] M. Mazzara and A. Bhattacharyya. On modelling and analysis of dynamic reconfiguration of dependable real-time systems. In *DEPEND, International Conference on Dependability*, 2010.

[23] G. Plotkin, C. Stirling, and M. Tofte, editors. *Proof, Language, and Interaction, Essays in Honour of Robin Milner*. The MIT Press, 2000.

[24] L. L. t. R. Descartes. *Discourse on Method and Meditations*. New York: The Liberal Arts Press, 1960.

[25] J. Reason. *Human Error*. Cambridge University Press, 1990.

[26] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 1997.

Failure Modes and Effect Analysis of Use Cases: A Structured Approach to Engineering Fault Tolerance Requirements

Elena Troubitsyna
Åbo Akademi University, Department of Computer Science
Joukhaisenkatu 3-5A, 20520, Turku, Finland
Elena.Troubitsyna@abo.fi

Abstract— **Fault tolerance – an ability of a system to cope with errors – is an important characteristic of dependable systems. However, software development approaches traditionally give precedence to modelling nominal system behaviour over modelling system behaviour in presence of faults. This leads to ad-hoc and error prone implementation of fault tolerance mechanisms. In this paper, we propose a systematic approach to elicitation and modelling of fault tolerance-related requirements. Our approach is based on using Failure Modes and Effect Analysis (FMEA) that is used to identify faults, their detection and error recovery. We rely on use-case modelling to structure system behaviour and propose to conduct FMEA of each individual use case. Our approach facilitates elicitation and structuring of fault tolerance behaviour. It enables an integrated modelling of nominal and abnormal system behaviour from early development phases.**

Keywords - use cases; failure modes and effect analysis (FMEA); fault tolerance; requirements

I. INTRODUCTION

The model-driven approaches to software development [1] usually represent system functionality in term of use cases. Use cases [2] describe system behaviour at different levels of abstraction. At the highest level of abstraction they depict the services that the system provides to its users. At the lower layers of abstraction, they describe functions of system components. Use-case modelling facilitates structuring complex requirements and serves as a basis for validating system design at the later stages of the development.

Traditionally modelling focuses on describing nominal system functionality. Yet, there are also many abnormal (exceptional) situations that arise during system execution. System dependability [3] can be jeopardized if such abnormal situations are not handled in a proper way, i.e., if fault tolerance mechanisms are implemented incorrectly. Although fault tolerance mechanisms constitute a significant part of software, they are often introduced at the implementation stage and in a rather ad-hoc fashion.

In this paper we propose an approach to conducing failure modes and effect analysis (FMEA) [4] over the use cases. FMEA is a widely used inductive safety analysis technique. We demonstrate how to apply FMEA to represent abnormal situations in use case execution. Our approach allows the designers systematically explore exceptional situations, identify their causes and error recovery strategy. We propose the patterns for conducting FMEA at different levels of abstraction and demonstrate how to incorporate the results of such an analysis into use case representation. The requirements obtained while conducting FMEA are systematically captured in the use case system model.

It is widely accepted that building in dependability and in particular, fault tolerance, early in the development process is more cost-effective and results in more robust design [3,4]. Our approach facilitates early consideration of fault tolerance in the design process. It allows the designers to uncover the additional requirements, which are needed to ensure fault tolerance. Moreover, it makes the process of requirements engineering more structured and hence improves requirements traceability.

The proposed approach is illustrated by a case study – modelling and analysis of an autonomous robot.

II. MODELING FAULT-TOLERANT SYSTEMS

The main goal of introducing fault tolerance is to design a system in such a way that faults of components do not result in a system failure [3,5,6]. A fault manifests itself as *error* – an incorrect system state [3,10]. Nowadays the main part of fault tolerance mechanisms are software implemented, i.e., software should detect errors and initiate error recovery. Error recovery is an attempt to restore a fault-free system state or at least preclude system failure. There are two types of error recovery: dynamic and fail-safe recovery. In the former case, upon detection of error software executes certain actions to restore a fault-free system states and then resumes normal system functioning without stopping the system. In contrast, fail-safe error recovery brings the system into a safe but non-operational state, i.e., executes system shut-down.

Initially the system is assumed to be fault free. Upon successful initialization, the system enters an automatic operating mode. While no error is detected, the system executes the normal control functions. Upon detection of an error software tries to execute error recovery and resumes normal function. If error is deemed to be fatal then software ceases its function and notifies the operator about it (e.g., by raising an alarm).

Use case modelling is a widely used technique for discovering and representing behavioural requirements of software-intensive systems [2]. A *use case* describes, without revealing the system implementation details, the system responsibilities and interactions with its environment while providing the requested services. A use case represents a distinct unit of interaction between an environment (human or machine) called an *actor* and the system. In general, the actors can be thought of as different stake holders that request the services to pursue their goals. Each use case describes a certain functionality to be designed. It can also include another use cases. Usually a use case contains several scenarios – the main scenario describing successful execution – and an alternative one describing various deviations.

There are ongoing debates on the principles of use case modelling. For instance, Fowler advices against breaking use cases into sub- use cases [7]. We argue that this approach is unsuitable for development of large-scale industrial systems. In this paper we adopt refinement-based approach to use case modelling. Namely, we propose to build the use case model gradually in a top-down manner.

Our initial model describes system functionality in terms of services delivered by the system in response to the service requests. The service requests are generated by the system environment represented by a certain actor. While designing a service, we should provide means for tolerating faults of various natures. In general, an execution of each service can fail. Hence each use case should contain means for fault tolerance. We propose to supplement each use case with an auxiliary use case defining a fault tolerance mechanism, which should be activated if an execution of the main use case fails. The initial use case diagram describes the basic use cases and supplements each of them with the auxiliary use cases to model error recovery. Observe that the auxiliary use cases confine all possible alternative actions to be undertaken for error recovery. In Fig. 1 we propose a general pattern for use-case modelling of a fault-tolerant system at the abstract level.

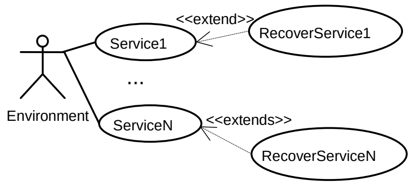


Figure1. Use case diagram of fault-tolerant system

Often at the abstract level of modelling the requirements describing the fault tolerance mechanism are yet to be discovered. However, even an abstract representation of them in the use case diagram shown in Fig. 1 enforces early consideration of fault tolerance aspect and facilitates elicitation of the requirements related to fault tolerance.

Usually a service provided by a system is a composition of certain subservices. In the use case modelling this can be depicted by decomposing the abstract use cases and refining the overall use case model. On the one hand, the refined use case diagram introduces the lower-layer use cases and defines relationships between the use cases on the higher and the lower layers. Such relationships are depicted via the stereotype `<<include>>`, since an execution of the upper-layer use case involves the execution of several lower-layer use cases. On the other hand, the refined use-case diagram specifies more precisely the fault tolerance mechanisms, which should be introduced to provide error recovery at each level of abstraction. Eventually we arrive at the use case diagram of the form presented in Fig. 2.

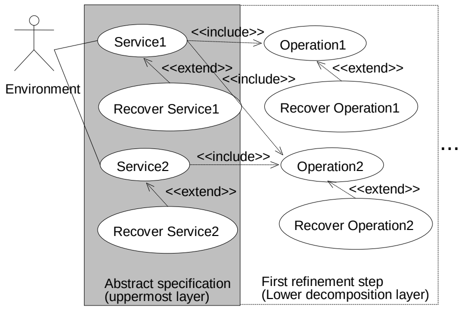


Figure 2. General pattern for final use case diagram

Observe that the use case diagram has the layered structure. The first layer encompasses the use cases describing functionality of the system from the environment perspective. They define the services, which the environment expects from the system. Each refinement step introduces the lower layers, which contain the use cases whose execution is required to provide the use cases at the upper layers. The decomposition is rendered via the `<<include>>` stereotype. The fault tolerance mechanisms are related with the corresponding use cases via the <<extend>> stereotype, as we discussed previously.

The occurrence of errors might prevent accomplishing the actor's goals. While designing a system it is important to ensure that each service request is acknowledged either by the successful result or by a meaningful error message. An error message might also contain the information that gives the actor a recommendation on how to achieve the desired goal by the other means.

In Fig. 3 we show the general template of use case description of a service as well as error handling use case. The general template can be applied to describe use cases at all levels of abstractions. It is easy to observe that fault tolerance mechanism has a hierarchical structure – failure of lower layer use cases are handled from higher-level recovery use cases.

It is easy to observe that completeness of fault tolerance requirements directly depends on whether our analysis of possible failures modes of use cases is exhaustive. To facilitate the analysis of possible failure modes of use cases, we propose to use Failure Mode and Effect Analysis (FMEA) [4]. It is a well-known inductive technique for eliciting failure modes of system components. Next we demonstrate how to apply this technique to facilitate discovery of failure modes of use cases.

Description of use case *Operation_Name*
Precondition When use case can be executed
Postcondition Normal result
Exceptional result
Includes Lower layer use cases
Normal sequence of events:

1. Check input parameters. If check fail execute use case *Recover_Operation*
2. Steps of use case. If a step includes invocation of lower layer use case then check the response. If check succeeds then proceed. Otherwise invoke use case *Recover_Operation*

Description of use case *Recover_Operation*
Precondition Failed input parameters or included use case execution
Postcondition Handling error
Extends: *Operation_Name*
Sequence of events:

1.Check invocation parameters. In case of input parameters failure, generate corresponding error and abort execution
2.In case of included use case failure apply appropriate recovery actions, e.g., retry, rollback, abort, reconfiguration. If recovery fails, generate corresponding error. Ensure that recovery actions are specified for each possible error.

Figure 3. Template of detailed use case description

III. INTEGRATING FMEA AND USE CASES

FMEA [4] is an inductive analysis method, which allows us to systematically study the causes of components faults, their effects and means to cope with these faults. FMEA is used to assess the effects of each failure mode of a component on the various functions of the system as well as to identify the failure modes significantly affecting dependability of the system. FMEA step-by-step selects the individual components of the system, identifies possible causes of each failure mode, assesses consequences and suggests remedial actions. The results of FMEA are usually represented in the tabular form that contains the following fields: component name, failure mode, possible cause, local effect, system effect, detection, and remedial action.

In this paper we propose to use FMEA to derive possible failure outcomes of each use case execution. To facilitate FMEA of use cases we introduce taxonomy of possible failure modes of use cases and outline corresponding detection procedures and remedial actions. Below we present the corresponding FMEA tables for the typical failure modes.

Use case	Use case name (uppermost layer)
Failure mode	Incorrect input parameters
Possible cause	Human or computational error
Local effects	Use case cannot be executed
System effect	Failure to execute requested service
Detection	Check value of input parameters before starting to execute use case
Remedial action	Abort service execution, return error message to environment

This failure mode represents an attempt to invoke a service with the incorrect input parameters. It is an unrecoverable error. While describing a use case, we should ensure that the returned erroneous service response identifies the causes of the failure.

Use case	Use case name (lower layer)
Failure mode	Incorrect input parameters
Possible cause	Computational error
Local effects	Use case cannot be executed
System effect	Failure to execute requested subservice
Detection	Check value of input parameters before starting to execute use case
Remedial action	Abort use case execution, suspend service provision, return error message to the service requester

This failure mode represents a failure to execute lower layer use case due to incorrect input parameters. Usually occurrence of such a failure would correspond to receiving an exception [8]. As an error recovery, the service requester should diagnose the cause of failure either by re-computing the input parameters or by propagating the exception further in the use case hierarchy.

