Evaluating System Dependability in a Co-Design Framework

M. Lajolo
Politecnico di Torino
Dip. Elettronica
Torino, Italy

M. Rebaudengo, M. Sonza Reorda, M. Violante
Politecnico di Torino
Dip. Automatica e Informatica
Torino, Italy
http://www.cad.polito.it/

L. Lavagno
Università di Udine
DIEGM
Udine, Italy

Abstract

The widespread adoption of embedded microprocessor-based systems for safety critical applications mandates the use of co-design tools able to evaluate system dependability at every steps of the design cycle. In this paper, we describe how Fault Injection techniques have been integrated in an existing co-design tool and which advantages come from the availability of such an enhanced tool. The effectiveness of the proposed tool is assessed on a simple case study.

1. Introduction

Embedded systems are widely used for performing safety-critical tasks, where failures can cost lives and/or money. Two key issues arise when designing these systems: time-to-market and dependability assessment.

The need to shorten time-to-market has motivated intensive research efforts which result is the co-design approach to embedded design. Co-design is an appealing solution since it allows designers to concentrate their efforts on defining the system behavior and analyzing alternative hardware/software implementations, while the system is automatically synthesized by the co-design tool.

Several approaches to co-design have been proposed, differing on the specification language, the mechanisms available to perform design space exploration, and the supported target architecture. In particular, [1] and [2] target control-dominated applications and system specification, while [3] is oriented to data-dominated applications.

As far as dependability assessment is concerned, it is usually addressed resorting to Fault Injection [4]. However, up to now Fault Injection techniques have been mainly applied to real systems (in their final or prototypical version) or when a low-level description is available.

When designing a safety-critical application the gap between the level of abstraction at which the design is carried out, and the level at which dependability is assessed is becoming unacceptable. If a safety-related design pitfall is discovered in a late phase, a complete redesign spin is required, resulting in a dramatic increase of the time-to-market. Design centers involved in the design of safety-critical systems are facing this problem, and some approaches have been proposed to cope with it [5][6] adopting Fault Injection to analyze the behavior of the system during co-design.

The basic idea is to move the dependability evaluation step from the bottom of the design flow to the top, thus providing designers with tools able to evaluate dependability during the system level design [5] through Fault Injection. The ability to inject faults immediately after the system behavior specification allows an early validation of the fault detection and tolerance mechanisms of the system, as well as an early identification of possible safety-related pitfalls in the system behavior.

When reasoning at the behavioral level, designers lack the details of the physical implementation of the system, and fault models exactly matching the behavior of all real faults can hardly be identified. In the best case, faults defined at the higher level correspond to a subset of the physical faults. Under this hypothesis, an analysis of the system carried out using a behavioral fault model can be fruitfully exploited for early evaluating the system dependability, and for verifying the correctness of the system. When proceeding through the design process, further Fault Injection experiments are likely to be performed on lower level descriptions to obtain more confidence in the system fault tolerance.

Fault Injection is a powerful approach to cope with this need; it can indeed be exploited at every level of abstraction, provided that a suitable fault model is identified, and effective techniques are available. Two kinds of Fault Injection techniques can be of interest to the designer: focused and statistical [6]. Focused Fault Injection consists in injecting carefully selected faults to validate either the behavior or the implementation of a module in a safety-critical system. Conversely, statistical Fault Injec-

tion consists in injecting statistically selected faults, and is intended to validate the whole system. Moreover, statistical Fault Injection carried out on the final architecture with a large set of faults can be used to measure the system reliability.

In the last years, several approaches to system-level dependability analysis have been proposed (e.g., [7] and [8]). The main drawback of these approaches is that they are not integrated in a design environment. Conversely, the full integration of Fault Injection features in a co-design tool leads to an environment where dependability is a design space dimension, along with performance, area, and power consumption. For example, the way of describing the system used during dependability evaluation must be the same used during system architecture design and hardware/software synthesis.

Following the above considerations, we describe a co-design tool which has been enriched with fault tolerance-oriented facilities, in particular with the possibility of performing Fault Injection experiments. The tool we developed allows analyzing:

1. *the system behavior dependability*: given a formal specification of the system behavior, the designer analyzes how the system reacts in presence of faults. This analysis allows to validate the fault detection mechanisms the system embeds, and it allows identifying possible critical conditions the given behavior is not able to detect;
2. *the system architecture dependability*: when the system behavior has been debugged and fixed, the designer analyzes the dependability of the architecture implementing it. This allows selecting the architecture that best fits the design constraints of speed and power consumption while preserving the required level of dependability.

We have defined a new fault model for the experiments at the behavior level. The behavioral fault model is used to perturb the system behavior, and it has to be coherent with the adopted co-design tool. In the current implementation of our methodology, we have integrated Fault Injection in POLIS [1].

