Piotr GAWKOWSKI, Konrad GROCHOWSKI, Paweł PISARCZYK
INSTYTUT INFORMATYKI, POLITECHNIKA WARSZAWSKA,
ul. Nowowiejska 15/19, 00-665 Warszawa

Fault injection approach towards dependability analysis in real time operating systems

Dr inż. Piotr GAWKOWSKI

Ukończył studia na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej, tam też obronił pracę doktorską w 2005 r. Obecnie jest adiunktem w Instytucie Informatyki na tym samym wydziale. Jego zainteresowania naukowe to wiarygodność systemów komputerowych, metody tolerowania błędów, symulacyjna ewaluacja wiarygodności systemów cyfrowych.

e-mail: P.Gawkowski@ii.pw.edu.pl

Mgr inż. Konrad GROCHOWSKI

Ukończył studia na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej, gdzie uzyskał tytuł inżyniera w roku 2008 i magistra w roku 2010. Obecnie jest studentem studiów doktoranckich w Instytucie Informatyki na tym samym wydziale. Jego zainteresowania naukowe to systemy wbudowane i czasu rzeczywistego, wiarygodność oraz inżynieria oprogramowania.

e-mail: K.Grochowski@ii.pw.edu.pl

Mgr inż. Paweł PISARCZYK

Ukończył studia na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej, gdzie uzyskał tytuł magistra inżyniera w roku 2001. Jest autorem prototypowego systemu operacyjnego Phoenix i jego komercyjnego następcy Phoenix-RTOS przeznaczonego dla procesorów konfigurowalnych. Przygotowuje pracę doktorską w Instytucie Informatyki Politechniki Warszawskiej dotyczącą wiarygodności systemów operacyjnych dla systemów wbudowanych.

e-mail: pawel.pisarczyk@phoenix-rtos.com

Abstract

The paper presents the fault injection approach applicable for dependability evaluation of real-time systems. The developed fault injection environment, called InBochs, is based on modified system emulator Bochs. It is highly flexible in terms of fault specification and results observability reflecting in rich feedback information for a target system developer. The low overhead of the InBochs fulfills tight requirements for RT-system evaluation testbeds. The paper describes the methodology of dependability evaluation basing on an exemplary process control task.

Keywords: dependability analysis, real-time systems, embedded systems, fault injection.

Symulacja błędów w analizie wiarygodności w systemach czasu rzeczywistego

Streszczenie

Wszechobecność systemów wbudowanych i czasu rzeczywistego niesie za sobą potrzebę analizy ich wiarygodności. Dotyczy to nie tylko systemów w zastosowaniach krytycznych (jak aeronautyka, czy sterowanie procesów przemysłowych), gdzie głównym aspektem jest bezpieczeństwo, ale także popularnych urządzeń życia codziennego, od których użytkownicy również oczekują określonego poziomu niezawodności i dostępności. Niezbędna jest więc analiza odporności systemów na różnego rodzaju zakłócenia, m.in. na rosnące niebezpieczeństwo zakłóceń przemijających w systemie cyfrowym, w szczególności tzw. SEU (ang. Single Event Upsets [1], efektem których mogą być przekłamania wartości logicznych w elementach pamięci). Omówiono szereg aspektów analizy eksperymentalnej przy wykorzystaniu techniki programowej symulacji błędów w kontekście badań systemów czasu rzeczywistego oraz przedstawiono system InBochs, który może być zastosowany m.in. do eksperymentalnej analizy wiarygodności systemów wbudowanych oraz czasu rzeczywistego. Bazuje on na programowym emulatorze systemu komputerowego Bochs [5]. Spośród innych rozwiązań ([2] i referencje) InBochs umożliwia m.in. abstrakcję czasu ukrywającą narzuty symulatora oraz język skryptowy symulacji błędów. Jego praktyczna użyteczność została potwierdzona eksperymentami dla dwóch różnych systemów czasu rzeczywistego (RTAI [7, 9] oraz Phoenix [8]) realizujących zadanie sterownika GPC w wersji analitycznej dla procesu reaktora chemicznego (opis w [6] i referencje).

Słowa kluczowe: analiza wiarygodności, systemy czasu rzeczywistego, systemy wbudowane, symulacja błędów.

1. Introduction

High dependability (and safety in particular) is required in many control systems used in industry, automotive, medicine, civil engineering applications, etc. So, an important issue is to analyze system susceptibility to internal faults and identify unsafe situations. On the other hand, a user expects a reasonable level of reliability and availability even from many commonly used digital processing devices (e.g. mobile phones, PCs, multimedia players, set-top boxes). For each digital processing system some fault threads exists. These threads have several origins: some of them are common for all digital systems, while others are target specific.