Use case	Use case name (uppermost layer)
Failure mode	Incorrect service provision (wrong postcondition)
Possible cause	Computational error or unrecoverable error of subservices, or physical component failure
Local effects	Incorrect provision of service
System effect	Service is executed incorrectly
Detection	Check postcondition, generate error message, implement logging
Remedial action	Abort service execution, return error message to environment, halt system

The failure mode described above analyses an occurrence of a failure while executing a service. The failure might be caused by a failure of a lower layer use case or by a computational error at a higher level of abstraction. The diagnostic of such a failure aims at identifying its causes and deciding on the appropriate error recovery strategy.

The use case describing a similar type of failure of lower layer use case can be defined in the same way. In case the failure is transient, the error recovery by retry would possibly bring the system back to the normal state. In case the failure cannot be recovered, the error message is propagate to the upper layers of hierarchy.

The use case below describes service omission error. It is detected by the missed deadline. The failure mode of the use case on the lower layer is defined in the similar way.

Use case	Use case name (uppermost layer)
Failure mode	No response
Possible cause	Computational error or failure of lower layer use cases or communication failure
Local effects	Use case is not executed
System effect	No response on service requests within the deadline
Detection	Timeout
Remedial action	Execute system diagnostics. If no error is detected then resume normal operation otherwise halt system

Let us demonstrate the application of the proposed method to structure and model requirements of an autonomic robot.

IV. CASE STUDY

We illustrate use-case modelling of a fault-tolerant system by an example – *an autonomic robot.* The robot should move on a surface, i.e., in XY- directions and grab the objects located at certain positions. Via a radio-link a human operator sends the robot commands to move from one position to another, grab and release objects. The robot works autonomously. Such kind of robots are used in the environments that are hazardous for humans, e.g., to perform rescue operations. Since faults might prevent the robot from executing the requested service (that might lead to a failure of the rescue operation), the system has strict fault tolerance requirements.

The service-level use case model of the robot shown in Fig. 4 is very simple. It has two main use cases – move to the target coordinates and grab/release – and two auxiliary use cases to implement error recovery.

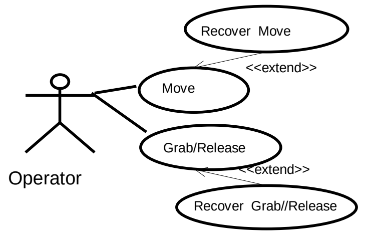


Figure 4. Service-level use-case model of the robot

Let us demonstrate how FMEA of the services-level use cases facilitates elicitation of fault tolerance requirements.

Use case	Move
Failure mode	No response
Possible cause	- Communication failure - Lack of mechanism to detect timeout of lower layer use cases - Computational error (non-termination)
Local effects	Use case is not executed
System effect	No response on service requests within the deadline
Detection	Timeout
Remedial action	- Retry execution of the use case. If execution succeeds then resume normal operation. - To diagnose communication failure send ping request. If no response then halt the system. - To ensure that execution terminates, set deadlines for execution of each lower layer use case. - To ensure termination guarantee termination of error recovery and proper handling of exceptions.

The example of FMEA allows us to identify important requirements, such as *introduce timers, ensure termination of error recovery and additional functionality required to implement diagnostics of communication failure.* Moreover, the system design should also ensure that *upon completing each service, the success or failure of the execution is checked.*

The service-level use case diagram is further refined to model the details of use-case implementation as shown in Fig. 5. Each service is decomposed into the lower layer use cases, which should be executed to implement it.

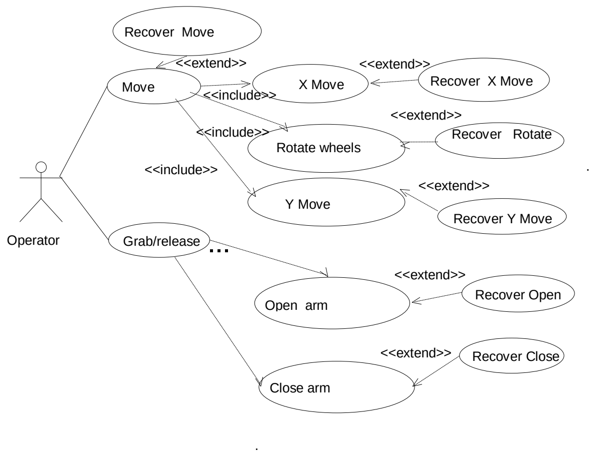


Figure 5. Refinement of use-case diagram of the robot

The diagram is created according to the pattern proposed in Fig. 2. The main use cases are decomposed into the lower layer use cases, which should be executed to implement them. At this refinement step, a more detailed description of error recovery can be given as well. We add the details of the possible causes of faults, detection and error recovery into our FMEA analysis. This allows us to arrive at the detailed description of use cases. Below we present a detailed description of the *Move* use case, which contains description of the possible errors and corresponding error recovery procedures derived from FMEA results.

The precondition might fail, if the operator inputs wrong parameters. In the use case *Recover Move* we model the notification of the operator and shut down. The execution of *Move* essentially consists of executing the included lower-layer use cases *Move to X position*, *Rotate Wheels* and *Move to Y position*. The recovery from failures is done by executing *Recover Move* with the parameters corresponding to the names of the failed use case.

At the final refinement step the details of executing the lowest-layer use cases and failure modes of the components, involved in the execution become available. This allows us to establish the exact causes of failures of the lowest-layer use cases and precisely define the required remedial action precisely. We omit presenting the detailed description of the lower layer use cases. The use case diagram obtained at the final step adheres to the pattern presented in Fig. 4. The detailed description of functional requirements and the requirements related to fault tolerance are obtained at this stage.

In this section we have illustrated the evolution of use-case model of a fault-tolerant system. We emphasized reasoning about fault tolerance at each refinement step.

In general the proposed approach can be seen as a generalization of Randell's recovery block approach [6]

Use case Move

Brief description This use case defines system reaction on the operator's command **"move to XY coordinate"**. It includes activating motor, rotating wheels, reading positioning sensors, reporting success or failure of the execution

Includes use cases **"***X Move", "Y Move", "Rotate Wheels"*

Extends use case *"Recover Move"*

Preconditions Operator requests service **Move to X,Y position,** the system is fault free

Postconditions The robot reaches the requested position before the deadline and success is reported. Otherwise failure is reported

Normal sequence of events

1. Verify that *X,Y* are valid coordinates. If the verification fails then **A_Failure1** in recovery sequence, else calculate the distance along the X direction and Y direction.
2. Execute the use cases *"Move to X position"*
3. If the execution of the use cases *"Move to X position"*, failed then **A_Failure 2** in recovery sequence, else proceed to execution of the use cases *"Rotate Wheels"*
4. Execute *"Rotate Wheels"*. If execution of *"Rotate Wheels"* failed then **A_Failure 3** otherwise proceed to execution of *"Move to Y position"*
5. If the use case *"Move to Y position"* failed then **A_Failure 4** in recovery sequence, else if execution of the use case succeeded then report the success of the service execution
5. **…**

Recovery sequence of events

A_ Failure1: Execute the use case *"Recover Move"* with the parameter "incorrect input parameters**".** Shut-down the system. Notify the operator

A_Failure2: If the use case *"Move to X position"* has failed then execute the use case *"Recover Aspirate"* with the parameter *"Move to X position"*. If the use case *"Move to Y position"* has failed then execute the use case *"Recover Aspirate"* with the parameter *"Move to Y position"*.

…

and its translation into use-case modelling. Indeed, a recovery block describes how the operator can achieve a desired goal in the normal as well as erroneous situations. The recovery block has a flat structure, i.e., the operations required to achieve the desired goal reside on the same level of abstraction. Complexity of modern systems requires an encapsulation of the low-level operations and hence, imposes the hierarchical style of system structuring. In our approach we generalized the recovery block mechanism by introducing the hierarchy of use cases defining how the desired goal can be achieved.

V. CONCLUSIONS

In this paper we demonstrated how to structure complex requirements by FMEA and refinement of use cases. We propose a pattern for use-case model of fault-tolerant systems and demonstrated how to structure the description of use cases to capture the fault tolerance aspect. The pattern aims at enforcing early consideration of fault tolerance in the design process. It allows the designers to uncover the additional requirements, which are needed to ensure fault tolerance. Moreover, the proposed pattern makes the process of requirements engineering more structured and hence improves requirements traceability.

Among the pioneering works on addressing dependability in UML modelling is research by Alexander on misuse cases [1]. He proposed to consider use cases with hostile intent to facilitate discovery of dependability-related requirements. While the developers of safety- and security-critical systems are familiar with misuse-case modelling, the developers of non-critical systems traditionally rely on use-case modelling. Often safety or security implications are unknown in the beginning of the system development and uncovered later in the development process. As a result, the development of a critical system is conducted in a traditional style. Our approach aims at addressing this problem by enabling dependability consideration in the traditional UML modelling.

Alenby and Kelly [9] studied use-case modelling as a tool for discovering safety-related requirements. Our approach is similar to their work, since the new requirements can be discovered as well. However, our main focus was to study how to systematically capture requirements by conducting FMEA.

Kelly and Weawer [10] proposed an extension of goal structuring notation (GSN) to support safety argument. They demonstrated how GSN can facilitate safety assurance via structuring safety case. Our approach employs the similar idea of decomposing the high-level goals into the low-level sub-goals. However, we focused on the development process rather than on description of safety cases.

Jurjens focused on algebraic formalization of various UML artefacts to reason about safety [11]. However, his work leaves aside the problem of capturing dependability-related requirements in the development process.

Use cases have been formalized as contracts by Back et al. [12]. However, this work does not consider dependability aspects.

Hassan et al. [13] proposed a methodology that enables architectural-level analysis of safety using a combination of safety techniques. Our approach provides a support for early development stages and hence can be considered as complementary to [13]. The opposite approach – deriving FMEA from UML models has been explored by David et al [14]. The similar issues have been studied by Mazzara is the problem frames [15]. The use of FMEA at different stages of UML-based development has been explored by Hecht et al [16] and Wentao [17].

In our future work we are planning to explore further various fault tolerance mechanisms and their modelling in UML.

REFERENCES

[1] I. Alexander. Misuse cases: Use Cases with Hostile Intent. *IEEE Software, vol.20 (1),* pp. 58-66, 2003.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. “*The Unifying Modeling Language User Guide*”. Addison-Wesley, 1999.

[3] J.-C. Laprie. Dependability: Basic Concepts and Terminology. Springer-Verlag, Vienna, 1991.

[4] N. Storey. *Safety-critical computer systems*. Addison-Wesley, 1996.

[5] T. Anderson and P.A. Lee. *Fault Tolerance: Principles and Practice.* Dependable Computing and Fault-Tolerant Systems, Vol 3**.** Springer Verlag; 1990.

[6] B. Randell and J. Xu, The Evolution of the Recovery Block Concept. In M. Lyu (ed.) *Software Fault Tolerance*. Wiley 1994.

[7] M.Fowler. UML Distilled: A brief Guide to the Standard Object Modelling Language. Addison-Wesley, 2004.

[8] F. Cristian. Exception Handling. In T. Anderson (ed.): *Dependability of Resilient Computers*. BSP Professional Books, 1989.

[9] K. Allenby, T. P. Kelly. Deriving Safety Requirements using Scenarios. In *Proc. of the 5th IEEE International Symposium on Requirements Engineering (RE'01),* Toronto, Canada, pp.228-235, 2001.

[10] T. Kelly and R. Weaver. The goal Structuring Notation – A Safety Argument Notation. In Proc. of *The Dependable Systems and Networks 2004 Workshop on Assurance Cases,* July 2004.

[11] J. Jürjens. Developing safety-critical systems with UML. In *Proc. of UML’2003***.** Lecture Notes in Computer Science, San Francisco, USA, October 2003, pp**.**360 – 372.

