A Dependable Software Development Kit for Commercial Applications in Embedded Systems

Alfredo BENSO, Silvia CHIUSANO, Paolo PRINETTO

Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24
I-10129 Torino, Italy
Email: {benso, chiusano, prinetto}@polito.it
http://www.testgroup.polito.it

Abstract[1]

In this paper we present a set of tools designed to support the software engineer in releasing dependable applications for embedded systems requiring commercial software. We propose three different tools: WRAP, a tool able to wrap a set of software modules that transparently enhance the dependability characteristic of any executable software, EXEM, an external world and device emulator tool, and BOND, a powerful Fault Injection environment able to emulate the occurrence of different types of faults within the system, in order to evaluate the dependability properties of the overall embedded system. Experimental results gathered on an embedded system running Embedded NT OS demonstrate the effectiveness and the usability of the tool.

1. Introduction

The swift advance of semiconductor technology and the high degrees of quality and performances required by the market are making the design of dependable digital systems a key problem from an economical point of view. Embedded system, including hardware and software components, are now commonly used in even many traditionally non-electronic products. To reduce the cost of design, development, and production, the hardware modules are usually based on embedded cores, i.e., reusable previously designed functional blocks (IP blocks). Moreover, to keep the costs at the lowest possible level, hardware-based fault-tolerant architectures are usually avoided. It is then straightforward that, in order to still guaranteeing the quality levels required by the market, the dependability issue has to be moved at the software layer, now in charge of also guaranteeing high levels of availability and maintainability.

Many researchers [1][2] have shown that, in computer-based systems, a high percentage of faults cause a *Fail-Silent Violation* behavior, e.g., the system produces incorrect results while the application seems to terminate correctly. This behavior is mostly caused by pure data errors, i.e., errors appearing in memory locations storing the data or in the microprocessor's user registers. For the sake of clarity, a computer-based system is said to be *Fail-Silent* if its outputs produce only corrects results only, i.e., the system does not output incorrect results even if they are generated internally as a consequence of a fault. In this context, the main goal is to ensure that a software application either outputs correct results, or detects, and possibly corrects, a faulty behavior of both the hardware and the software components of the system. Unfortunately, commercial and also custom software components are usually not designed, individually, to guarantee the required dependability levels. Essential to this task are therefore tools that support the software engineer in the insertion and integration of the software components into the system, allowing to

[1] This work was partially funded by the Italian Space Agency (ASI) under Research Grant #332, and by Politecnico di Torino (Italy) under the project Progetto Giovani Ricercatori 2000 "*Development of a software environment for highly dependable space applications*".

continuously monitor and evaluate the dependability properties of the system itself.

In this paper we present three tools able to support the integration of commercial software components into an embedded system, and to perform Fault Injection experiments into the system in order to evaluate its dependability level.

The four different tools are presented in Figure 1. When the C/C++ source-code of a software application is available, its dependability characteristics can be significantly increased running RECCO (*RE*liable *C*ode *CO*mpliler), a Source-to-Source C++ compiler based on *code re-ordering* and *variable duplication* techniques coupled to detect run-time *hardware errors*. After being elaborated by RECCO, the reliable source code can be compiled using any commercial compiler to obtain an executable code with increased fault-tolerance characteristics. A detailed description of this tool can be found in [3]. In this paper we shall therefore address the other three tools, which address the case of commercial software to be integrated into the embedded system.

The first of the mentioned tools is WRAP (application *WRAP*per). This application can be exploited to wrap around the user application a set of software modules that transparently enhance its dependability properties, for example encoding and decoding all the file systems data, or adding redundancy to data residing in critical memory regions.

The second tool is EXEM (*EX*ternal world *EM*ulator), which has been designed to aid the development and the integration of software that requires a heavy interaction with the outside world. EXEM is able to emulate the behavior of the outside world in terms of communications through the I/O channels of the system.

Finally, BOND is a Software Fault Injection tool based on interposition techniques, able to emulate many types of abnormal behaviors of a computer system. BOND allows performing both statistical and deterministic Fault Injection experiments, trading-off between overhead and precision of the obtained results. Moreover, BOND is capable of injecting faults into different locations, at any level of the application context (code and data sections, stack, heap, processor's registers, system calls, …).