The tool we developed, described in Section 2, is thus able to effectively support a design flow where dependability can be evaluated at every design step. To assess its effectiveness a case study is presented in Section 3. Finally, in Section 4 some conclusions are reported.

2. Proposed approach

The co-design environment we have considered in our work is POLIS, which has been developed at the University of California Berkeley in collaboration with Politecnico di Torino and several industrial partners such as Cadence Design Systems and Magneti Marelli.

2.1. Background

The internal model of computation of POLIS is based on Co-design Finite State Machines (CFSMs) which are extended finite state machines with asynchronous buffered communication. The choice of an asynchronous type of interaction between CFSMs is due to the need to support a *neutral* high-level specification of hardware and software components in which the execution delay of a CFSM transition is unknown a priori. It just assumes an unbounded delay for the software response and the communication between CFSMs is not performed with shared variables (as in the classical composition of FSMs), but by means of events. An event is a unidirectional communication that can be sent and received, and that can carry a value.

POLIS can synthesize both software and hardware implementations for each CFSM, as well as a Real-Time Operating System that coordinates the execution of the software CFSMs (one task for each CFSM).

One important characteristic of the software POLIS synthesizes, from a CFSM, is that the model of execution of CFSMs is globally asynchronous, but locally synchronous. This means that the (acyclic) path of execution for a given invocation of each task is fully determined by the presence/absence status of its input events, and by their value. This provides an easy mechanism for linking low level simulators (e.g., an Instruction Set Simulator for the target processor and/or a gate level simulator for the hardware) to the functional simulator.

2.2. Behavioral fault injection

Behavioral fault injection is intended to analyze the behavior of the system in presence of a transient fault perturbing the system. The fault model has to be coherent with the description of the system in order to perturb its behavior and to easily fit in the adopted design tool.

Given the model of computation of POLIS we can perturb the system by modifying the sequence of events the CFSMs detect and emit, the value associated to events and the values stored in the CFSMs internal variables.

We have therefore devised the following *behavioral fault model*:

1. faults on output events:
   a. suppress an output event when it has to be emitted
   b. change the value associated to the output event before emitting it
   c. emit an output event in an erroneous time

2. faults on input events:
   a. suppress an input event when it should has been detected
   b. change the value associated to the input event
   c. detect an input event that has not been emitted
3. faults on internal variables: change the value stored in an internal variable.

In POLIS variables and events at the behavioral level corresponds to registers or memory locations at the architectural level, therefore we expect a relatively strong correlation between the adopted fault model and real physical faults affecting memory elements.

A complex system is usually described as several interconnected CFSMs. A fault is therefore identified by the following information:

- the CFSM on which the fault has to be injected (*target_CFSM*)
- the signal (input event, output event or internal variable) affected by the fault (*target_signal*)
- the value to be injected in the selected signal (*faulty_value*)
- the time at which the fault has to be injected (*fault_time*).

Our approach assumes that when affecting input events or internal variables the fault is injected immediately before the execution of a CFSM. Conversely, when the fault affects output events, it is injected immediately after the simulation ends, and before the simulation results are committed to other CFSMs. Faults are thus injected at the CFSM boundary, only. This approach allows to effectively analyze the effects of transient faults with a low simulation performance penalty. Moreover, from the implementation point of view it does not require to modify the mechanism exploited by POLIS to execute the CFSMs.

2.3. Architectural fault injection

At the architectural level the system is modeled as a single cache-free CPU with an external memory storing the program and the data of the software partition, and a set of hardware components connected to the CPU via a bus, and among each other by dedicated connections.

For fault injection experiments we can thus exploit the most commonly adopted fault models. Currently, we support the transient single bit-flip fault model in the memory cells storing the code and the data of the software partition and in the registers of the CPU and of the hardware partition.

The hardware partition is simulated as a digital circuit that produces its output in a single clock cycle. All the inputs and the outputs of the circuit are buffered through registers. Conversely, the software partition is simulated resorting to an Instruction Set Simulator (ISS) that simulates CPU and memory behavior during program execution.

We have modified the simulation routine of POLIS so that when injecting a fault in the hardware partition of the system, at *injection_time* the register storing the *selected_signal* is modified before executing the target CFSM.

To inject faults in the software partition of the system, we have instrumented the ISS so that at *injection_time* we can inject the *fault* in any locations of the memory and the CPU registers.

2.4. Analyzing the results

To observe the effects of a single transient fault with respect to a given sequence of input stimuli, we store the outputs in a trace file. Moreover, to analyze the system behavior we sample the state of the system at a given sample frequency: the values of all observed variables and events are stored in a trace file. This approach allows measuring the fault latency with an accuracy that depends on the sampling frequency. By adopting a sampling strategy, we limit the disk space required for the trace file. Nonetheless, we are able to trade-off accuracy and disk space by acting on the sample frequency.

