

FITS – A Fault Injection Architecture for Time-Triggered Systems

René Hexel

School of Computing and Information Technology
Griffith University,
Nathan, QLD 4111, Australia
Email: rh@cit.gu.edu.au

Abstract

Time-triggered systems require a very high degree of temporal accuracy at critical stages during run time. While many software fault injection environments exist today, none of these make provisions to meet the timing requirements of such systems.

This paper introduces a fault injection environment for time-triggered systems. We describe the architecture of FITS and how it addresses the requirements of temporal accuracy in the time-triggered paradigm.

An implementation of FITS was used to conduct fault injection experiments on a prototype implementation of a time-triggered communication protocol for safety-critical hard real-time applications. We describe the fault injection strategy that was used for these experiments to assess the dependability of this protocol.

Through the white box fault injection experiments that were performed using FITS, a number of small design problems were found and subsequently corrected in the protocol. The results of the experiments also enabled an important quantitative assessment of the dependability measures for the protocol, which is vital for its inclusion into dependable systems. Together with the large quantity and variety of experiments this implementation of FITS was able to perform, these are important factors of evidence that demonstrate the viability and utility of the presented fault injection architecture.

Keywords: fault injection, fault tolerance, time-triggered systems, hard real-time systems, validation.

1 Introduction

Time-triggered systems are becoming increasingly important in fault tolerant, hard real-time applications. Unlike event-triggered systems that follow a best-effort model, where the response time varies with load, time-triggered systems guarantee timeliness by adhering to a pre-defined schedule. The temporal behaviour of such systems in worst-case scenarios is equivalent to best or average-case behaviour.

Assessing the dependability of a fault tolerant system is a difficult task, especially in a distributed design, which is often used to avoid a single point of failure. The inherent complexity of a distributed design makes it impracticable or even infeasible to use formal verification as the sole method for validation.

Recently, software-implemented fault injection has become a valuable tool in the evaluation and analysis of system dependability. While less expensive and easier to implement, in many cases, a software-implemented approach has been shown to be as powerful as hardware fault injection methods

(Fuchs 1996, Karlsson 1996). Fault injection can be used as a tool to achieve two major goals during dependability analysis: fault removal and fault forecasting. *Fault removal* aims at reducing the number of faults present in the design or implementation of fault tolerance algorithms and mechanisms (FTAMs), while *fault forecasting* allows the system designer to assess the dependability of the system in the presence of faults (Arlat, Aguera, Amat, Fabre, Laprie, Martins & Powell 1990). Together with other quality assurance techniques, fault removal is often used in the life cycle of a software to achieve fault avoidance, i.e., to avoid internal errors within the software that stem from design, specification, or implementation errors. Fault forecasting, on the other hand, is a method for testing the behaviour of the system and the effectiveness of individual strategies (e.g., fault detection and fault tolerance) under fault conditions.

In a traditional, non-real-time environment, the outcome of a computation is solely defined in the value domain. This means that the correctness of a result only depends on the value of a result, not its timing. In a real-time environment, however, timing plays a crucial role. A result is only correct if it has the correct value and arrives on time. A result that does not arrive on time is useless, regardless of whether its value is correct or not. Whether a real-time system is considered a soft or a hard real-time system depends on the consequences of failure. Failure to meet a timing constraint in a soft real-time system can cause reduced usability, a similar problem in a hard real-time system can have disastrous consequences. A multimedia system is an example for a soft real-time application, while a flight control system in an aircraft is an example for a hard real-time application.

One of the current challenges is to design a fault injection environment that is suited to be used in time-triggered systems. Using software-implemented fault injection in time-triggered systems presents some unique challenges. To date, no generic fault injection environment exists that accounts for the temporal requirements of time-triggered hard real-time systems.

A major problem when injecting faults into a time-driven system is what Gait termed the probe effect (Gait 1985). The probe effect occurs when the behaviour of a system is modified by an attempt to make an observation. As such, it has an impact on both the value and the time domain. Avoiding any adverse probe effects in the value domain can be achieved by using good software engineering practices and modular design. In most cases it is sufficient to show that the system state remains unaffected by the presence of fault injection software when no faults are actually injected. A comparable methodology to do the same in the time domain does not exist. Since all instructions execute in finite time, introducing fault injection code generally changes the temporal behaviour of the

affected software.