The effectiveness and the flexibility of the tools has been assessed running a significant number of Fault Injection experiments on an Embedded System designed for space applications and running Embedded NT as Operating System.

The paper is organized as follows: the three tools are presented in Section 2, whereas a complete set of experimental results on different application programs demonstrates their effectiveness and flexibility is presented in Section 3. In Section 4 we finally draw some conclusions.

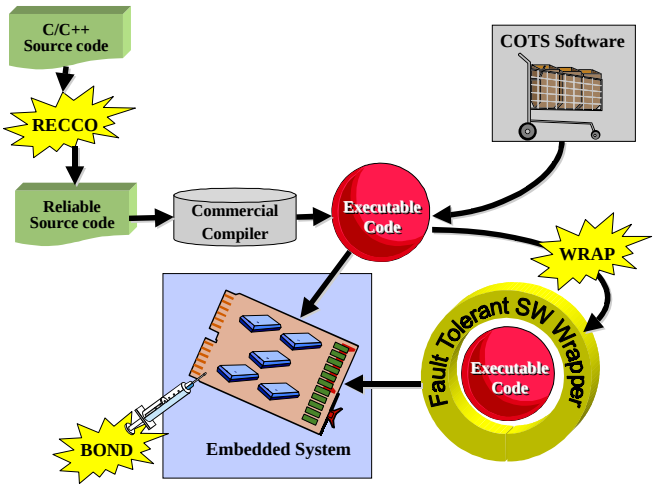

Figure 1: The Software Development Kit overview

2. The Dependable Software Development Kit

In the following subsections, we detail the main features and functionality's provided by the three tools proposed in his paper.

2.1. WRAP: the Software Wrapper

The WRAP tool uses the idea of *interposition agents* to address the problem of the system dependability and fault tolerance. *Interposition* is the process of capturing events crossing an interface boundary and forwarding these events to an interface extension (the *Agent*). Interposition agents are software components that are conceptually placed between the user application and the Operating System and communicate with both of them [4] (Figure 2). Other interesting researches about interposition techniques can be found in [5], [6], [7], [8], and [9]. WRAP is able to insert a Fault Tolerant Software Wrapper around any running application, including commercial software, completely transparently to both the target application and the Operating System. The Wrapper allows achieving a high degree of software-based reliability in a wide variety of domains (such as file system and inter-process communication) using a uniform technique.

In particular, the WRAP is able to transparently encode and decode all the data stored in the *process pipes*, *file system*, and *mail-slots*. Moreover, it allows adding redundancy to the data residing in certain

memory regions, detecting and possibly correcting hardware transient faults. This is particularly useful in high-radiation environments, where the most common radiation effect is the Single Event Upset (SEU), i.e., a change in the status of a single bit in a memory location. The complexity and hence the power of the coding scheme can be chosen by the user, any aspect of the hardware system being affected at all.

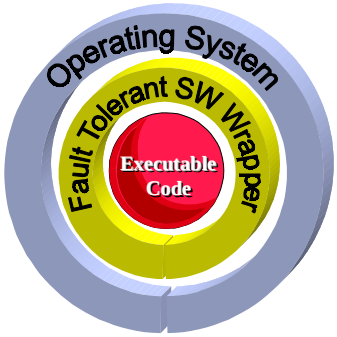

Figure 2: The Software Wrapper

The Software Wrapper inserted by the WRAP tool can provide the following capabilities: the *Memory Coding* and the *File System Coding*.

The idea behind the *Memory Coding module* is to intercept the set of memory accesses made by the target application during its execution, and modify them to improve the system fault tolerance. Data written to and read from selected segments of the user memory are transparently coded and decoded in order to detect and possibly correct soft and hard hardware faults. In particular, when the Wrapper detects a memory access, it is able to retrieve the address of the memory access causing that exception and to perform the data coding or decoding depending on the requested operation type. The Memory Coding module currently supports two kinds of codes: a simple repetition code and an extended Hamming code.