By analyzing the simulation trace, we categorize the fault effects according to the following classification:

- *no effect*: the transient fault does not modify the system behavior; the system produces the expected outputs and reaches the expected state;
- *latent error*: the fault affects the system state but it does not reach the system outputs, thus the simulation ends with the system producing the expected outputs with an unexpected state;
- *failure*: the fault affects the system behavior and produces an error on the system outputs
- *time out*: the outputs cannot be produced in a given time.

2.5. Fault Model Discussion

The transient single bit-flip is often adopted as a fault model when working on lower level descriptions, mainly due to its closeness to real faults. Moreover, other fault models can be supported by simply allowing the injection of multiple bit-flips.

When moving to behavioral-level descriptions, new fault models have to be devised, taking into account that the higher abstraction level obviously makes more difficult to match the behavior of real faults.

A detailed analysis of the fault model we propose for the behavioral level (not reported here for lack of space) shows that, due to the synthesis rules implemented in POLIS, each behavioral fault is equivalent to a single or

multiple bit-flip at the architectural level. This observation is crucial to support the usefulness of performing Fault Injection experiments at the behavioral level, since it proofs that they allow verifying the correctness of the implemented fault tolerance mechanisms with respect to a subset of the faults adopted at lower levels. Passing the test corresponding to a behavioral-level Fault Injection campaign is therefore a necessary condition for going forward in the design process. Although the full confidence in the ability of the system under design to cope with the whole set of possible faults can only be achieved by repeating the Fault Injection experiments at the architectural level, the ability to start the validation process early in the design process can be crucial in reducing its cost as a whole.

3. A case study

To evaluate the proposed approach, a case study has been implemented and analyzed. We designed a receiver and transmitter system for the Internet Protocol (IP) transmission.

3.1. System description

The system, depicted in Figure 1, is composed of three operative modules, **create_pack**, **ip_check** and **checksum**. The modules access to a shared memory, **mem**, through an arbiter, **arbiter**. The module **create_pack** stores in **mem** a bit stream coming from a network cable. When all the required bits have been loaded, the module **checksum** computes a checksum code and passes it to **ip_check** that compares it with the one received by the net and already stored in **mem**. If the two values match, a new bit stream is loaded, otherwise an error signal is activated, and the system requests the sender to re-transmit the former message. The module **mem** exploits error correction codes, thus single transient faults have no effect on it. Moreover, the address and data buses exploit a word-level error correction code, which guarantees detection and correction of single bit transient faults. All the modules have been initially described at the behavioral level in terms of interacting processes, and then synthesized into corresponding hardware or software components at the architectural level.

Based on a comparative evaluation of the different partitioning solutions, we found that the cheapest architecture (in terms of required area) still compatible with the time requirements corresponds to implementing only **ip_check** as a software module.

The system is described in POLIS as 6 CFSMs, and amounts to 565 lines of ESTEREL code and 248 lines of C code. Fault injection experiments have been performed on a Sun UltraSparc 5 equipped with 256 Mbytes of RAM. We experimented that 40 seconds of CPU time are required to inject a fault at the behavioral level, while at the architectural level 65 seconds are required. For the purpose of this paper, both input stimuli and faults used for fault injection experiments are supposed to be selected by the designer.

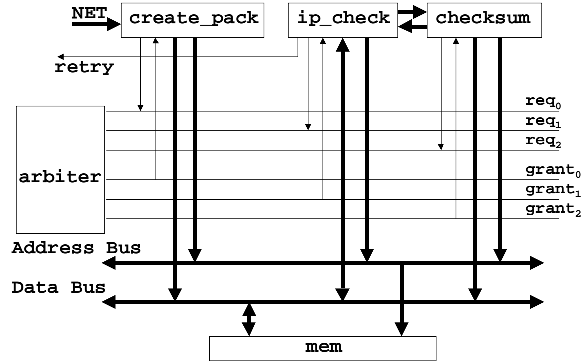

Figure 1: *The case study*

3.2. Fault tolerance evaluation

We analyzed the system by performing focused behavioral level Fault Injection on each module. We were first interested in analyzing how the detection mechanism provided by the IP protocol itself works in presence of transient faults. Moreover, we were interested in analyzing the effects of transient faults on the behavior of each module and on the communication between modules. We injected an amount of 1,000 faults, whose effects are summarized in Table 1.

Module	No Effect [%]	Failure [%]	Time Out [%]	Latent [%]
create_pack	85.80	0.00	11.90	2.30
ip_check	75.60	8.50	13.80	2.10
checksum	86.20	2.10	11.70	0.00
arbiter	83.70	0.00	12.40	3.90

Table 1: *Fault Injection results*

In our case evaluation, the time-out situation always happens when a fault corrupted a message issued by a module that is idle waiting for an acknowledge from another module. This observation suggested that the communication protocol between the modules has to be hardened to cope with transient faults, and that some of the modules in the system had to be redesigned in order to attain a sufficient degree of fault tolerance.