The achievement presented in this paper is a generic fault-injection environment that can be easily integrated in time-triggered, distributed, hard real-time systems. The rest of this paper is structured as follows. Section 2 gives a short overview of existing fault injection work. In Section 3 the architecture of FITS, which stands for *Fault Injection for Time-triggered Systems*, is explained. An example implementation of FITS to inject faults into a prototype implementation of a time-triggered communication protocol is described in Section 4. A summary and suggestions for future work concludes this paper in Section 5.

2 Related Work

Fault injection has been used to assess system dependability through the intentional activation of faults by hardware or software means. Since many of these faults are very unlikely to occur during normal operation, this process is often termed *fault acceleration* (Arlat, Crouzet & Laprie 1989). Depending on the abstraction level fault injection tools are located at, these can be classified as physical, simulation-based, or software-implemented fault injection (SWIFI).

SWIFI tools are typically tailored to specific architectural or operating system features. FIAT (Segall, Vrsalovic, Siewiorek, Yaskin, Kownacki, Barton, Dancey, Robinson & Lin 1988) introduces faults into a cluster of nodes connected through a local area network (LAN). One node in the cluster acts as the Fault Injection Manager (FIM), while the other nodes constitute the tested system, the so-called Fault Injection Receptors (FIREs). The FERRARI system (Kanawati, Kanawati & Abraham 1992) consists of a manager module that coordinates various operational modules for initialisation, pre-runtime fault injection, data collection and analysis, and user interaction. Xception (Carreira, Madeira & Silva 1995) uses the PowerPC debugging and performance monitoring mechanisms inject and activate faults. DOCTOR (Han, Rosenberg & Shin 1993) is based on a software fault injector (SFI) that was developed for the experimental distributed real-time system HARTS (Shin 1991). SFIT (Avresky & Tapadiya 1994) is an object oriented Software Fault Injection Tool based on a client/server model. FIMD (Blough & Torii 1997) is a software tool for fault injection in message passing parallel computers. Numerous other approaches utilise similar mechanisms on different architectures and operating systems for the injection of faults (Chillarege & Bowen 1989, Echtle & Leu 1992, Lovrić & Echtle 1993, Kao, Iyer & Tang 1993, Dawson, Jahanian, Mitton & Tung 1996, Stott, Hsueh, Ries & Iyer 1997).

Although modularity has very early been introduced into software-implemented fault injection systems (Kanawati et al. 1992) and such modularity has recently been refined to produce very flexible, lightweight fault injection systems (Stott, Floering, Burke, Kalbarczyk & Iyer 2000), these systems still rely on an infrastructure provided by an underlying operating system. Fault injection environments for embedded systems, such as EXFI (Benso, Prinetto, Rebaudengo & Reorda 1998), are tied to specific hardware. None of the existing systems provide support for time-triggered environments.

3 FITS Architecture

The goal of FITS was to create a generic fault injection environment that is flexible enough to be inte-

grated into time-triggered systems without violating the temporal requirements of such systems. This goal was achieved through a component-based approach that combines a set of system-independent management modules with lightweight, target-specific injection and monitoring modules.

3.1 The Structure of FITS

FITS consists of several modules (Figure 1). The *Campaign Manager* coordinates the fault injection experiments. It is independent of the target system and can therefore be located on a separate (non real-time) computer system. The Campaign Manager is responsible for uploading the setup for a series of experiments onto the target modules and starting the campaign.