The *File System Coding* capability allows transparently encoding and decoding all the communication involving a *File System* (e.g., RAM disks, hard drives, pipes). It is able to on-line check the data integrity and possibly correct the detected errors. The agent is able to selectively protect file system areas used by an application, choosing the coding scheme that is more suited to the fault tolerance level required (detection, correction, or more sophisticated coding schemes). The agent is currently able to handle a repetition code or an extended Hamming code; nevertheless, a non-systematic code (e.g., a convolutional code) may be used as well. When the application software performs an I/O operation on one of the mentioned system resources, the File System Coding module executes a modified functions codifying the data before the real OS function is actually called or returns. Data are encoded on writes and decoded on reads by the Wrapper, in such a way that the entire process is completely transparent to the target application. The user can choose the coding scheme implemented by the Wrapper before actually wrapping the target application. Using this capability to install commercial software it is possible to have a fully running commercial application that uses coded files (including configuration and other application specific files). We succeeded in installing several commercial applications and in running them properly.

2.2. EXEM: the External world emulator

EXEM is a very powerful tool able to emulate the behavior of the outside world in terms of communications through the I/O channels of the system. Since the behavior of the emulated device is fully programmable in C++, the module can be effectively exploited for different purposes:

- *Application Development*: the module can be used to simply emulate external devices. Such a capability proved to be extremely helpful in the early phases of the application development process, when the actual target external device has been fully specified but it is not yet available. We successfully exploited this feature to support the development of control software for a scientific experiment that will be installed on a space station. The software controls a “weight device” to be used in a 0-gravity environment and receives the input data from a set of optical serial data acquisition channels. At design time, since the device and the optical channels were not yet available, EXEM has been implemented and successfully exploited to emulate the DAQ channels and the device behavior, thus allowing a realistic calibration of the control software.
- *Dependability Evaluation*: the module can be used to emulate faults occurring in the communications with the outside world through serial or parallel interfaces, network devices, etc., in order to evaluate the fault tolerance of the system to such kind of errors.
- *Coded Communications*: when used at both end of a communication channel (e.g., two embedded systems communicating on a serial interface or over a LAN), EXEM can be exploited to code (or encrypt) the exchanged data completely transparently to the communicating applications without any modifications to the program source code.

2.3. BOND: the Fault Injector

As the WRAP tool, the Fault Injector is based on interposition techniques, which guarantees a low impact on the execution of the target program, and allows the injection of Commercial off-the-Shelf software programs. The insertion of artificial faults into a real or simulated computer system under test is intended to yield to three major benefits: an understanding of the effects of real faults, feedback for system correction or enhancement, and a forecast of expected system behavior. A great advantage of the Fault Injection technique is that the effects of faults in the system can be studied when it is executing realistic programs.

BOND provides unique characteristics in terms of *Fault Model*, *Fault Injection location* and *time*, and *Fault Effects* classification:

- *Fault Model*: the type of injected fault is, by default, the Single Event Upset, i.e., the Single Bit Flip fault, but the injector can be extended to inject any type of user-defined fault model.
- *Fault Injection Location*: The Injector can modify all the areas of allocated memory of the injected process (committed memory). These areas include stacks (one for each thread), heaps (both main process heap and user-created heaps), code and data of any loaded image (executable images include main EXE and dynamic libraries). Additionally, the Injector is able to inject faults in all the processor's registers.
- *Fault Injection Time*: BOND allows a *Statistical* injection, where the injection time is expressed as the elapsed execution-time from the beginning if the target application, and a *Deterministic* injection, where the injection triggering event is based on the count of some events (e.g., the number of executed instructions). The Statistical injection allows a very low overhead in terms of execution time of the target application obtained at cost of a non-reproducibility of the exact experiment. On the contrary, the Deterministic Injection is quite expensive in terms of execution time overhead but guarantees a full reproducibility of the experiments.
- *Fault Effects Observation*: BOND provides a very detailed analysis of the effects of the injected faults. In particular, after the injection, the tool classifies the fault effects in one of the following categories:
  - *Fail Silent (FS)*: the results produced by the target program are correct. No faulty behavior is detected.
  - *Fail Silent Violation (FSV)*: the program terminates correctly but the results are not correct. This is the worst case since the user is not able to distinguish between faulty and correct behavior of the application.
  - *Detected*: the injected fault either caused a system exception or triggered a user error detection mechanism, and is therefore detected.

2.4. The Graphical User Interface

As shown in Figure 3, we implemented a very user-friendly interface, which allows the user to run the different tools, wrapping commercial applications, emulating external devices, or setting up Fault Injection experiments.