[12] R.-J. Back, L. Petre and I. Porres. *Analysing UML Use Cases as Contracts.* In Proceedings of UML'99. Fort Collins, Colorado, USA, October 1999. Lecture Notes in Computer Science 1723, pp. 518-533, Springer-Verlag.

[13] A.Hassan, K. Goseva-Popstojanova, K and H. Ammar. UML based severity analysis methodology. In Proc. of Reliability and Maintainability Symposium. Computer Press, 2005

[14] P. David, V. Idasiak & F. Kratz. Towards a better interaction between design and dependability analysis: FMEA derived from UML/SysML models. In Proc of ESREL 2008 and 17th SRA-EUROPE, Spain, 2008.

[15] M. Mazzara, Deriving Specifications of Dependable Systems: toward a method. CS-TR 1152 –Technical report Newcastle University, May 2009.

[16] H.Hecht, X.An and M.Hecht. Computer-Aided Software FMEA. In Proc. of RAMS 2004, Computer Press, 2004.

[17] W.Wentao and Z.Hong. FMEA for UML-Based Software. In Proc. of World Congress on Software Engineering, Computer Press, 2009.

Timing Failures Caused by Resource Starvation in Virtual Machines

Sune Jakobsson
NTNU, ITEM
Trondheim, Norway
e-mail: sune.jakobsson@telenor.com

***Abstract*—This paper discusses cascading effects of resource starvation in virtual machines, and how that affects end-user experiences in certain cases. The paper presents the occurring issues on an N-tier server system, and the way the starvation causes unexpected delays in a service for an end-user. The initial observations were on unexpected communication delays, and by using a simplified test system, this behaviour can be confirmed. The delays can traced back to memory management in the individual servers, causing the timing failures.**

Keywords- Java virtual machines; garbage collection; application servers; resource starvation.

I. Introduction

This paper addresses cascading effects of resource starvation, in particular where a server side system is built using multiple tiers, and they call each other in a serial fashion to a depth of N. This effect is observed by end-users intermittently, where their service fails once, and by reloading their browser or application the service is restored. The definition of timing failures is from [3] and is defined as: “The time of arrival or the duration of the information delivered at the service interface (i.e., the timing of service delivery) deviates from implementing the system function.” A Cascading Effect is an unforeseen chain of events due to an act affecting a system [9].

A typical service would consist of a client running in a terminal, accessing a server frontend exposed on the internet. The underlying system is often a multi-tier system consisting of one or more application servers and one or more databases. The frontend, does load balancing, and then passes the request to a HTTP server, which in turn forwards the request to a Servlet container. The Servlet processes the requests, and in turn will contact other servers on the Internet, databases, etc. Once all the information is returned the Servlet builds the response page to the requesting user, and the information is returned.

Each server uses a dynamic amount of memory for their task, and with modern programming languages the memory is allocated when needed and freed when the virtual machine is running low on the free memory pool, or is idle and decides to clean up its memory pool [6]. This process is referred to as garbage collection, and there are many strategies for this mechanism [7]. If one observes the amount of free memory on a virtual machine over time, the waveform is an inverse saw tooth form with a maximum value matching the total amount of free memory, and the minimum values when the garbage collection mechanism is run.

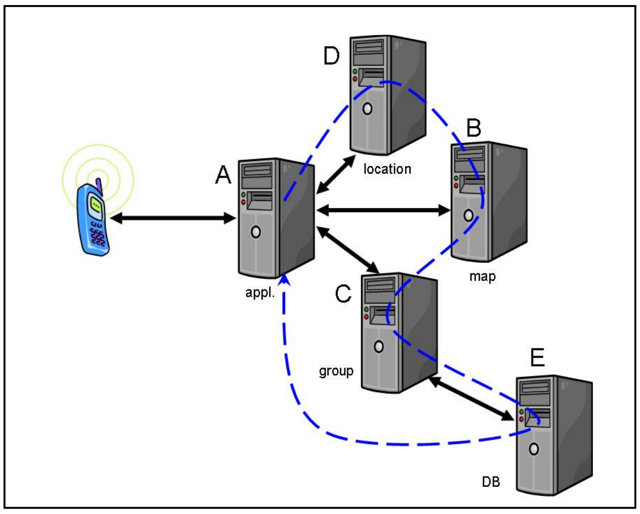


Figure 1. A server system.

Fig. 1 shows how a mobile client interacts with an application server, marked “A”, and how this server in turn interacts with the other servers, the blue dotted line is the path used when checking the availability. The figure also gives an indication on how the servers are deployed, but they might belong to different administrative domains on different networks. Fig. 2 is a sequence diagram, showing how the HTTP invocations in the test system are chained together, when they are called form each server to the next server, starting from server A, until they reach the depth of N servers, where the result is returned. It is assumed that the servers are instances of application servers like Tomcat [5].

II. Memory Allocation

Applications need memory for their task and the communication requires buffers to store data. When new objects are allocated they are taken from the memory pool. When the available memory runs low, a garbage collector inspects and frees objects that are no longer in use, and if one observes the available free memory on a typical virtual machine this shows an inverse staircase pattern. Fig. 3 shows a set of available free-memory patterns, showing how they are allocated and garbage collected over time. The amount of free memory available is collected from the JVM, using system calls on each application server. The figures are normalized, so that they can be compared, and the x-axis shows a cycle time of approximately 9 minutes for the servers in blue and brown, whereas one of the servers shown in purple has a cycle time of 23 minutes. The frequencies of the cycles depend on the load of each individual virtual machine. The load impacts the memory usage, and when observing the graphs they show a frequency modulated inverse saw tooth shape.

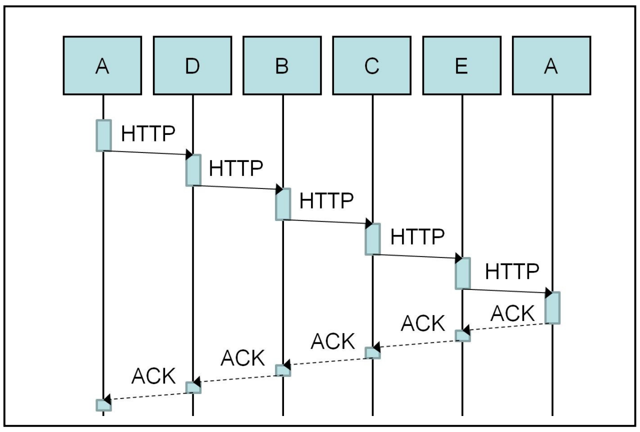


Figure 2. Cascaded HTTP invocations.

From a dependability point of view, the interesting point in time is when the garbage collection is running and hence the requests for free memory are stalled. Since the frequency this occurs at does not happened at regular intervals, and one single run of garbage collection does not impact the overall service execution time. The cascading effect occurs when multiple garbage collectors run right after each other effectively stalling the ability to reserve resources in the virtual machines involved. This condition is a late timing failure as shown in Fig. 8 in [3]. For each virtual machine the probability that the garbage collector runs is p_x. If we number the probabilities p_1 to p_n the interesting scenario occurs when multiple garbage collectors run after each other in a close sequence, effectively stalling the response back to the end-user. For all garbage collectors to run the probability is the product of the individual probability, and for all but one, the probability is sum of the individual probabilities but one, and so on. Given that the number N is small, one can construct a formula and find the N with the biggest likelihood for performance failure.

III. A TEST SYSTEM

To model a real service that would exercise as many parts as possible of an N-tier system, a simple test system has been programmed, which uses the involved virtual machines, and invokes each other into N levels with using a fixed amount of memory in each level. This test system therefore mimics the behaviour of a hypothetical service including a number of servers interacting, communicating over the Internet to provide a composite service. The core issue of application availability is trying to establish a method that ensures that the individual nodes are able to communicate, and that their application servers are functional and available, when they interact as shown in Fig. 2. A typical service would run in an application, and retrieve miscellaneous data from other servers connected to the internet, like group information, location, and maps.

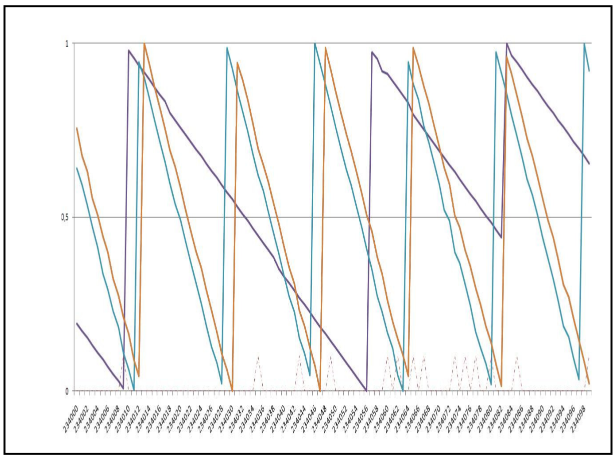


Figure 3. Plotted set of free memory samples (normalized).

Each node communicates with the next node defined by a data object consisting of a fixed list of URL's that is passed between all servers. As part of the local logging process of each node they also collect the amount of free memory available in the virtual machine, and the data object also contains timestamps for later correlation of the results. The list is continually passed around at fixed intervals (30 seconds), and when there are delays or processing issues this impacts the amount of free available memory. This logged data on each node can then be post-analyzed and the real cause can be determined, for the failure of the hypothetical service. The interesting part here is the strong correlation on how many buffers are occupied due to delays or errors in the transmission between the participating nodes. When there are HTTP messages that are not acknowledged, they use up the common memory of the system, and this can be measured with the available free memory. The amount of free memory can be obtained directly from a Java virtual machine if the application is implemented in the Java programming language or from the operations system when other programming languages are used. The elegance of this approach is that the measurements are non-intrusive to the application, and eliminates other hooks into the communication channel or their respective drivers.

IV. OPERATION

Each node, when invoked, obtains a time reference and the amount of free memory, and when the communication is done with the next node, these values are written to the standard output file of the application server. At the sending end there is a standalone program issuing the requests at fixed intervals as shown in Fig. 1, from server marked “A”.

If the transmitted URL list makes it all the way to the destination the sender receives an acknowledgment in the form of HTTP status (200) OK, but there are plenty of observations where the list is delayed or its acknowledgment is delayed, but not lost. If the list is lost that is an obvious case, and also indicated in the HTTP status from the underlying TCP implementation in the operating system, but there is a group of cases where the list is not lost and makes it all the way to the last node, and the acknowledgment is returned, and all seems fine, yet the time seen from the initiating node is unacceptable long, and we have a case of timing failure. The term “unacceptable long” varies from case to case, but if a human does the interaction a delay for a second on a response from a service might be unacceptable.

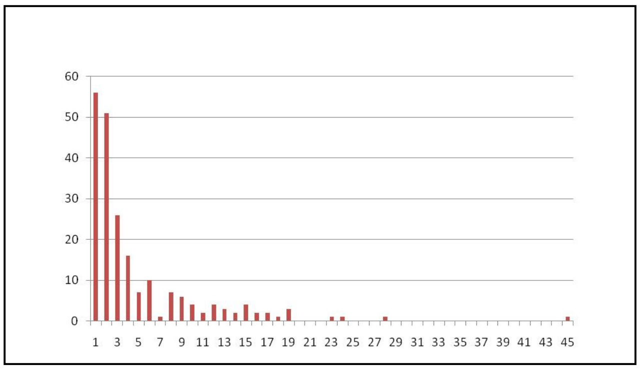


Figure 4. Time between timing failures.