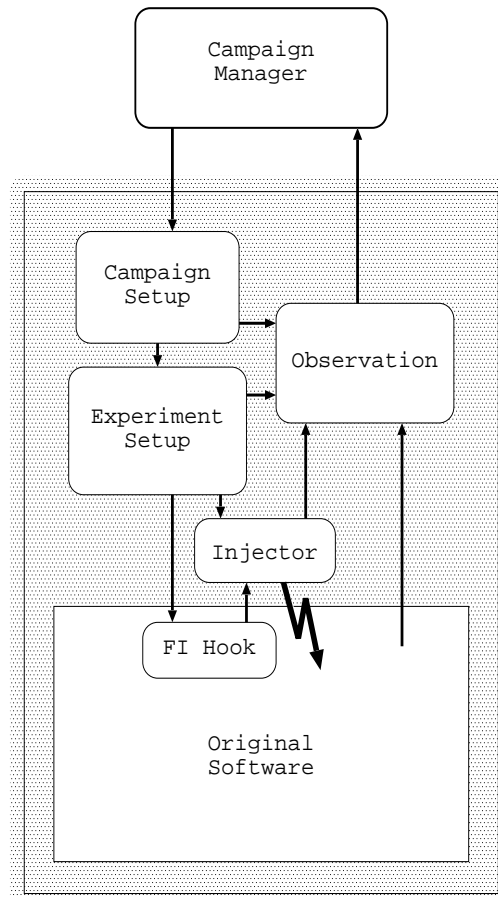


Figure 1: FITS Modules

The *Campaign Setup* module receives campaign data from the Campaign Manager. These include the number of experiments to perform, the location of the fault target area, the timing parameters, the type of faults to be injected, and the duration of each experiment. The Campaign Setup module also prepares a list of fault injection times and locations for each experiment. This list is either received explicitly or derived from a random number seed received from the Campaign Manager.

The *Experiment Setup* module is responsible for managing a single fault injection experiment. It sets up the fault injection time and location for the actual fault *Injector* and, depending on the fault type, selects the Injector to be used.

Once the experiment has been started, the fault Injector module activates the corresponding fault. This can either be done pre-runtime or while the tested system is actually running. Pre-runtime fault

injection does not require any modification of the original real-time system software. Such experiments therefore do not change the temporal behaviour of the device under test. Unfortunately, pre-runtime fault injection is only suitable for a limited number of fault classes such as permanent faults.

For runtime fault injection, interference with the target system cannot be avoided. In order to keep such interference to a minimum, FITS interacts with the tested software using a *Fault Injection Hook* (FI Hook). The FI Hook communicates with the Fault Injector using a standard callback interface. Placement and design of the Fault Injection Hook is crucial and needs to be such that interference with the timing of the target system is kept to a minimum. This is highly system-specific; the place and type of hook depends on the actual software that is being tested.

Since execution time is hardly constant, time-triggered software often contains synchronisation stages at which it waits until a certain point in time has been reached. What is important in a time-triggered real-time system is that the external timing of the system remains unaffected. This means that neither the value nor the arrival time of a result may change because of the presence of fault injection software. If the execution time of the FI Hook is guaranteed to be below the minimum wait time, the external timing of the software will not be changed by the introduction of a FI Hook. A further possibility is the execution time analysis of different program paths. Any path that differs from the worst case execution time by more than the FI Hook execution time is suitable for introducing the hook.

For safety-critical systems, the FI Hook can be designed such that it can be left in place, even when no fault injection experiments are performed. This allows the system to run under the exact same conditions during normal operation as during fault injection. Therefore, there will be no temporal discrepancy, even if the timing was changed when the FI Hook was originally introduced.

During the experiments, the *Observation* module gathers the experiment results and reports them back to the Campaign Manager. The Observation module contains a system-independent part that communicates with the setup modules as well as the fault injector. A workload-dependent part is responsible for gathering the actual system output and experiment data.

3.2 Fault Injection Scenario

In a distributed system, multiple nodes can perform different roles within the system and in relation to fault injection. Figure 2 shows a typical scenario where the Campaign Manager communicates with a cluster consisting of the *Device under Test* (DuT), a *Golden Unit*, an independent *Observer*, and an independent node that does not communicate with the Campaign Manager.

The Device under Test is the node where fault injection actually occurs (FI node). Depending on the scenario, there can be more than one such node. To simulate correlated faults, for example, the tested system could contain multiple such nodes.

In addition to the DuT, an Observer can be used to report back independent experiment data. The Observer is not directly affected by fault injection and might therefore have a different view of the outcome of an experiment than the DuT.

A Golden Unit is a node that is an exact replica of the DuT, except that no faults are injected there. This means that a Golden Unit contains the exact same fault injection software as the DuT (including

a possible FI Hook), with the exception that no actual faults are injected. Golden Units, like FI nodes, can report observation data back to the Campaign Manager. More importantly, FI nodes can be turned into Golden Units during normal operation when no fault injection is performed. This guarantees that the temporal behaviour of a cluster of such nodes will be exactly the same, with or without fault injection.