One of the major sources of faults is undisputedly software due to the complexity of implemented functionalities, interfaces, unclearly stated system specification, and, finally, programming errors of different nature within a code. The software engineering techniques and methodologies are targeted to minimize the above mentioned threats. However, it has to be in mind that the software is assumed to be executed on the fully functional hardware. In many cases this assumption cannot be fulfilled as transient soft errors may occur. With the growing complexity of digital devices, and power voltages and diameters scaling down, the probability of Single Event Upsets (SEU [1]) in memory elements within digital chips is becoming more and more relevant for system dependability. The system corrupted with SEU may behave unexpectedly, including the critical errors in produced results or safety violated outputs. To fulfill the dependability requirements, a critical design (not only in terms of safety) should be examined with both: the classical software testing (i.e. with software engineering methodologies) and also towards its sensitivity/robustness to erroneous conditions.

For fault-sensitivity examination purpose some various fault simulation techniques have been proposed ([2] and references therein). Most of them have been used to study calculation oriented applications (sorting algorithms, matrix multiplication, FFT, etc.). Some are targeted at specific target system or cover the soft error threads at specific system level (e.g. at physical level, VHDL model, interfaces, protocols). The purpose of the fault injection approach presented in this paper is not limited to fault injection of different types but also to provide high level of controllability over injections and observability of their effects in order to support real time and embedded systems software developers.

Section 2 addresses the important aspects of adapting software implemented fault injection approach to dependability analysis of real time systems and applications. Section 3 presents the InBochs fault injection system. Exemplary experiments are summarized in Section 4. Then, the paper is concluded.

2. Software fault injection in RT systems

Software Implemented Fault Injection (SWIFI) has a numerous advantages over other fault injection techniques [2]. Two of the most important are the very high level of controllability over injected disturbances as well as the observability of fault propagation and effects. It is worth noting that for practical reasons it is useful to collect and correlate with each other any observable effects with the execution profiles. Such rich feedback from experiment can give additional knowledge for the target system developer: the statistical dependability overview is supplemented with details of the most fault sensitive system parts, software modules, their code lines and variables, a set of possible system behavior, etc. [3]. That knowledge lets further system hardening that can be verified in the subsequent fault injection experiments hopefully leading to reach the desired level of dependability with respect to its different aspects (like safety, availability etc.).

However, the above mentioned advantages are sometimes not easily achievable in practice. The problems are usually not related to the fault/error injection technique applicability but rather to the controllability and observability problems (due to the complexity of system behavior etc.). Using SWIFI concept to the analysis of a real time operating system some additional aspects arise.

As the target System Under Tests (SUT) considered in the paper is real-time OS, the fault injection technique should not interfere noticeably with the SUT operation. The low overhead of the fault injection tool is crucial. In fact, a fault injector usually consists several modules:

- Fault triggering mechanism (responsible for identification of the proper instant for fault injection – decides *when* the fault is injected) – fault triggering may be related to the wall-time, in-system ticks, internal or external system events. Moreover, it can be related to the executed instructions or even correlated with the source code lines. Different fault distribution profiles are useful for different purposes of analysis [4].
- Fault injecting facilities (responsible for simulation of a fault) – depending on the complexity of a given fault model it can be as simple as corruption of a single binary value or it requires execution of sophisticated scenario (to mimic complex errors or device's failure modes).
- Fault effects tracing facilities (responsible for observability of possible after-injection effects and target application/system behavior monitoring) – the availability of such functionality in the SWIFI testbed allows getting more valuable feedback for a system developer as detail information upon single test run is provided.

Each of the listed above modules utilize some resources on the target system. That implies additional overhead in both: resources (hardware and software) and the execution time. In practice, the level of controllability/observability and the details of collected data from the tests are limited. This problem is much stronger in case of real-time or embedded systems than in case of common PC system, as the first ones have usually much less resources and time gaps to be used by fault injector than the second ones. So, for instance, fault effects tracing facilities have to be relatively limited making the analysis hard to set up. In fact, the real time systems should not be aware of the existence of the fault injection/monitoring machinery not just because of its interference during experiments but also during its development (e.g. BDMs can use resources applicable also for fault injector so the debugging might be not possible in parallel).

In order to get valuable data out of the experiment results it is good to have strict limitations over the injected fault space. In some experiments it is valuable to examine the whole system against some resources failure (e.g. memory corruption, device failure) and in others one can be interested mostly in real-time kernel itself, the specific application, its module, task or given function dependability. As a result, the fault injection testbed should provide mechanisms of distinguishing specific *targets* within the SUT. In particular, it is related to identification of instants when a specific task is executed, resource used, kernel function is called etc.