By post-processing the log files from each node, more details about the cause can be obtained. The graph in Fig. 4 shows the time between failures for a period of a month, where only a few other errors occurs, where the data object fails on its route, and in this period only timing failures occurred. The steps on the x-axis are 3000 seconds each. The number samples with timing failures is 212 out of totally 80624 data objects dispatched. In this period there are also 15 cases of halt failures in the data set. The shape of the graph indicates that it resembles a Poisson distribution, with a λ of approximately 0.955. This fits well with the assumption that the events occur continuously and independently at a constant average rate.

V. State of the Art

In closed and well monitored systems, with probes and other means of surveillance there are many commercial available solutions to detect timing failures and other failures and faults. However when it comes to distributed systems across different domains where there is no common administration, there is little material available. Several authors have studied causes of catastrophic failures in Web Applications [8] and the failures impacts on operation and how the failure has impacted the companies’ respective brand. Porter defines a system called X-Trace [1], to collect trace data to figure out what went wrong in an Internet scale system. This is done by adding extensions to HTTP headers in the requests, in order to be able to trace or locate them afterwards. His approach has some scaling issues, and also requires insertion of monitoring nodes or additional SW on the servers.

In the approach we propose, one would add the proposed minimalistic test application on each node one has control over and call the other nodes with some dummy data, in order to decide if the communication and the application servers are indeed available.

VI. Conclusion

This paper showed the cascading effect of the individual memory allocation processes, and how they do affect the performance and availability for a service using multiple serves loosely connected over the Internet. The data collected also supports the assumption that the occurrence of timing failures occur continuously and independently at a constant rate. This paper outlines one of the issues observed on practical data collected in my research work, and will be further validated and modelled in my thesis work.

Acknowledgment

I would like to thank Professors Rolv Bræk and Bjarne Helvik at Department of Telecommunication at NTNU, for their advice and guidance in my research work.

References

[1] G. Porter, “Improving Distributed Application Reliability with End-to-End Datapath Tracing”, PhD at Electrical Engineering and Computer Sciences, University of California at Berkeley Technical Report No. UCB/EECS-2008-68 http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-68.html (last seen Jul. 2011)

[2] W. G. Bouricius, W. C. Carter and P. R. Schneider, “Reliability Modelling Techniques for Self-Repairing Computer Systems” Proceedings 24th National Conference ACM, 1969.

[3] A. Avižienis , J. Laprie, B. Randell, and C. Landwehr, “Basic Concepts and Taxonomy of Dependable and Secure Computing,” IEEE Trans. Dependable and Secure Computing, vol. 1,no. 1,pp. 11-33, Jan.-Mar. 2004.

[4] J. Engel: “Programming for the Java Virtual Machine”, Addison-Wesley, 1999. ISBN 0-201-30972-6

[5] Tomcat application server. http://tomcat.apache.org/ (last seen Jul. 2011)

[6] Java virtual machine. http://java.sun.com/javase/ (last seen Jul. 2011)

[7] R. Jones, “The Garbage Collection Page”, http://www.cs.kent.ac.uk/people/staff/rej/gc.html (last seen Jul. 2011)

[8] S. Pertet and P. Narasimhan, “Causes of Failure in Web Applications (CMU-PDL-05-109)”. Parallel Data Laboratory. Paper 48. http://repository.cmu.edu./pdl/48 (last seen 2011)

[9] Cascading effect: http://en.wikipedia.org/wiki/Cascade_effect (last seen 2011)

Article in conference proceedings:

[10] S. Jakobsson, “A Token Based Approach Detecting Downtime in Distributed Application Servers or Network Elements”, Networked Services and Applications - Engineering, Control and Management, 16th EUNICE/IFIP WG 6.6 Workshop, EUNICE 2010, Trondheim, Norway, June 28-30, 2010. ISBN 978-3-642-13970-3 Proceedings, pp. 209-216

Video Surveillance in the Cloud: Dependability Analysis

Aleksandra Karimaa
Turku Centre for Computer Science, Teleste Corp.
Turku, Finland
alkari@utu.fi

Abstract—**Cloud computing with storage virtualization and new service-oriented architecture brings a new perspective to the aspect of dependability of video surveillance solutions and other safety-critical applications. The existing research is focused mainly on security challenges of cloud applications in general. The area of safety-critical systems is relatively unexplored especially beyond aspects of system security. We believe the overview of system dependability shall be done to cover other dependability attributes. It shall bring new arguments for expansion of video surveillance systems towards cloud technology, global resources distribution and virtualization. The article reviews the main drivers towards surveillance in cloud technology. We research the dependability characteristics in context of transition of video surveillance architecture towards cloud solutions. Finally, we propose the areas of focus for system development and design.**

Keywords-surveillance; cloud; dependability

Introduction and motivation

The main motivations driving video surveillance towards cloud computing is the scalability of computing and storage resources which provide; cost effective scalability, flexibility of resource management and improve system performance.

The transition of video surveillance towards cloud solutions can be seen rather as a continuous process than a disruptive innovation. There are multiple factors which indicate that the development towards a cloud solution is a natural evolution for the development of surveillance systems.

The architecture of video surveillance systems develops towards a model that includes dumb clients and a core of servers - it is identical with the architectural principles of cloud computing. Moreover, the systems themselves are becoming more often distributed, creating the structure of multiple local sites connected in mesh-like structures. The connectivity between single locations is based upon IP (Internet Protocol). Additionally, there are ongoing processes of standardization for architectural solutions and external interfaces (refer to ONVIF [1], PSIA [2], and Web Services architecture standardization by W3C [3]). In the case of surveillance systems, the process is driven by the requirement for multi-vendor integration with open and well defined interface environment enabling a cloud-based architectural solution. An open and standardized environment motivates the development of other value-added features (such as quality or security) which can then be offered as a product differentiator driving market maturity. Addressing the current weak points of dependability should have a positive impact on a number of cloud-based safety-critical systems. Finally, typical video surveillance systems have high demands for massive storage requirements (recording of video streams) and high-performance computing (coding the streams and system intelligence) which are major advantages of cloud and virtualization technologies.

However, system transition towards cloud solutions is not without its challenges. In order to benefit from a cloud-based system, there is certain amount of system and software development required. The systems should be able to accommodate automated mechanisms available in the cloud infrastructure, utilize advantages of cloud architecture and also handle cloud architecture limitations. A good example is the video format conversions requirement which accommodates available service models (cost of storage at data centre and the cost of network transmission). Another example includes extension of system failover mechanisms to accommodate virtual machine failover availability.

One of the problems with this transition towards a cloud-based system might be the flexibility of the offering. A cloud service offering may be focused on supporting the scenario where the video transmission happens from the cloud to the user whereas in surveillance system the video is transmitted from the user or camera towards the cloud (for storage purposes).

This article reviews the transition alternatives from a traditional to a cloud-based surveillance system. An overview of dependability objectives for a surveillance system is also described. Next, the objectives of dependability objectives: availability, security, reliability and maintainability are analyzed. The article is closed by a discussion chapter containing a short summary of the topic for the transition process.

Transition alternatives

The process of transition of a surveillance solution towards a cloud-based system is expected to be quite complex, this is due to security concerns of cloud-based systems and the immaturity of the current market offering of cloud services. Therefore, it is expected that the transition will be gradual and some of scenarios will be more attractive than others.

Hardware virtualization provides an interesting alternative for being a first step for the transition to cloud technologies. It changes traditional relationship between software and hardware – software is no longer dependent on hardware location. One application can run in multiple locations and many applications can share the same hardware. Hardware virtualization increases resource utilization and efficiency, as well as lowers capital investment and maintenance cost. Some level of hardware virtualization is well presented even for popular desktop environments. More advanced options can be introduced by Private Clouds and Public Cloud as part of IaaS (Infrastructure as a Service) services.

Private Enterprise Clouds are an especially attractive scenario as they offer cost efficiency while maintaining traditional level of security. A Private Cloud is a pool of resources available for sharing within a given private or enterprise entity. In the simplest form, it is dedicated storage or computation hardware with a virtualization layer allowing for the management of multiple virtual units under the same physical unit. Private clouds offer effective resource sharing which provides cost efficient and a scalable alternative for dedicated recording hardware. In a situation where the recoding hardware represents a significant part of surveillance deployment cost, this solution may offer a cost effective solution. Private Clouds offer advantages, such as reliability, performance and without (typical for cloud environments) concerns, such as level of security, management of hosting (especially sharing multiple customer installations on the same physical resources). Additionally, a Private Cloud scenario might be encouraged from a business perspective to provide an adaptation period for familiarization with cloud technology tools and processes. Another argument towards Private cloud computing being first step of the transition to cloud computing is the cost of software adaptation which is relatively low. The basic level of such adaptation should ensure that the system is able to accommodate distributed nature of system storage.

However, the scalability of the Private Cloud might be limited in comparison to Public Cloud..

The choice between Public and Private Cloud computing should be evaluated separately for each part of the surveillance system. Typically, security-critical resources such as user access database, incident video recordings or audio information will not be placed in a Public cloud. However, public camera video recordings or computational resources could be subjects of "full cloud conversion" under public domains.

Dependability analysis for surveillance transition towards cloud

The dependability is one of main properties characterizing the system, next to functionality, performance and cost [4].

There are multiple definitions of dependability. Dependability of a system can be described by the ability to deliver a service that can be trusted and where potential service failures not frequent [4]. ISO definition is focused on availability. IEC definition of dependability combines availability with reliability. In case of safety-critical applications, the safety and security attributes, especially confidentiality and integrity shall be addressed where evaluating system dependability. Additionally, the system maintainability shall be analyzed providing complete overview of dependability of the systems based on a cloud design. Summarizing, the following attributes of dependability shall be analyzed for video surveillance systems: availability, reliability, security (confidentiality, integrity) and maintainability.

A. Availability

Availability of surveillance system is focused on ensuring service continuity by providing access to the system and its resources. Availability and disaster recovery is an essential value of all security-critical systems. Lack of access to system resources and inability to react, investigate and record the incidents are probably the largest risks for surveillance system dependability.

High-end distributed video surveillance systems support high- availability is already implemented providing availability near 99.999 % (five nines availability) of system uptime. The mechanisms to ensure this high level of availability include: keep alive communication, automatic failover for backup devices, software and hardware watchdogs, life cycle management and resource management programs, failover, redundancy and reliability support in the architecture, but also services offering (to provide fast reaction times in case of system problems).

Cloud solutions can offer improvement for availability especially for two types of systems: local small installations and geographically distributed systems. In the case of local, one-box type installations, the offering of cloud services can be used to introduce redundancy mechanisms and disaster recovery tools – before not available for this class of system. It is worth to underline that in case of these local small systems improved availability and virtual accessibility opens new business opportunities, for example outsourcing of business processes which might be a major advantage for small businesses. In case of distributed systems where topology consists of multiple interconnected entities clouds improves availability in a cost efficient way by introducing common redundancy and backup resources and tools. Also, the distributed nature of the cloud itself improves availability – a system hosted in single location is more vulnerable.

A cloud-based offering provides improved system redundancy and a new range of failover mechanisms, for example; the IaaS model disaster recovery service providing a cost competitive alternative to an internal system (both software and hardware) with built-in solutions traditionally relying on watchdog-like applications. The process of migration of a surveillance system is likely to require development to accommodate cloud internal mechanisms for failover and availability and to build-in failover and redundancy mechanisms between hardware and software.

The definition of availability can be extended further to guarantee access to system resources that are provided only for entitled users (unauthorized access is denied). A Cloud provider's knowledge, awareness and best practices greatly contribute to the level the availability by providing protection of the system infrastructure against low level DoS attacks whilst providing tools and services to protect the resources from being unavailable or corrupted.