Figure 3: The Graphical User Interface

3. Experimental results

In the following we present different sets of experimental results gathered to demonstrate the effectiveness and the flexibility of the presented tools. The considered target embedded system is a PC-based platform, running Embedded Windows NT on a Radstone PENV1 hardware hosting an Intel Pentium processor. The system, which will control a scientific experiment, will be installed on Columbus, an international scientific space station. The platform is a typical example of COTS based environment, where neither the hardware modules nor the software are certified for space applications. The Dependable Software Development Kit presented in this paper will support the integration of the software modules within the system. The project is funded by the Italian Space Agency and is developed in cooperation with the Italian most important aerospace company, Alenia Aereospazio.

To evaluate the performances of the different tools, it has been necessary to analyze the system behavior in presence of faulty data. We therefore executed the benchmark programs under the control of BOND to emulate the appearance of data faults

corrupting memory locations storing data. Each experiment involved the injection of a fault list including 1,000 faults whose location and injection time have been randomly selected. BOND classified the system reaction to the injected faults in one of the following categories: *Fail Silent* (FS), *Fail-Silent Violation* (FSV), or *Detected* thanks to the introduced fault-tolerant mechanisms.

As target software applications, we used two different sets of benchmarks:

- *A custom benchmark set* that includes three programs, a *Matrix Multiplication* algorithm and a *Floating Point* benchmark, whose source code is available[2]. This benchmark is used to demonstrate the improvements introduced by WRAP on the executable code of the applications.
- *A commercial benchmark set* that includes two commercial software applications (Excel 2000 and Word 2000) and is used to show the flexibility and the performances provided by the WRAP and BOND tools.

3.1. WRAP

In order assess the performances provided by the Software Wrapper introduced by WRAP, we executed a set of Fault Injection experiments on a "wrapped" version of the Custom Benchmark set. In this experiment, the only module inserted in the Wrapper was the Memory Coding module implementing both a simple *repetition code*, and then an *error correcting code*. In the first case, the results produced by the application were still faulty (FSV) but the Wrapper was always able to notify the user the misbehavior of the target programs. The percentage of detected faults is usually higher than the FSVs because the Wrapper was able to detect (and then to notify) also the appearance of faults that resulted in a FS behaviors. When implementing the error correcting code, the module was able to detect and correct all errors, resulting in a 100% of Fail Silent behaviors. Nevertheless, in the case of the Custom benchmark set, the execution time overhead introduced by the Wrapper can be extremely high (between 1000 and 10,000 times).

The performances of the Memory Coding module were much better when dealing with the Commercial benchmark set. This is due to the fact that the Custom benchmark set included only performance-oriented benchmarks, whereas the Commercial set includes programs with different characteristics, which do not extensively stress the Software Wrapper. We measured the Agent overhead in terms of execution time on Microsoft Word 2000.

Table 1: Memory Coding on the Custom Benchmarks

	Error Detection			Error Correction		
Benchmark	FS	FSV	Detected	FS	FSV	Corrected
Matrix Mult.	65 %	35 %	40 %	100%	0	100 %
Floating Point	74 %	26 %	33 %	100%	0	100 %

The Word-based benchmark used for the evaluation runs an auto-executing macro that opens a document, displays it, saves it with another name, and then closes Microsoft Word. The document contains text as well as images and it is about 2.1 Megabytes in size. The data for a normal execution (without agents), and for differently wrapped executions are in Table 2. The table displays values resulting from an average on 10 distinct executions of the Microsoft Word benchmark discussed above.

Table 2: Agent Execution Time Overhead

	Average execution time
No Wrapper	1542 ms
Debug Events Logger module	1586 ms (+2,8%)
File System Coding module	8726 ms (+465%%)
Memory Coding module	37369 ms (+2,323%)
All modules	192012 ms (+12,352%)

Even if the average execution time may appear still high, it is important to notice that the current version of the Software Wrapper has been implemented without taking into account memory or execution time overhead. In this preliminary phase, all the efforts have been put in evaluating the effectiveness of the approach from the fault-tolerance point of view. Developing the Wrapper at user level obviously implies a high time overhead, since all the interactions between the OS kernel and the debugger module involve level switch, i.e., a passage from ring 0 execution (privileged mode) to ring 3 execution (user mode) and vice versa. The implementation of the Wrapper using kernel-mode programming, i.e., as Windows device Drivers, is under development and preliminary experimental results showed a drastic reduction of the execution time overhead.