SWIFI can emulate the existence of faults of different types (stuck-at, bit-flips, etc.) or even can emulate complex fault or error models. However, the most commonly used in the community is the single bit-flip fault as it well mimics the occurrence of Single Event Upsets (SEU [1-4]). To examine the fault sensitivity of the given fault location, the set of tests is performed (i.e. experiment). Each test is a single execution of the workload with the disturbance of a given type injected. During the test fault injector pauses the SUT (or just the target workload – Application Under Tests - AUT) at the precisely defined instant of its execution, injects the fault into the target fault location (e.g. CPU register, memory location, instruction code), and (after target resumes the execution) collects (with the post-injection monitoring module) all events occurring in SUT (like exceptions triggered by the target, messages, result files, etc.). As the workload finishes its execution (or the test scenario is finished), the results are judged for correctness by the injector oracle module on the basis of the data collected during the referential (i.e. fault-free) workload execution (called golden run). The whole experiment can be conducted automatically by the fault injector or supervised by the experimenter (e.g. to allow additional interaction and subjective effects observation). At the end of the experiment summarized results are given. In general, four classes of test results are distinguished: C (correct results produced), INC (incorrect/unacceptable results), S (test terminated by the system due to un-handled exception, e.g. memory access violations, invalid opcode), and T (timed-out test). Additionally, if the SUT signals a user that error is detected before the termination, the U category may also be present (workload's result consumer being warned).

3. InBochs fault injector

In the research the InBochs fault simulator is used. It is developed in the Institute of Computer Science at Warsaw University of Technology. It is based on the open-source x86 system emulator Bochs project [5]. The whole system is purely emulated (see Fig. 1) – no virtualization features are used. However, Bochs executed at the single core of double core AMD Opteron 280 (running at 2.4 GHz) assures the performance at the level of 30 million of machine instructions per second enabling quite smooth work with the guest operating systems. As the InBochs extends the original Bochs emulator with fault injection capabilities, the InBochs can also be executed on a variety of hosting operating systems (e.g. Windows, Linux, Solaris, MacOS) and it supports any available x86 operating systems as guest OS.

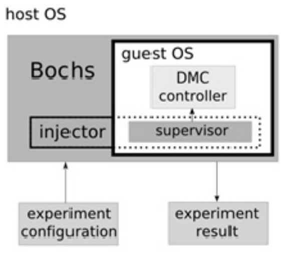

Fig. 1. Concept of InBochs fault injector
Rys. 1. Koncepcja symulatora błędów InBochs

The impact of the injector module within the InBochs is practically imperceptible as it uses the Bochs internal structures and functions to pause the system execution at the precisely defined instant of the target application/task execution and to inject the disturbance. It is worth noting that the fault injection is realized by a script language dedicated for fault injectors. Experiment showed that such an interpreter was not introducing great time delays while it significantly increased the flexibility of the fault injection system (e.g. new fault models can be easily implemented as scripts).

InBochs uses supervisor application which controls experiment flow from inside of guest OS. Its responsibility is to run AUT given number of times, gather data needed to distinguish currently running AUT instance and send them to InBochs, so it may inject faults just into AUT context. Originally, for Linux like guests, InBochs provided some kernel patches needed for supervisor application to perform that. However, if the AUT source code is available, an "AUT's self-registration" feature allows omitting this requirement, which is very important in cases like RTAI, where some kernel patches are already in place. Adding a call to UD2 processor instruction (opcode 0x0F0B) with some specific values in registers will "mark" the AUT within the SUT (InBochs can limit the faults only to the AUT context). When the AUT exits (or is killed) a supervisor informs InBochs about the test result. With that feature no modifications for a guest system are needed: providing a proper supervisor for particular guest platform should be sufficient for performing tests in InBochs.

InBochs supports several time abstractions for the guest OS. In particular, it can provide the real wall time or it provides the emulated time passage in respect to the particular guest OS execution speed. The second time abstraction in fact makes the InBochs capable to handle correctly the RTOS guests, as the guest is not aware of the time overhead of the system emulation. Thanks to that, regardless to the overheads introduced by the Bochs, the guest operating system runs smoothly with all the time related events occurring at the expected instants.

4. Fault sensitivity experiments

In experiments the explicit implementation of the Generalized Predictive Control algorithm (GPC) for a chemical reactor was examined (same as described in [6]). Injecting faults only the context of the control task was disturbed. The experiments were performed in two different RT operating systems: RTAI (Real Time Application Interface - an extension of the Linux kernel, which provides real-time functionality, including strict timing constraints [7]) and Phoenix [8].