Despite the fact that a level of resource availability will be guaranteed by an IaaS cloud service provider (example for Denial of Service attacks) the development related to the improvement of system availability is one of the major tasks for surveillance transition toward cloud computing. There is a significant amount of system development required for the improvement of the authentication, authorization and accounting (AAA) mechanisms as traditionally, these rely partially on high security level of physical access. Special attention shall be paid to improving identity verification which, originally being part of accounting now plays a critical role in security mechanisms applied for communication between the cloud and the rest of the system.

B. Security

Security of a cloud solution is of major concern in the context of a safety-critical system, mainly due to the fact of virtualization; the data no longer resides on dedicated hardware on location that is easy to identify. System security vulnerabilities, such as weak passwords, inefficient virus protection, unauthorized use of access devices or too flexible access rights exist and should be addressed in both traditional and cloud environments.

Security is not defined as single attribute of dependability by Avižienis et al. [4] but it as composite notion of other dependability attributes. Similar approach has been described by Krutz and Vines [6] where security is the concurrent existence of confidentiality, and integrity when availability is ensured.

Confidentiality guarantees the absence of unauthorized disclosure of system resources and other relevant information. Confidentiality is traditionally provided by elements, such as: network security protocols, network authentication services and data encryption protocols. Confidentiality in a cloud system is focused on the confidentiality of transferred data with use of these elements. This means that in the case of a safety-critical system based on cloud architecture there should be a clear security policy in place, defining the exchange of data. It should define entities authorized for access and exchange; the data itself shall be categorized as confidential, sensitive, private, and public to specify the level of protection to be applied. Confidentiality of safety-critical systems based on cloud computing shall be focused on addressing authorization (including identity establishment), access control, rights managements, and encryption requirements (mechanisms and architectural solutions). Identity establishment and management play an important role as being a critical part of process of securing the communication channels between a system and the cloud infrastructure. Cloud-based safety-critical systems shall have at least two factor authentication where type 1 authentication is “something you know” (such as password), type 2 is “something you have” (such as smart card) and type 3 is “something you are“(such as fingerprint). The above suggest the popularity of bio-identification will increase and is being demanded by the safety-critical application market. Additionally, cloud-based surveillance systems shall introduce a public key infrastructure and encryption key management whilst implementing digital certifications and all related issues such as: handling certificate revocations lists, key management, distribution, revocation, recovery, renewal, and destruction. Additional mechanisms can be applied to secure system-to-cloud communication channels, including layered security, segmentation of virtual local area networks and applications, clustering of DNS (Domain Name System) servers for fault tolerance, load balancing and firewalls.

Integrity defines the absence of improper system alterations. The cloud system should ensure the integrity of data during transfer and storage. The system should be able to detect and/or correct data errors and alterations whilst identifying the origin of the data and its accuracy. The integrity of the provided solution shall rely on access control and rights management. It shall be stressed that the integrity of the data shall be secured ‘end-to-end’ including the means of data export. The subject of data integrity is extremely important in the case of video surveillance solutions providing evidence export, reporting and auditing functionalities.

Video surveillance systems, especially the ones with focus on production and delivery of evidence, usually have data integrity mechanisms implemented: they include RAID (Redundant Array of Independent Disks) technologies (to maintain the integrity on the hardware level), file checksums, etc. Data integrity on the physical level can be easily maintained by cloud technology and it is typically part of cloud service offering. Higher level of cloud service offerings, such as PaaS (Platform as a Service) or SaaS (Software as a Service) can also offer file level integrity tools even for exported material. It offers great advantages in terms of flexibility. Different integrity mechanisms can be provided for different surveillance system owners taking into account their different needs and legal considerations, without any internal development required, including: firewall services, communication security management services and intrusion detection services.

C. Reliability

Reliability is focused on service continuity by defining the mechanisms of fault prevention, tolerance (avoidance) to deliver trusted service and removal (reducing) and fault consequence forecasting to define and meet required system dependability specifications.

A Cloud offering provides improved reliability in the form of services and infrastructure. Reliability oriented cloud services include for example; automatic backup and redundancy services, incident response services and safe failover mechanisms. These services can provide the required reliability level without a large investment in capital and human resources, for example; incident response services typically provide analysis of event notification, response, escalation procedures, post-event follow up and incident response management (including for example risk mitigation planning). This type of service offering is very important for large scale applications; the scalability of the offering also provides cost efficiency for small installations where similar services were not previously available.

Automatic backup and redundancy services are provided by the architecture of cloud infrastructure and its distributed nature. It eliminates the need of expensive backup hardware, software and locations providing resources (such as SAN storage areas or computational resources) on demand. Koslovsi et al. [5] provides an overview of reliability advantages brought by cloud infrastructure virtualization which enables transparent and customized reliability provisioning.

It should be underlined that in order to utilize reliability improvements available in cloud technologies the system should meet specific requirements, which include secure access from remote locations and towards cloud, truly distributed architecture where the available mechanisms are independent from location, discipline in traffic monitoring, management and other security mechanisms and also associate processes to be in place.

D. Maintainability

Maintainability is the ability to undergo, modifications, and repairs without the need to disable access to the system. The advantage of the transition of video surveillance systems to the cloud depends in great on the service or infrastructure provider. The providers of the cloud infrastructure shall be carefully evaluated for their ability to maintain the agreed terms through service lifecycle starting from adoption phase. However, the current cloud offering is mature enough to provide full range of services assisting the system customer from early phases of the system planning, through phase of deployment, maintenance and ending on systems disposal services (for example to guarantee correct disposal of system information). It is a considerable improvement compared to traditional systems where quite often the availability of resources to deploy surveillance systems project depends on maintenance demands for the existing installation. Therefore, the transition to cloud opens new opportunities in terms of business models for companies providing such systems.

Conclusions

The transition of video surveillance into a cloud service can offer great advantages on system dependability only if the challenges of the transition are known and addressed. Cloud technologies offer different service models of cloud – IaaS (Infrastructure as a Service), PaaS (Platform as a Service) or SaaS (Software as a Service). The cloud infrastructure of the IaaS-based model seems to be the most suitable for the first step of the transition of video surveillance by providing advantages of hardware virtualization, cost scalability and performance. The transition shall be a continuous process. The plan of gradual transition into cloud computing shall be investigated for each system- external system functionalities such as video content analysis modules could be a good candidate for the first step of such transition.

Acknowledgment

The author gratefully acknowledges the contribution of Pete Ward. It should be stressed that the views in this paper are authors own and do not necessarily represent the views of Teleste.

References

[1] ONVIF, www.onvif.org, visited 14.7.2011

[2] Physical Security Interoperability Alliance PSIA, www.psiaalliance.org, visited 14.7.2011

[3] World Wide Web Consortium W3C, http://www.w3.org, visited 14.7.2011

[4] A. Avižienis, J. C. Laprie, B. Randell, “Dependability and its threats: a taxonomy,” IFIP International Federation for Information Processing, vol. 154/2004, pp. 91–120.

[5] G. Koslovsi, Wai-Leong Yeow, C. Westphal, Tram Truong Huu, J. Montagnat, and P.Vicat-Blanc, “Reliability Support in Virtual Infrastructures,” Proc. IEEE 2nd Internat.Conf. on Cloud Computing Technology and Science (cloudCom) IEEE Press, Dec. 2010, pp. 49-58, doi: 10.1109/CloudCom.2010.23.

[6] R. L. Krutz and R. D. Vines, “Cloud Security: a comprehensive Guide to Secure Cloud Computing,” Wiley Publishing, Inc., Indianapolis, 2010, ISBN: 978-0-470-58987-8.

Diagnostic Fusion for Dependable Vehicle Architectures

Patrick E. Lanigan, Priya Narasimhan
Carnegie Mellon University
Electrical & Computer Engineering
planigan@ece.cmu.edu, priya@cs.cmu.edu

Thomas E. Fuhrman
General Motors Research & Development
Electrical & Controls Integration Lab
thomas.e.fuhrman@gm.com

Abstract – **Despite extensive design processes, emergent and anomalous behavior can still appear at runtime in dependable automotive systems. This occurs due to the existence of unexpected interactions and unidentified dependencies between independently-developed components. Therefore, system-level mechanisms must be provided to quickly diagnose such behavior and determine an appropriate corrective action. DIAGNOSTIC FUSION describes a holistic process for synthesizing data across design stages and component boundaries in order to provide an actionable diagnosis.**

Keywords – diagnosis; dependable computing; automotive; data fusion

1 Introduction

A growing trend in the automotive industry is toward features that assist the driver in maintaining safe control of the vehicle under a variety of conditions. Previously, such assistance has been provided *passively* in the form of information or warnings. These features are now being given increasing amounts of authority to control the vehicle's motion by *actively* supplementing the driver's inputs. The long term trend is towards fully autonomous operation [17, 19].

This trend has largely been enabled by advances in software-intensive distributed systems. Because these are safety-critical systems, they must be designed to tolerate faults and provide high levels of dependability. Typically, a systematic safety analysis is conducted during the design phase to evaluate both the severity and likelihood of the consequences of possible faults. Formal verification methods are used to analyze system dependability. The upcoming ISO 26262 standard for functional safety in automotive electronics recommends that fault-injection also be included as part of the dependability analysis of critical systems [10].

Despite these extensive design processes, emergent and anomalous behavior can still appear at runtime in dependable automotive systems. This occurs due to unexpected interactions and unidentified dependencies between independently-designed components. These interactions are not readily apparent to the system designers and might not be captured by system models. Therefore, system-level mechanisms must be provided to quickly diagnose such behavior and determine an appropriate corrective action at runtime. Diagnostic approaches that operate strictly at the component or subsystem level and rely mainly on functional models may not provide a satisfactory diagnosis. A holistic approach that analyzes empirical metrics *as well as* functional models, and then synthesize the information *across* component and subsystem boundaries is needed.

2 Diagnostic Fusion

Sensor fusion is a well-known technique for combining multiple sources of sensor information, and then correlating that information to get a composite view of the state of the environment being sensed, as well as the state of health of the sensors being fused. Sensor fusion does not itself come up with new sensing technologies, but combines the existing sensing technologies at the system level. By analogy, *diagnostic fusion* does not define new diagnosis algorithms or methodologies, but finds ways to combine existing diagnosis algorithms and methodologies at the system level to satisfy the goals, requirements, and constraints of the system.

Diagnostic fusion is parameterized by the *instrumentation* that it uses to collect data (see Section 2.1) and the *algorithms* that it uses to extract and combine information from the collected data (see Section 2.2). Defining these parameters involves navigating the tradeoff space discussed in Section 3.

2.1 Instrumentation

Instrumentation refers to a source of data from which information can ultimately be obtained through analysis. The holistic approach that we propose for diagnostic fusion is unique in that it instruments the design process as well as the developed system. The data obtained through design-time instrumentation ultimately drives the instantiation of run-time instrumentation (see Section 3.1).

Design-time instrumentation points are derived from the artifacts produced at each stage of the design process. Such artifacts include design documents, models and empirical data. For example, Failure Mode and Effects Analysis (FMEA) documents provide information that can be used to develop fault signatures. Fault Tree Analysis (FTA) documents can contain information that characterizes dependencies and interactions between system components. Component and system models (e.g., MATLAB/Simulink models) can be used to identify potential run-time instrumentation points. Fault models, functional requirements, and safety specifications can be used to derive the level of detail that an actionable diagnosis is required to provide. Empirical data from fault-injection processes and vehicle prototypes can provide information on normal versus abnormal system behavior.