3.3 Fault Classes

Selection of the fault classes in FITS can be done by choosing or designing an appropriate Injector module. There are two major categories for fault injection experiments. White box fault injection, on the one hand, aims at fault removal. Knowledge about design and implementation specifics of the target system is used to detect and remove existing design and implementation flaws. This requires a significantly different, qualitative fault model that is highly specific to the device under test. Corresponding Injectors and FI Hooks can be implemented and integrated as an addition to the standard fault injectors within the fault injection framework.

Black box fault injection, on the other hand, aims at fault forecasting, i.e., at evaluating the effectiveness of the FTAMs. Such experiments accelerate the number of faults that would occur infrequently during the normal operation of the system by injecting faults at random locations within the system. A large number of experiments is required for a campaign to obtain quantitative, statistically significant measures of error detection coverage and timing parameters.

Regardless of whether fault injection experiments are aimed at fault forecasting or fault removal, there are two classes of faults that can be distinguished by their time of occurrence and their persistence. Permanent faults persist throughout the whole duration of an experiment. Such faults can often be injected pre-runtime. An example for a permanent fault that is typically used is the stuck-at fault model, where a bit gets set to a fixed value (either 0 or 1). For a campaign of experiments, a probability factor can be used to determine the distribution between “stuck-at-0” and “stuck-at-1” faults.

Transient faults are temporary and only occur once within an experiment. A typical example of a transient fault is the bit flip model, where the value of a bit at a random location is flipped during the experiment. Intermittent faults are situated between transient and permanent faults. They persist for a certain period of time after their initial occurrence. FITS allows for any range of faults, from transient to permanent, to be injected by specifying the persistence and time ranges for these faults.

Other criteria for fault classification are location and multiplicity. The fault Injectors can be set up to inject faults only within a limited range of addresses. The actual addresses that are used are target-dependent and can, for example, include specific RAM or ROM areas. The multiplicity parameter specifies the number of faults that should be injected within a single experiment. A multiplicity of one causes the injection of a single, independent fault, while a higher multiplicity factor allows for the simulation of multiple, correlated faults.

4 FITS Experiments

FITS was used to validate the fault tolerance mechanisms of a prototype implementation of TTP/C (Kopetz & Grünsteidl 1994, Hexel 1999). TTP/C is a time-triggered communication protocol for fault tolerant real-time systems. It provides consistent

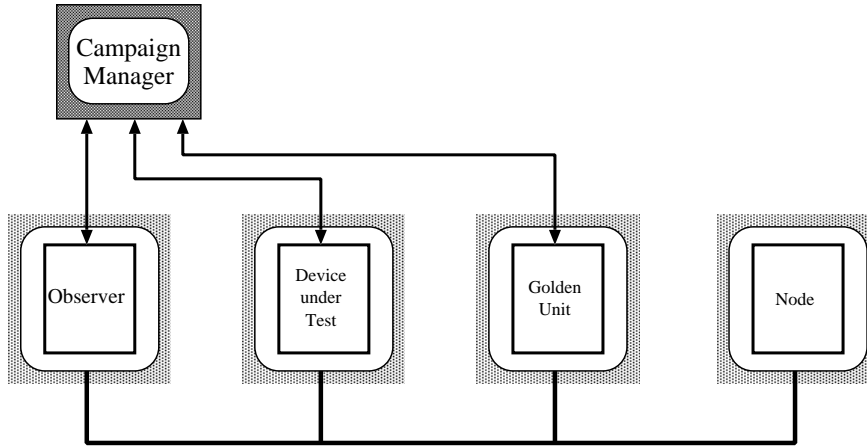


Figure 2: Fault Injection Scenario

message transfer over redundant buses, fault tolerant clock synchronisation, short latency error detection, mode change support, a membership service, and distributed redundancy management.

A TTP/C system consists of at least one cluster of distributed nodes as depicted in Figure 2. The nodes communicate through a set of two broadcast buses for redundancy. A global time base is established within the cluster through clock synchronisation between the nodes (Kopetz, Hexel, Krüger, Millinger & Schedl 1996). Each node is considered fail-silent, i.e., only crash and omission failures can occur.