RTAI implementation consists of two parts: kernel patches with loadable modules and set of programing tools. Kernel modifications are based on Adeos – a nanokernel HAL (Hardware Abstraction Layer) – which was designed to support hardware access from multiple OS instances [9]. Using it, RTAI can take over some interrupts and leave others to the standard kernel, which allows co-operation of the real-time and standard Linux applications on single machine. RTAI adds to Linux a dedicated scheduler, queues, signals etc. RTAI supports a wide range of platforms, including x86, x86_64, PowerPC, ARM, and MIPS. From the programmer's point of view RTAI provides library with a set of functions that allows turning standard Linux application into real-time application by adding a few lines (for initialization, starting timer, creating task, cleaning on exit etc.). Programs can be compiled to be run in kernel or in user mode. For kernel-mode they should be loaded as dynamic modules, which is a common solution in real-time Linux extensions. RTAI specific feature is to enable use of real-time applications in user-mode, which is the result of using Adeos and is achievable by RTAI-LXRT module.

Phoenix operating system was developed in the Institute of Computer Science of the Warsaw University of Technology in 2001 as the prototype of a real-time, time sharing operating system for embedded systems. The system has been implemented in IA32 architecture. A port for ARM7 is available. The commercial successor (Phoenix-RTOS) is devoted to configurable processors (e.g. EnSilica eSI-RISC) and will be used in industrial applications.

Examining the dependability of the explicit implementation of GPC [6], the source code of the controller as well as the model of the controlled chemical reactor process was compiled for both target platforms and verified against single bit flips injected randomly in time domain during the execution of the control algorithm task into the CPU registers and code memory ("AUT's self-registration" was used). The statistics of the observed fault effects or control quality was collected. The time overheads (due to the fault injection) from the point of view of the SUT are not noticeable – the whole "machinery" is on the host side of the InBochs.

Comparing the results with the ones reported in [6], several differences were observed. First of all, the controller code under RTAI seems to be not refreshed between the subsequent executions. As a result, the fault injected once was present in the code memory of the newly created tasks. Because of that, 99% of tests were classified as terminated by the OS due to unhandled exceptions. This is a very critical observation as any fault detection mechanism seems not to be present upon task creation in RTAI (e.g. verification of code image). Moreover, multiple instances of a task (if present) would experience a single fault in their code (as only single code instance exists). The code image sharing is commonly used; however, it creates additional threats from dependability perspective. Contrary, in Phoenix the same fault model resulted in 7%, 58% and 35% of tests in categories C, INC and S, respectively (see description at the end of chapter 2). Here, each creation of a task is preceded with clear image loading into the task memory space.

Other differences observed between the two analyzed RTOSs require deeper investigations. For instance, differences in results for faults in CPU registers were observed (e.g. 27% of INC tests in Phoenix comparing to 1% in RTAI). Additional note from the conducted experiments is that the applied SWIFI technique is applicable to detect any low level properties of the SUT and compare different RTOS properties.

5. Conclusions

The paper presents InBochs fault injector which is based on the pure target system emulator (Bochs). The advantages of such a system are very high flexibility in fault simulation (thanks to the InScript fault modeling language), universality (many target OSes may be easily examined for given target platform), high efficiency (targets are emulated on contemporary machine), controllability and observability (unlimited access to SUT resources and detectability of each task), and in imperceptible overheads to the SUT (time abstraction).

The experiments proved the InBochs as a valuable tool not only for task developers but also for RTOS designers, as it can trace down very internal problems of the systems (e.g. dependability mechanisms, stability against task failures, task fault robustness).

6. References

[1] Baumann R.C.: Radiation-induced soft errors in advanced semiconductor technologies. IEEE Trans. Device Mater. Reliab. 5, 305–316, 2005.

[2] Cabodi G., Murciano M., Violante M.: Boosting software fault injection for dependability analysis of real-time embedded applications, ACM Trans. On Embedded Computing Systems. ACM, Vol. 10, No. 2, December 2010.

[3] Gawkowski P., Sosnowski J.: Analysing system susceptibility to faults with simulation tools. Annales UMCS Informatica AI 4, ISSN 1732-1360, pp. 123-134, Lublin-Polonia, 2006.

[4] Sosnowski J., Gawkowski P., Lesiak A.: Fault injection stress strategies in dependability analysis. Control and Cybernetics, Vol. 33, No. 4, pp. 679-699, 2004.

[5] The Bochs project homepage, http://bochs.sourceforge.net/

[6] Gawkowski P., Ławryńczuk M., Marusak P., Tatjewski P., Sosnowski J.: Dependability comparison of explicit and numerical GPC algorithms. In M. Iskander et al. (Eds.), Technological Developments in Education and Automation, Springer Science+Business Media B.V., pp. 419-424, 2010.

[7] RTAI, Real Time Application Interface, https://www.rtai.org/

[8] Phoenix homepage, http://www.phoenix-rtos.com

[9] Adeos, http://home.gna.org/adeos/

otrzymano / received: 13.05.2011
przyjęto do druku / accepted: 04.07.2011 *artykuł recenzowany*