At run-time, there are numerous discrete instrumentation points that can provide diagnostic data. These instrumentation points can provide black-box, white-box or gray-box views and exist at the system-level as well as the component-level. Examples of potential run-time instrumentation points are the error indicators provided by the communication controller, hooks inserted into software components [13], and Operating System (OS) metrics such as context-switch rates. Another approach demonstrates the extent to which diagnosis is possible using only passive monitoring in FlexRay-based networks [2]. Several aspects of the FlexRay protocol that can be used to aid diagnosis under such restrictions, such as syntactic failures in the value domain (e.g., Cyclic Redundancy Check (CRC), header values), semantic failures in the value domain (e.g., application specific plausibility checks) and failures in the time domain (e.g., early, late or missing messages).

2.2 Algorithms

By combining and analyzing the trends of data at the component-level, subsystem-level and system-level, the diagnostic fusion module can detect escalating anomalous behavior in the system, localize the source of the problem and provide an actionable diagnosis. This approach involves employing data analysis through machine-learning and data-mining techniques on the generated error logs and the instrumented data that is extracted out of the system and its components. The diagnostic fusion process will need to correlate time-stamped data across multiple Electronic Control Units (ECUs) and subsystems, not only to localize the source of a failure, but also to examine possible propagation paths that can lead to additional, related failures.

Failure diagnosis approaches in enterprise systems typically localize anomalous system behavior through statistical analysis of time-series data [6, 8, 9, 11, 14] or through control-flow and data-flow analysis [1, 3, 5, 7, 12]. However, the failure diagnosis approaches developed for enterprise systems might not be directly applicable to automotive systems because automotive systems have limited processing and storage capacity and might not support the level of instrumentation and processing needed by the enterprise approach. Automotive systems also generally require a higher degree of accuracy and lower diagnosis latencies than enterprise systems due to the safety-critical and interactive nature of chassis and powertrain subsystems.

For example, peer comparison is a valuable tool for anomaly-detection, especially in enterprise environments with fluctuating workloads. However, peer comparison is less effective with correlated failures, which occur when a fault originates in one node and then propagates to other nodes in the system. Emergent behavior is likely to arise from correlated failures between components with unidentified dependencies. Peer comparison also requires a certain level of homogeneity to exist between the compared peers, whereas automotive distributed systems are largely heterogenous.

The classic formulation of system diagnosis is the *Preperata-Metze-Chien (PMC) model* [16]. Under the PMC model, components test each other according to predetermined assignments. The set of test results (called the **syndrome**) can then be collected and analyzed to determine the health of each component. Subsequent work extended the PMC model by addressing limitations that made it impractical for application in real fault-tolerant systems; an extensive survey of such work is available [4]. System-diagnosis algorithms developed for automotive systems leverage local status-indicators provided to produce a global view of the network's health [15, 18]. This is accomplished by disseminating and aggregating diagnostic information via diagnostic messages, and then performing analysis on the aggregated data.

3 Diagnostic Requirements and Tradeoffs

Developing a run-time instantiation of the diagnostic fusion process will require navigating a complex tradeoff space, which is comprised of the relationships between *coverage*, *latency*, *accuracy*, and *cost* requirements.

- The diagnostic **latency** is the time between the activation of a fault and the output of an actionable diagnosis.

- Safety process standards define **coverage** as the percentage of possible errors that can actually be detected by a system. In functional safety standards such as ISO26262 [10], the required coverage will vary with

Automotive Safety Integrity Level (ASIL). For example, an ASIL D system may require 99% coverage of all memory errors, while an ASIL C system may require only 90% coverage.

- Diagnostic **accuracy** is the probability that any given diagnosis is correct, and can be expressed in terms of false-positive and false-negative rates.
- Diagnosis will impose some overhead, or **cost**, on the system. The cost can also be expressed in economic terms if additional resources are required to compensate for or implement diagnostic functions.

Some very simple examples serve to illustrate the trade-off space. Clearly, a highly accurate diagnosis might take longer to produce, increasing latency. If you need low latency, you might have to allow for a reduction in accuracy. On the other hand, in some instances, the latency could be decreased by adding resources, thereby raising the cost. High coverage might require more resources or instrumentation, which would also increase the cost of diagnosis. The diagnostic fusion process could cover a large space of faults with reduced accuracy, or a small space of faults with greater accuracy, especially when trying to discriminate between fault types.

3.1 Diagnosis Advisor

Manually balancing these tradeoffs can be a significant challenge for system designers. Therefore, we further propose a Diagnosis Advisor (DA) that characterizes the system at design-time and develops set of parameters that are used to instantiate the diagnostic fusion process at run-time. Artifacts from each stage of the design process are provided as inputs to the DA. Such inputs could include fault models, FMEA documents, dependability requirements, feature specifications, or functional models. The DA analyzes these inputs, and provides the system designer with a set of parameters that can be used by the diagnostic fusion process to fulfill the diagnostic requirements of the system.

The analysis performed by the DA is aimed at determining an appropriate set of parameters for instantiating the diagnostic fusion process at run-time. The DA does not develop new algorithms by itself. Rather, it aids the system designer in choosing from a set of existing algorithms that can later be combined by the diagnostic fusion process at run-time. The DA performs its analysis at design-time, and so can be a centralized tool. However, the DA could output either a centralized or distributed configuration for the diagnostic fusion module, depending on the system's diagnostic and dependability requirements.

4 Research Questions and Challenges

This work seeks to address three key research questions.

Research Question 1 *Given specific input requirements relating to cost, dependability and performance, how can a configuration of instrumentation-sources and analysis-algorithms be synthesized, which can diagnose emergent behavior effectively and within the latency required for an actionable response?*

Research Question 2 *What is the coverage of the diagnostic configuration output by the DA, and how do we handle cases when such a configuration is impossible due to the constraints imposed on the system?*

Research Question 3 *What is the trade-off space of the cost (i.e., the overhead of increased instrumentation) vs. the accuracy of fault-localization?*

There are challenges related to each of the following aspects: *holistic*, *data-driven* and *diagnosis*. For instance, diagnostic fusion aims to extract data from every phase of the development cycle. However, it is even more important to extract the *right* data. Moreover, this data comes at the expense of human overhead (e.g., development hours) as well as system and communication overhead. This overhead will need to be minimized, as well as balanced with the benefits provided by diagnostic fusion. Appropriate analytic techniques will then need to be developed, applied and evaluated in order to provide actionable diagnostic output. Finally, the the dependability of the DA and its outputs must be analyzed.

Potential sources of data have to be identified, even though it is not clear what instrumentation points will be available in future automotive architectures. Each potential instrumentation point will then need to be characterized with respect to the utility it provides and the overhead it imposes. Moreover, the relationships between instrumentation points have to be studied. For example, if some sources of instrumentation are redundant or synergistic, can they be correlated as a sanity check? On the other hand, can sources of instrumentation that are disjoint or independent be leveraged to provide a more complete picture of the vehicle's health?

Identifying algorithms that can detect specific kinds of failures based on the instrumentation available to them is a key issue. Once these algorithms have been identified, they will need to be implemented in a resource-constrained environment. For algorithms developed in enterprise environments, this could be a significant challenge. Just as with instrumentation points, the utility provided and overhead imposted by the algorithms will also need to be characterized. Further, the diagnostic accuracy and granularity (e.g., component-level, subsystem-level, etc.) provided by various combinations of algorithms and instrumentation should be shown experimentally as well as analytically.

5 Summary

Despite extensive design processes, emergent behavior can still appear at runtime in dependable automotive systems. The holistic approach used in diagnostic fusion can address this problem in two ways. First, by synthesizing data across design phases, dependencies and interactions that could have otherwise been undetected can be identified. Second, by coherently characterizing the expected behavior of the system, diagnostic fusion will provide a more robust means of detecting and diagnosing emergent behavior as it occurs.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings, 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 74–89, New York, NY, USA, October 2003. ACM.

[2] E. Armengaud and A. Steininger. Pushing the limits of online diagnosis in flexray-based networks. In *Proceedings, 2006 IEEE International Workshop on Factory Communication Systems*, WFCS '06, pages 44–53, Piscataway, NJ, USA, June 2006. IEEE.

[3] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. *ACM SIGCOMM Computer Communication Review*, 37(4):13–24, August 2007.

[4] M. Barborak, M. Malek, and A. Dahbura. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2):171–220, June 1993.

[5] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings, 6th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '04, pages 259–272, Berkeley, CA, USA, December 2004. USENIX Association.

[6] S. Bhatia, A. Kumar, M. E. Fiuczynski, and L. L. Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proceedings, 8th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '08, pages 103–116, Berkeley, CA, USA, December 2008. USENIX Association.

[7] R. P. J. C. Bose and S. H. Srinivasan. Data mining approaches to software fault diagnosis. In *Proceedings, 15th International Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications*, RIDE-SDMA 2005, pages 45–52, Los Alamitos, CA, USA, April 2005. IEEE Computer Society.

[8] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. *ACM SIGOPS Operating Systems Review*, 39(5):105–118, December 2005.

[9] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *Proceedings, 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOSPLA '04, pages 251–269, New York, NY, USA, October 2004. ACM.

[10] *ISO/DIS 26262: Road vehicles – Functional safety*, volume 4–6. International Organization for Standardization, Geneva, Switzerland, 2009.

[11] S. Kavulya, R. Gandhi, and P. Narasimhan. Gumshoe: Diagnosing performance problems in replicated file-systems. In *Proceedings, 2008 IEEE Symposium on Reliable Distributed Systems*, SRDS '08, pages 137–146, Los Alamitos, CA, USA, October 2008. IEEE Computer Society.

[12] E. Kiciman and A. Fox. Detecting application-level failures in component-based internet services. *IEEE Transactions on Neural Networks: Special Issue on Adaptive Learning Systems in Communication Networks*, 16(5):1027–1041, September 2005.

[13] P. E. Lanigan, P. Narasimhan, and T. E. Fuhrman. Experiences with a CANoe-based fault injection framework for AUTOSAR. In *Proceedings, 2010 IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, pages 569—574, Los Alamitos, CA, USA, June 2010. IEEE Computer Society.

[14] S. Pertet, R. Gandhi, and P. Narasimhan. Fingerpointing correlated failures in replicated systems. In *Proceedings, 2nd USENIX Workshop on Tackling Computer Systems Problems with Machine Learning Techniques*, SysML '07, pages 9:1–9:6, Berkeley, CA, USA, April 2007. USENIX Association.

[15] P. Peti, R. Obermaisser, and H. Kopetz. Out-of-norm assertions. In *Proceedings, 11th IEEE Real Time and Embedded Technology and Applications Symposium*, RTAS '05, pages 280–291, Los Alamitos, CA, USA, March 2005. IEEE Computer Society.

[16] F. P. Preperata, G. Metze, and R. T. Chien. On the connection asssignment problem of diagnosable systems. *IEEE Transactions on Electronic Computers*, EC-16(6):848–854, December 1967.

[17] J. D. Rupp and A. G. King. Autonomous driving – a practical roadmap. SAE Technical Paper Series 2010-01-2335, SAE International, Warrendale, PA, USA, October 2010.

[18] M. Serafini, N. Suri, J. Vinter, A. Ademaj, W. Brandstäter, F. Tagliabò, and J. Koch. A tunable add-on diagnostic protocol for time-triggered systems. In *Proceedings, 2007 IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, pages 164–174, Los Alamitos, CA, USA, June 2007. IEEE Computer Society.

[19] C. Urmson et al. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8):425–466, July 2008.