To assess the validity of the assumptions made in the fault hypothesis for TTP/C, several white box fault injection campaigns were performed. Additionally, black box fault injection was used to evaluate the overall dependability of the protocol.

4.1 State Machine Structure

For a time-triggered system, temporal predictability is of utmost importance. The tested TTP/C prototype software runs on a custom-built embedded system (Figure 3). The system consists of a Motorola MC68332 microcontroller that consists of a MC68k compatible CPU and an independent Time Processing Unit (TPU). The CPU runs the main TTP/C software while the TPU is responsible for maintaining the global time and clock synchronisation. The software and static configuration data are stored in a Flash EPROM while dynamic systems state data are stored in a DPRAM. A part of this DPRAM is externally read/writable and harbours the communication network interface (CNI). A dual-channel Universal Serial Controller (USC) handles all communication over the two redundant broadcast buses. Un-timely write accesses to the bus is prevented by a Bus Guardian (BG), an independent entity that monitors the temporal bus access behaviour of the system. A field programmable gate array (FPGA) acts as glue logic for the components and performs access arbitration and on-the-fly calculation of cyclic redundancy checksums (CRCs) for the communication packets.

The core TTP/C protocol in the tested prototype is executed as a state machine (Pallierer 1996). This state machine is embedded within a time-triggered loop. After system initialisation, this loop is entered and the periodic execution of the two major protocol phases, the Transmission Phase (TP) and the Inter Frame Gap (IFG) commences. These two phases constitute one Time Division Multiple Access (TDMA) slot of the protocol. The beginning of each phase is triggered by a flag set by the TPU once the start time (action time, AT) for the corresponding phase

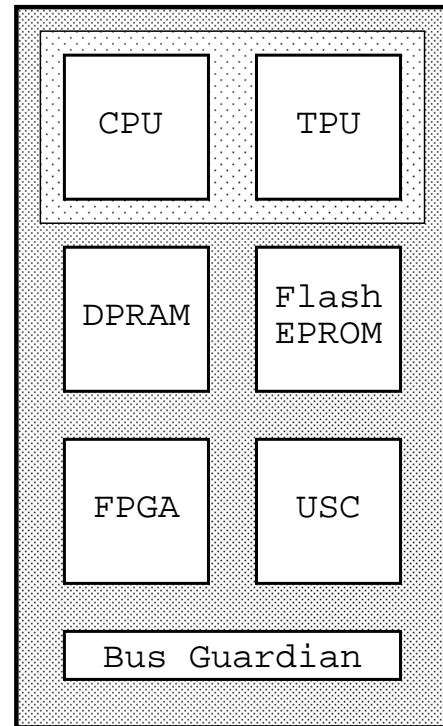


Figure 3: TTP/C Prototype Board

has been reached (Figure 4).

To avoid jitter at action time, the system has been designed to work completely without asynchronous context changes, such as interrupts. This is an essential aspect for SWIFI, because it does not allow the FI Hook to be implemented as (part of) an interrupt handler. Instead, the hook inserts directly into the protocol software. Typically, the core overhead of the fault injection software came from a read, modify, and write operation, regardless of the chosen fault model. This overhead was significantly smaller than the difference between execution times of different branches within the protocol software. More importantly, this overhead was several orders of magnitude smaller than the minimum wait for action time at the end of each loop. This allows the FI Hook to be inserted anywhere within the state machine without changing the external timing of the protocol.

Moreover, FITS was designed to leave FI Hooks in place, resulting in the same amount of code being executed, with or without fault injection. If no fault injection occurs, the fault injection data cause the

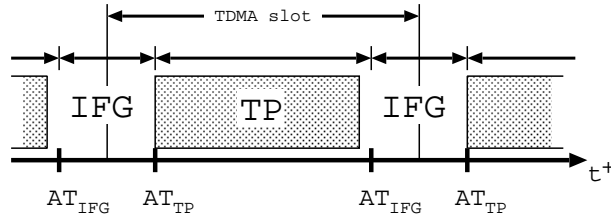


Figure 4: TTP/C Protocol Timing

modify operation to have no effect. As a result, both the internal and external temporal behaviour of the protocol remain identical, regardless of whether fault injection is being performed or not.