The last set of results has been obtained to test the effectiveness of the File System Coding module. As a target program we used Microsoft Excel 2000 running a simple macro that performs some computations on a worksheet. Results are presented in Table 3. When the File System Coding module implementing a simple error detection scheme wrapped Excel, the behavior was unchanged, but the Wrapper was able to detect 100% of the faults. Therefore, it never happened that a

[2] The source code of the Custom Benchmark set can be found at http://www.testgroup.polito.it/download_material.html

faulty file was opened without a notification to the user. Instead, when a fault correction scheme was implemented, Excel did not notify any failure in opening the file since the injected errors were always promptly corrected.

Table 3: The File System Coding module wrapping Excel 2000

	FS	FSV	Faults detected by Excel	Faults Detected by the Wrapper
Without Wrapper	77%	16%	7%	-
Fault Detection	77%	16%	7%	100%
Fault Correction	100%	0%	0%	100%

4. Conclusions

In this paper we presented a Dependable Software Development Kit able to support the integration of commercial software components into an embedded system, and to evaluate its dependability. The problem has been addressed implementing a set of tools able to add very powerful fault tolerant capabilities to a target embedded system. We presented different sets of experimental result showing the effectiveness and the flexibility of the environment, particularly from the fault tolerance point of view.

In the paper it has been shown how the tools can be exploited at different stages of the product life cycle:

- *In field Fault Tolerance*: When running commercial software, the Fault Tolerant Software Wrapper provided by WRAP allows to transparently co-decode memory and file accesses.
- *Development and Dependability Validation*: tools like BOND and EXEM can be used to support the development and validation of complex applications. They can be exploited to analyze the correct responses to critical conditions or to emulate the behavior of the outside world before the application is actually installed in its final environment.

5. Acknowledgments

The authors would like to thank Emiliano Bacci, Andrea Baldini, Federica Faggiani, Marco Sambin, and Luca Tagliaferri for their hard work and excellent contributions given to this research while working for their Master thesis in EECE at the University of Illinois, Chicago.

6. References

[1] A. M. Amendola, A. Benso, F. Corno, L. Impagliazzo, P. Marmo, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, *Fault Behavior Observation of a Microprocessor System through a VHDL Simulation-Based Fault Injection Experiment*, EURO-VHDL'96, September 1996, Geneva (CH), pp. 536-541

[2] J. G. Silva, J. Carreira, H. Madeira, D. Costa, F. Moreira, *Experimental Assessment of Parallel Systems*, Proc. FTCS-26, Sendaj (J), 1996, pp. 415-424

[3] A. Benso, S. Chiusano, P. Prinetto, L. Tagliaferri, *A C/C++ Source-to-Source Compiler for Dependable Applications*, IEEE International Conference on Dependable Systems and Networks (FTCS 30), New York (NY), June 2000

[4] D. P. Ghormley, S. H. Rodrigues, D. Petrou, T. E. Anderson, *SLIC: An Extensibility System for Commodity Operating Systems*, In Proc. of the USENIX Annual Technical Conference, June 1998, New Orleans (LO)

[5] Michael B. Jones, *Interposition Agents : Transparently Interposing User Code at the System Interface*, In Proc. of the fourteenth ACM symposium on Operating systems principles, December 1993, Asheville (USA), pag. 80-93

[6] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser and Chris J. Scheiman, *Ufo: A Personal Global File System Based on User-Level Extensions to the Operating System*, in ACM Transactions on Computer Systems, Vol. 16, No. 3, August 1998, pages 207-233

[7] A.K. Ghosh, M. Schmid, F. Hill, *Wrapping Windows NT Software for Robustness*, in Proc. Of the 29th IEEE Fault Tolerant Computing Symposium, Madison (WI), USA, June 1999

[8] P. Narasimhan, L. E. Moser, P.M. Melliar-Smith, *Using Interceptors to Enhance CORBA*, IEEE Computer, Vol. 32, N. 7, July 1999

[9] S. Bagchi, K. Whisnant, Z. Kalbarczyk, R. K. Iyer, *Chameleon: A Software Infrastructure for Adaptive fault Tolerance*, in Proc. of the 28th Fault Tolerant Computing Symposium (FTCS-28), June 1998, Munich.