Downtime Analysis of Virtual Machine Live Migration

Felix Salfner
Department of Computer Science
Humboldt-Universität zu Berlin, Germany
salfner@informatik.hu-berlin.de

Peter Tröger, Andreas Polze
Operating Systems and Middleware Group
Hasso-Plattner-Institute at University Potsdam, Germany
{peter.troeger,andreas.polze}@hpi.uni-potsdam.de

***Abstract*—Virtual machine environments like VMware, XEN, KVM, and Hyper-V support live migration of guest operating systems, which is used in data centers to provide uninterrupted service during maintenance or to move computation away from failure-prone hosts. The duration of migration, as well as the virtual machine downtime during this process are essential when assessing if service availability agreements might be violated.**

We present the result of an experimental study that analyzed virtual machine live migration downtime and duration. We show that total migration time as well as downtime are dominated by specific memory utilization patterns inside the virtualized guest. We experienced that downtime involved by live migration can vary by a factor of more than 23, which can have significant impact on service availability.

***Keywords*-virtual machine, live migration, downtime**

I. Introduction

Virtualization as a concept for isolation and multiprogramming has been known since the late 60's [1]. Today, many virtualization products for commodity server and desktop environments exist. Most platforms support *live migration*, which allows to move a running virtual machine (VM) to a new physical host with minimal service interruption. This renders live migration a very attractive tool for various scenarios in dependable computing. Currently the predominant use of live migration is in data centers or compute clouds where VMs can be moved across physical hosts for load balancing, server consolidation or maintenance. In all these cases knowing the downtime involved by moving the VM is essential when service availability guarantees have to be fulfilled: the time of service interruption must not exceed the clients retry intervals.

A second area where live migration is used is proactive fault management. VMs are moved away from a physical node that has been predicted to show a failure in the near future (see, e.g., [2]). In addition to the downtime involved in moving the VM, the total duration of migration is an important characteristic. This is because the entire migration has to have finished before the failure occurs.

However, the majority of existing work that builds on live migration of VMs simply assumes some fixed (in many cases arbitrary) duration of the live migration and the downtime involved by it. According to the experiments presented in this paper, such an assumed value has to be chosen very carefully since migration time as well as downtime can vary by an order of magnitude or more, depending on the memory workload. It is the goal of this paper to systematically investigate the factors determining the time needed for VM live migration.

II. Virtual Machine Live Migration

Within the different existing virtualization frameworks with live migration support, the basic principle is that the virtualization cluster management actively moves a virtualized system while it is still executing and is still changing the hardware's and software's state. Today's products realize this by a delta-copying approach where modified memory regions are incrementally transferred until a lower threshold for data to be moved is reached. In the subsequent phase in which the VM is stopped, the remaining resources are copied and reconfigured and the VM is resumed on the destination host. This leads to the two characteristics investigated in this paper:

- *migration time* is the time from start of the live migration process until the virtualization framework notifies that the source host can be deactivated.
- *downtime* or *blackout time* is the phase during migration when there is a user-perceptible service unavailability.

The most difficult procedure in live migration is the transfer of main memory state. As live migration environments typically share storage within the migration cluster, swapped out memory pages do not have to be considered. Read-only pages from the working set (such as code pages) need to be migrated only once, whereas data pages could be modified again after their initial transfer. Transfer of writable memory can happen theoretically in three phases [3]: In the initial *push phase*, in which the source machine's actively used set of pages is copied to the destination host in rounds. In the subsequent *stop-and-copy phase*, in which the source VM is suspended, remaining memory regions are transfered, and the VM is resumed again on the destination host. The length of this phase is the VM downtime. The last step is the *pull phase*, where the VM running on the destination host might access memory regions that are still not migrated, which are then pulled from the source host. The end of the pull phase marks the end of the migration time. The time of transition from one phase to the next is controlled by adaptive algorithms that take into account various aspects such as the number of dirty memory pages, etc. Most live migration products combine the first two phases in a so-called *pre-copy* approach.

III. Experiment Design and Load Model

Live migration duration is influenced by load factors from inside the VM and from the underlying physical host. For our investigations we assumed a typical (and recommended) setup where applications only run in VMs and there are no applications running on the physical host directly (except for the hypervisor).

Experiments carried out prior to the ones presented here have shown no impact on migration time and downtime when running multiple VMs on one physical host. Results shown here have therefore been measured with just one VM per physical host. We assumed a model of VM operation without *over-commitment*, which is a VM configuration where the sum of virtual memory of all VMs on a host does not exceed the amount of physical memory.

The focus of this work is on application-specific effects on live migration. Since in most scenarios the migration network is not a controllable parameter we did not investigate effects of the migration network on migration performance. Additional tests investigating network usage showed that the migration frameworks handle network capacity issues carefully so that this assumption appears valid.

The goal of our experiments are to investigate the effects of the following factors on migration time as well as downtime: CPU load, configured memory size of the VM, utilized amount of memory, and memory modification pattern for two different virtualization products.

A. Load Generators

In order to be able to analyze the effects of each factor and its combinations we used three different load generators - one for CPU load and two for memory load.

The *CPU load generator* was based on *burnP6* and *cpulimit* generating a controllable CPU utilization in the running VM.

The *locked pages generator* is used for analysis of static memory allocation that cannot be swapped out. This is achieved by allocating a given amount of memory, writing random data to it and locking it in physical memory using the according system call.

In order to investigate the effect of modifying memory pages while the VM is migrated, we programmed a *dirty pages generator*, simulating memory write access of applications running in the VM. It implements a cyclic memory modification pattern by continuously writing pre-computed random data to locked memory locations. This pattern is motivated by server applications, which modify memory regions based on incoming requests. These modifications can be expected to have comparable characteristics for the majority of requests, e.g., by always reading some data, storing logging information, and returning the result.

A list of all parameters investigated in our experiments is provided by Table I.

Table I
Investigated parameters

HYPERVISOR	The virtualization framework used
VMSIZE	Amount of main memory statically configured for the VM
LOAD	CPU utilization of the virtualized operating system
WSET	Working set, the sum of utilized memory
PERIOD	The period for one memory modification cycle
BPC	Blocks per cycle. Number of modified blocks per cycle
FILL	Filling degree. The average percentage of a memory block being actively modified

B. Technical Setup and Issues

All tests were performed on two Fujitsu Primergy RX300 S5 machines with a shared iSCSI drive, the migration was performed for a VM running Linux 2.6.26-2 (64 bit). All VMs were configured to have one virtual CPU and a varying amount of (virtualized) physical RAM. In all cases, the virtualization guest tools / drivers were installed. Native operating system swapping was activated, but not aggressively in use due to the explicit limitation of the allocated amount of memory.

We conducted all experiments with the two hypervisors VMware and Xen. Experiments for VMware were performed with ESX 4.0.0, using the vCenter server software for migration coordination. High availability features were deactivated. Experiments for Xen were performed with Citrix XenServer 5.6 (Xen core 3.4.2). Both Xen hosts were configured to form a pool, the test scripts were executed in the dom0 partition of the pool master.

Total migration time was measured by capturing the runtime of the product command-line tool that triggers a migration. Downtime was measured by a high-speed ping (50 ms) from another host, since the virtualization products do not expose this performance metric by themselves. The downtime is expressed as the number of lost Ping messages multiplied by the ping interval.

Live migration, similar to every performance-critical software feature, is influenced by a manifold of hardware / software factors. We are aware of the fact that new product versions, node and networking hardware as well as special optimization switches can lead to better or worse results. Nevertheless, the point of our investigations is to identify major impact factors when using live migration for dependability purposes.

IV. Single Variable Experiments

Since the number of parameters (also called *factors*) is too large for an investigation of all combinations of factors with each factor tested at multiple levels, we first aimed at reducing the set of factors. In order to do so, we investigated each parameter individually. As will be shown in this section, this analysis helped to eliminate the two parameters LOAD and FILL.

In order to investigate a single factor we set all but the investigated parameters to fixed values and measured both downtime and total migration time at various levels of the investigated parameter.

A. VMSIZE

We investigated the influence of the configured VM main memory with different settings for both products. Specifically, we have investigated idle VMs with VMSIZE set to 512MB, 1GB, 2GB, 4GB and 8GB RAM. The virtual machines were idle, so no load generator was used. While VMware showed a nearly constant migration time, Xen had a linearly growing migration time with increasing VMSIZE.

B. LOAD

For the investigation of the influence of CPU load on migration time, we performed at least 10 migrations per CPU utilization degree, ranging from 0% to 100% artificial CPU load in steps of size ten using the CPU load generator. Results showed almost no impact with a 95% confidence interval of not more than -/+ 1s for all load values.

Both products also showed a constant (significantly smaller) downtime in all constellations, with a 95% confidence interval of not more than +/- 10% of the average downtime.

The results suggest that both virtualization frameworks reserve enough CPU time for their own management (migration) purposes. Live migration scenarios seem to only depend on non-CPU utilization factors. We could hence safely drop CPU load as an influencing factor in subsequent experiments.

C. WSET

Using the locked pages generator, we varied the size of the working set from zero to 90% of the virtual memory available to the VM. Results show that VMware downtime as well as migration time depend on WSET. Xen also shows dependency on WSET so that the effects of WSET are analyzed further in Section V.

D. FILL

In order to rely on the trap and page table mechanisms of the operating system, all VM migration approaches copy memory content in pages. Hence an entire page has to be migrated even when only a fraction of a page is written. We tested this assumption by “filling” pages to a varying degree using the dirty page generator. As expected, both virtualization toolkits showed no effect on downtime or migration time.

E. PERIOD and BPC

The parameters PERIOD and BPC determine how frequently memory pages are modified. In order to assess how total migration time and downtime are affected by them we conducted a series of experiments where we varied PERIOD for a number of settings for BPC using the dirty page generator. Results show that both downtime and migration time are strongly affected by the two parameters. To check for stochastic variability, we determined 95% confidence intervals, which never exceeded 5% of the average value. However, the influence of PERIOD and BPC are complex and will be further investigated in Section V.

To better understand the complex behavior we performed a source code analysis of Xen and had personal communication with VMware representatives. The behavior seems to be mainly related to the rate-adaptive migration control. The relevant aspect here is the dirty page diff set, the fraction of pages that are scheduled to be copied in each next round of the pre-copy phase. Both virtualization products obviously identify ”hot pages” in this set and shift such pages more aggressively to the stop-and-copy phase because for hot pages a block transfer in the stop-and-copy phase is potentially more effective (depending on “hotness” of the page, network link speed and other factors). Akoush et al. [4] made similar investigations in their live migration performance analysis.

F. Summary

Summing up these experiments, we observe that live migration duration as well as downtime can depend heavily on the investigated factors. On the other hand we saw that CPU load as well as the degree to which memory pages are filled do not influence migration performance significantly which allows us to exclude them from further investigations.

In order to deal with the mutual non-trivial dependencies seen in this section we subsequently devised experiments that investigate all combinations of factors, as will be presented in the next section.

V. Multi Parameter Experiments

From the experiments shown in the previous section we were able to conclude that the factors LOAD and FILL can be omitted from further analysis.

A second reduction in the number of factors can be achieved by leveraging on the fact that BPC (blocks per cycle) and PERIOD (duration of one cycle) can be combined into one factor $RATE = \frac{BPC}{PERIOD}$, which denotes the number of blocks that are modified per millisecond.

We have hence reduced the number of factors to the following three parameters: VMSIZE, WSET, and RATE.