4.2 Validation of the Fault Hypothesis

To deliver fault tolerant communication services, several interfaces are defined in the specification of TTP/C that separate the autonomous subsystems. The basic principle behind these interfaces constitutes subsystem independence on the node level: no single fault injected into the device under test should have a negative impact on the correct function of other nodes.

The first targets were the external interfaces used by the protocol. These interfaces comprise the CNI and the broadcast buses that carry out the low-level data transfer within the cluster. Through these interfaces, the TTP/C controller interacts with other components, such as the application host, or the connected nodes in the cluster.

In addition to the external interfaces, TTP/C employs a message descriptor list (MEDL), a static configuration data structure that controls the operation of a node within the cluster. Through this data structure, a logical temporal interface between the nodes is formed. In addition, the MEDL carries information about message properties, i.e., their locations and sizes within the CNI. Thus, the MEDL also constitutes an internal, semantic interface between the controller and the application host, making it another valuable target for the injection of faults.

In total, 14 different, TTP/C specific, white box fault injectors were implemented using FITS. The protocol mechanisms that were tested include protocol startup, clock synchronisation, implicit acknowledgement, clique avoidance, and the host interface. A total of 16849466 experiments were performed in 76 different campaigns. These revealed three protocol design flaws in the TTP/C prototype that were subsequently fixed.

4.3 Dependability Evaluation

In addition to the white box fault injections, the generic stuck-at and bit flip Injectors were used for black box fault injection. Specific FI Hooks were designed for the TTP/C prototype for the introduction of runtime and pre-runtime faults. The injection of a large number of random faults was expected to provide insight into the dependability of the system. An area that was of specific interest in this context was the detection of undesired error conditions that could cause fail-silence violations in the time or value domains.

Dependability evaluation was performed in a series of over 5 million experiments distributed over 6 different fault campaigns (Table 1). A number of errors were triggered by these experiments. Over 8000 fail-silence violations were found in the value domain that require an additional end-to-end checking mechanism

Campaigns Performed	6
Experiments Performed	5366862
Null Frames	105067
Invalid Frames	4392
Blackouts	1241
Value FSVs	8056
Attempted Time FSVs	575

Table 1: Results of the Dependability Evaluation Experiments. The upper part of the table denotes the total number of campaigns and experiments performed. The lower part shows the errors that were triggered for various classes, ordered by increasing severity. *FSVs* denotes *fail silence violations*.

within the application on top of the communication protocol for proper detection. In 575 cases, fail silence violations were attempted in the time domain by the device under test (one of the most severe errors in a time-triggered environment). All these attempts were successfully suppressed by an independent device on the TTP/C node, called a bus guardian, that has been designed to detect and prevent such violations from occurring.

5 Conclusions

In this paper, an architecture for fault injection into time-triggered systems was described. This architecture, FITS, was designed to address most of the shortcomings exhibited by traditional fault injection tools. FITS was the first generic fault injection system that accounts for the temporal requirements of time-triggered hard real-time systems.

To meet these temporal requirements, a component-based architecture was presented that performs the most complex operations in a system-independent pre-runtime stage. Only a small set of target-specific modules was required for the actual introduction of faults during runtime.

The flexibility of FITS was demonstrated through an example implementation using both generic and system-specific fault injectors. This implementation was used to validate the fault tolerance mechanisms of TTP/C, a time-triggered communication protocol for distributed, fault-tolerant real-time systems. A large number of experiments was performed to assess the dependability of the system as well as the validity of its fault hypothesis. A number of design problems in the TTP/C prototype were uncovered in these experiments and subsequently rectified.

For a longer-term contribution, the versatility of FITS can be enhanced by building a library of system-independent fault injection modules for additional, generic types of fault classes. And finally, a fault injection editor that interfaces with FITS is expected to facilitate an easier design of comprehensive fault-injection campaigns for time-triggered systems.