The majority of the faults we injected (68%) degrades the system performance, as they are detected as transmission errors and thus retransmission requests are issued. As a consequence, the throughput of the system is reduced.

3.3. Fault tolerant case study

Given the above considerations we modified the system as follows:

- we added a watchdog timer to detect the time-out situations;
- the **arbiter** is the most critical module in the system, therefore we replicated it three times and we added a voter (TMR architecture)
- we duplicated the module **checksum**: if the outputs produced by the two replicas do not match, a retransmission request is issued
- we adopted a temporal redundancy for the module **ip_check**: it performs its computation twice and issues a retransmission request when the two computed values mismatch.

The fault tolerance of the new version of the system was evaluated by performing a Fault Injection campaign in which a total of 1,000 faults were injected. The campaign was first performed at the behavioral level, and then repeated at the architectural one. During the whole campaign we did not identify any fault causing a failure or hanging up the system (i.e., triggering a time out condition), therefore significantly improving the confidence in the correctness of the adopted solution.

At the same time, by exploiting the features provided by POLIS we evaluated the performance and area occupation of both versions of the system. In Table 2 the hardware overhead (number of gates) and the software requirements (bytes of memory) are reported for the original system and for the fault tolerant one.

Module	Original System		Fault Tolerant System	
	Type	*Size [#gates/ #bytes]*	*Type*	*Size [#gates/ #bytes]*
create_pack	hw	1,092	Hw	1,092
ip_check	sw	1,040	Sw	1,040
checksum	hw	2,134	Hw	4,268
arbiter	hw	1,198	Hw	3,594
voter			Hw	35
watchdog			Hw	55

Table 2: *Resource requirements*

When evaluating the system partitioning, we found that the voter used by the fault tolerant arbiter had to be implemented in hardware due to timing constraints of the communication protocol. As far as the overhead is concerned, we observed that the hardware and software partitions of the resulting architecture are about 2 times larger than the original ones. Finally, we evaluated the impact of the redundancy on the system performance. The original system receives and processes three packets (for a total amount of 1,024 bytes of data) in 107,305 clock cycles. The modified system requires 107,569 clock cycles to perform the same operation. We are thus able to obtain a dependable system with a tight control of resource overhead and with negligible performance degradation (less than 1%).

4. Conclusions

The need for early evaluation of system dependability is becoming an issue for designers involved in the design of safety-critical embedded systems.

We have described a tool able to support a co-design methodology where Fault Injection is used to assess system dependability both at the behavioral level and at the architectural one.

The analysis of a simple case study has demonstrated the effectiveness of our tool as well as that of the adopted design methodology.

5. References

[1] F. Balarin, E. Sentovich, M. Chiodo, P. Giusto, H. Hsieh, B. Tabbara, A. Sangiovanni Vincentelli, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, *Hardware-Software Co-design of Embedded Systems*, Kluwer Academic Publishers, 1998

[2] A. Balboni, W. Fornaciari, D. Sciuto, *TOSCA: A pragmatic approach to co-design automation of control-dominated systems in Hw/Sw codesign*, in Hardware-Software co-design, G. De Micheli, M. G. Sami, Kluwer Academic Publishers, 1996

[3] R. Ernst, J. Henkel, T. Benner, *Hardware-Software Co-synthesis for Microcontrollers*, IEEE Design & Test of Computers, pp. 64-75, 1993

[4] J. Clark, D. Pradhan, *Fault Injection: A method for Validating Computer-System Dependability*, IEEE Computer, June 1995, pp. 47-56

[5] R. von Hanxleden, A. Botorabi, S. Kupczyk, *A Codesign Approach for Safety-Critical Automotive Applications*, IEEE Micro, 1998, no.5, pp. 66-79

[6] Y. le Guédart, L. Marneffe, F. Scheerens, J. P. Blanquart, T. Boyer, *Functional and Faulty Behavior Analysis: Some experiments and Lessons Learnt,* Proc. IEEE Int'l Conf. Fault-Tolerant Computing Systems, 1999, pp. 348-351

[7] K. Goswami, R. Iyer, L. Young, *DEPEND: A Simulation-Based Environment for System Level Dependability Analysis*, IEEE Transactions on Computer, Volume 46, Number 1, Jan. 1997, 60-74

[8] J. Carreira, H. Madeira, J. Silva, *Xception: Software Fault Injection and Monitoring in Processor Functional Units*, DCCA-5, Conference on Dependable Computing for Critical Applications, 1995, pp. 135-149