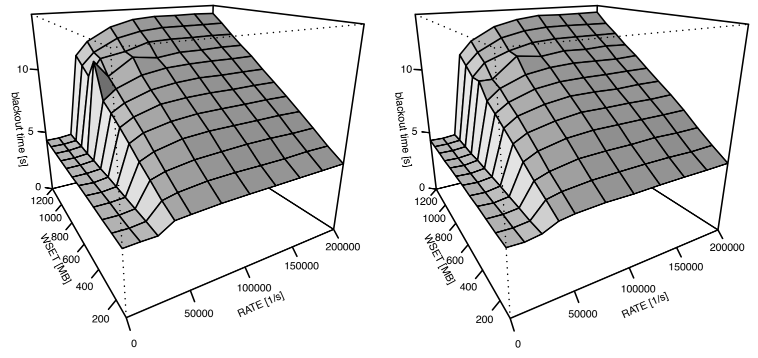


Figure 1. Mean downtime for Xen plotted over WSET and RATE for VMSIZE=4096 (left) and VMSIZE=8192 (right)

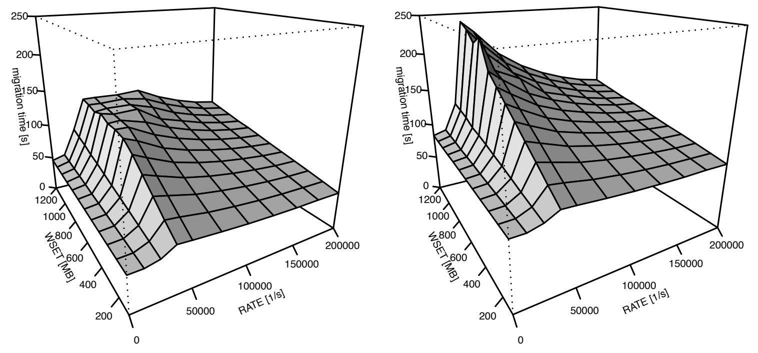


Figure 2. Mean migration time for Xen plotted over WSET and RATE for VMSIZE=4096 (left) and VMSIZE=8192 (right)

We performed experiments according to a full factorial design, meaning that all possible combinations of parameter levels have been measured in the experiment. More specifically, for Xen we investigated a total number of 528 parameter combinations (treatments), each with 20 measurements resulting in an overall number of 10560 migrations. In each experiment we measured migration time and downtime as response variables. In case of the VMware hypervisor, we performed experiments for 352 combinations resulting in 7040 migrations.

In the following we will discuss results for each virtualization framework separately.

A. *Analysis of XenServer*

As we have three factors (plus a response variable) we cannot present the entire results in one plot. Since VMSIZE has the least number of levels, we decided to plot the mean response, i.e. mean migration time or downtime, over WSET and RATE for fixed values of VMSIZE (see Figures 1 and 2). Comparing the two figures, we can see that downtime shows a very different behavior in comparison to migration time, although the first is part of the latter.

Downtime (Figure 1) in general increases with increasing WSET and increasing RATE. This is not surprising as an increased usage of memory (more pages written at an increasing rate) requires more memory to be transferred in the stop-and-copy phase. We can also conclude from the figure that WSET seems to have a linear effect on downtime, regardless of the values of VMSIZE and RATE.

Turning to total migration time (Figure 2) we observe that the mean migration time is more irregular. It came as a little surprise to us that for RATE levels ”above the jump” total migration time decreases with increasing RATE. In order to check that this behavior really occurs we have carried out separate experiments specifically targeted to this question with the same consistent result. The behavior can be explained by the documented stop conditions for the precopy phase in these products. The precopy phase of Xen stops (1) when a sufficiently small amount of memory is left on the source or (2) if an upper limit for the transferred data was reached or (3) if the time taken becomes too long (measured by the number of pre-copy rounds) [4]. Hence if the modification rate grows beyond a certain value close to the link speed, the algorithm will end the pre-copy phase earlier resulting in the observed behavior of constant downtime and decreasing overall migration time.

One peculiarity in Figures 1 and 2 is the abrupt change at a RATE level around $30{,}000\frac{1}{s}$. In order to analyze this further, we conducted additional “zoom-in” experiments that investigated a sub-range of values for RATE at greater level

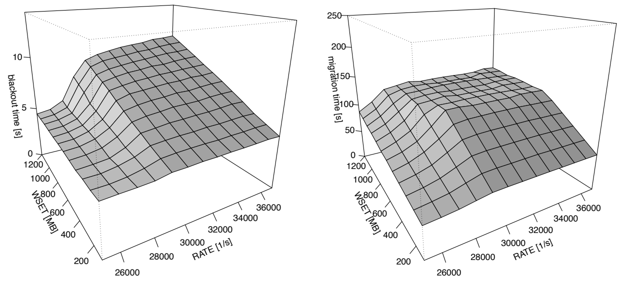


Figure 3. Mean downtime (left) and migration time (right) for Xen with additional WSET and RATE levels in the "zoom-in area", VMSIZE=4096

of detail (see Figure 3). As it can be seen from the plot, the change is not as abrupt as might have been concluded from Figures 1 and 2 and only appears to be abrupt due to the scale of the plot and due to a lack of intermediate measurement points.

The effect of VMSIZE can be observed by comparing the two plots in Figure 2. It can be seen that VMSIZE has a non-trivial effect on migration time: since the shapes look very different at different levels of VMSIZE, the effect does not appear to be linear, except for the case where RATE equals zero. There is no effect of WSET if RATE is zero.

The plots in Figures 1 to 3 show times averaged over all 20 measurements. In order to assess the variability in the data, we report the ratio of maximum to minimum values as well as standard deviations for the data in Table II. Specifically, two ratios and two standard deviations are reported: the ratio of the maximum treatment mean to the minimum treatment mean and the ratio of the maximum to the minimum values across all measurements. Regarding standard deviations we report the largest standard deviation computed within each treatment as well as the standard deviation for the overall data set. In addition, the table reports the mean time averaged across all measurements.

The table quantifies what has also been observable from the plots: Both migration time as well as downtime vary tremendously depending on the three investigated parameters.

B. Analysis of VMware

Due to space limitations we report results for VMWare only for VMSIZE equal to 4GB (see Figure 4). This is no severe limitation as the behavior is very similar for other values of VMSIZE.

As can easily be observed the behavior differs significantly from the one of Xen, which emphasizes that the choice of the hypervisor product can have significant impact on availability. The main reason for the different behavior seems to be the different rate-adaptive algorithms employed in the two virtualization products.

Variability of the data for VMware is also listed in Table II. Regarding the max:min ratio of downtime computed

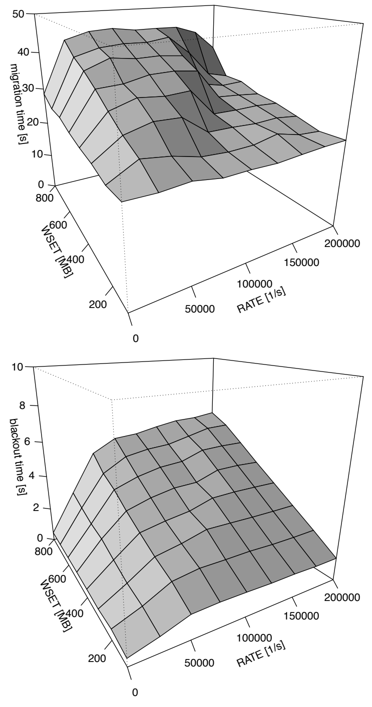


Figure 4. Mean migration time (top) and downtime (bottom) for VMware plotted over WSET and RATE, VMSIZE=4096

from treatment means we have observed a ratio of 16.27. This shows that due to different memory load the maximum mean downtime can be 16.27 times as large as the minimum mean downtime. If we instead consider the maximum and minimum value observed across all experiments, the factor even goes up to 23.83! The conclusion from this observation is that if service downtime is critical for meeting availability goals, a realistic assessment of availability can only be achieved if the maximum downtime for the application-specific memory load is used.

VI. Related Work

In the area of dependable computing, VM live migration has primarily been used as a tool. Two examples are *proactive fault tolerance* [2] and the approach to resource allocation proposed in [5].

A second group of related work deals with various aspects of implementing VM live migration. Hines and Gopalan [6] discuss the modification of Xen for *post-copy* live migration. Sapuntzakis et al. [7] introduced several optimization approaches for VM live migration, among which *ballooning* is best-known, which forces the VM to swap out as much memory as possible.

This work, however, is somewhere in between using live migration as a tool and investigating aspects of its implementation: We have focused on the factors determining downtime and migration time from an application's perspective. A work that is closer related to ours is [3], which

Table II
DATA VARIABILITY

Hypervisor / Guest	time	Mean time [s]	Max:Min Ratio		Standard Deviation	
			Mean	Overall	Treatment Max [s]	Overall [s]
XenServer / CentOS	migration	89.73	9.01	9.10	6.32	39.08
	downtime	7.69	3.17	3.46	0.62	2.94
VMware / Linux	migration	30.93	2.24	2.96	7.72	7.51
	downtime	3.10	16.27	23.83	0.50	1.80

investigates the effect of the size of the *writable working set*. The migration times reported are much smaller than the ones reported here. This is probably due to the fact that the workloads used in their experiments do not result in significant memory load.

VII. CONCLUSION

With growing capacity of commodity server hardware and increased consolidation efforts, virtualization has become a standard approach for data center operation - not only on the mainframe and for UNIX systems, but also in the world of Intel servers. Live migration of virtual server workloads can be employed to implement workload-driven system management as well as mechanism to free server hardware that is due for maintenance and repair. However, in order to give guarantees on application availability or responsiveness as well as for proactive fault management, solid estimations either about the total duration of live migration or the length of service downtime are mandatory.

In this paper, we have reported about our research on major factors that influence migration time and migration-induced downtime. Our measurements are based on two representative virtualization products, namely VMware ESX 4.0.0 and Citrix XenServer 5.6. By carrying out a wide range of experiments, our analysis shows that performance of live migration can vary significantly depending on the memory load and memory access patterns of the guest system.

The results can be used, e.g., to investigate the applicability of VM live migration in the context of proactive fault management: If VMs are to be migrated away from failure-prone hosts the failure prediction algorithm needs to predict failures further in the future than the total duration of migration. Our results also help to assess if service availability assertions are violated by the downtime introduced by live migration of a VM running the service. Even if the absolute numbers may be different for future versions of the virtualization products our results highlight that application-specific investigations are crucial to assess the feasibility of live migration in a particular scenario.

A second area where our results are useful is to help to improve live migration features of virtualization products. For example, the observation that Xen migration time depends on the size of virtual RAM configured might be an indicator how live migration can be improved futher, or might even stimulate new ideas for VM live migration.

Future work will involve investigation of other virtualization approaches (e.g., KVM and Microsoft Hyper-V). We will also focus on the relationship between the load generators used in this work and real-world applications.

VIII. ACKNOWLEDGMENTS

We would like to thank Matthias Richly from Hasso Plattner Institute for the realization and management of the experimentation environment. All hardware resources were made available by the HPI FutureSOC lab infrastructure.

REFERENCES

[1] R. P. Goldberg, "Survey of Virtual Machine Research," *IEEE Computer*, vol. 7, no. 6, pp. 34–45, 6 1974.

[2] C. Engelmann, G. R. Vallée, T. Naughton, and S. L. Scott, "Proactive Fault Tolerance Using Preemptive Migration," in *Proceedings of 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP, 2009*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 252–257.

[3] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *Proceedings of Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation, NSDI, 2005*. Berkeley, CA, USA: USENIX Association, 2005, pp. 273–286.

[4] S. Akoush, R. Sohan, A. Rice, A. W. Moore, and A. Hopper, "Predicting the Performance of Virtual Machine Migration," *Modeling, Analysis, and Simulation of Computer Systems, International Symposium on*, vol. 0, pp. 37–46, 2010.

[5] S. Fu, "Failure-aware resource management for high-availability computing clusters with distributed virtual machines," *Journal of Parallel and Distributed Computing*, vol. 70, no. 4, pp. 384–393, 2010.

[6] M. R. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," in *Proceedings of 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, Washington, DC, USA*, ser. VEE '09. New York, NY, USA: ACM, 2009, pp. 51–60, washington, DC, USA.

[7] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum, "Optimizing the migration of virtual computers," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 377–390, 2002.