References

- Arlat, J., Aguera, M., Amat, L., Fabre, Y. C. J.-C., Laprie, J.-C., Martins, E. & Powell, D. (1990), 'Fault Injection for Dependability Validation: A Methodology and Some Applications', *IEEE Transactions on Software Engineering* **16**(2), 166–182.
- Arlat, J., Crouzet, Y. & Laprie, J.-C. (1989), Fault Injection for Dependability Validation of Fault-Tolerant Computing Systems, in 'Proc. 19th Int. Symposium on Fault-Tolerant Computing', IEEE Computer Society, Chicago, Illinois, USA, pp. 348–355.
- Avresky, D. & Tapadiya, P. (1994), SFIT-2: A tool for validation of software under hardware faults, Technical Report 94-002, Department of Computer Science, Texas A&M University, USA.
- Benso, A., Prinetto, P., Rebaudengo, M. & Reorda, M. S. (1998), 'EXFI: a low-cost fault injection system for embedded microprocessor-based boards', *ACM Transactions on Design Automation of Electronic Systems*. **3**(4), 626–634.
- Blough, D. M. & Torii, T. (1997), Fault-injection-based testing of fault-tolerant algorithms in message-passing parallel computers, in 'Proc. 27th Int. Symposium on Fault-Tolerant Computing', IEEE Computer Society, Seattle, Washington, USA, pp. 258–267.
- Carreira, J., Madeira, H. & Silva, J. (1995), Xception: Software fault injection and monitoring in processor functional units, in '5th IFIP Working Conference on Dependable Computing for Critical Applications (DCCA-5)', IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance, Urbana Champaign, Illinois, USA, pp. 135–149.
- Chillarege, R. & Bowen, N. S. (1989), Understanding large system failures — a fault injection experiment, in 'Proc. 19th Int. Symposium on Fault-Tolerant Computing', IEEE Computer Society, Chicago, Illinois, USA, pp. 356–363.
- Dawson, S., Jahanian, F., Mitton, T. & Tung, T.-L. (1996), Testing of fault-tolerant and real-time distributed systems via protocol fault injection, in 'Proc. 26th Int. Symposium on Fault-Tolerant Computing', IEEE Computer Society, Sendai, Japan, pp. 404–414.
- Echtle, K. & Leu, M. (1992), The EFA fault injector for fault-tolerant distributed system testing, in 'IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems', Amherst, Massachusetts, USA, pp. 28–35.
- Fuchs, E. (1996), Software Implemented Fault Injection, PhD thesis, Technisch-Naturwissenschaftliche Fakultät, Technische Universität Wien, Vienna, Austria.
- Gait, J. (1985), 'A debugger for concurrent programs', *Software — Practice and Experience* **15**(6), 539–554.
- Han, S., Rosenberg, H. A. & Shin, K. G. (1993), DOCTOR: An integrated Software fault injection environment, Technical Report CSE-TR-192-93, University of Michigan.
- Hexel, R. (1999), Validation of Fault Tolerance Mechanisms in a Time-Triggered Communication Protocol using Fault Injection, PhD thesis, Technisch-Naturwissenschaftliche Fakultät, Technische Universität Wien, Vienna, Austria.
- Kanawati, G. A., Kanawati, N. A. & Abraham, J. A. (1992), FERRARI: A tool for the validation of system dependability properties, in 'Proc. 22nd Int. Symposium on Fault-Tolerant Computing', IEEE Computer Society, Boston, Massachusetts, USA, pp. 336–344.
- Kao, W., Iyer, R. & Tang, D. (1993), 'FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults', *IEEE Transactions on Software Engineering* **SE-19**(11), 1105–1118.
- Karlsson, J. (1996), Transient Fault Effects in the MC6809E 8-bit Microprocessor: A Comparison of Results of Physical and Simulated Fault Injection Experiments, Technical Report 96, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden.
- Kopetz, H. & Grünsteidl, G. (1994), 'TTP — A Protocol for Fault-Tolerant Real-Time Systems', *IEEE Computer* pp. 14–23.
- Kopetz, H., Hexel, R., Krüger, A., Millinger, D. & Schedl, A. (1996), A Synchronization Strategy for a TTP/C Controller, in 'Application of Multiplexing Technology (SP-1137)', Society of Automotive Engineers, SAE Press, Detroit, MI, USA, pp. 19–27. SAE Paper No. 960120.
- Lovrić, T. & Echtle, K. (1993), ProFI: Processor fault injection for dependability validation, in 'International Workshop on Fault and Error Injection for Dependability Validation of Computer Systems', Göteborg, Sweden.
- Pallierer, R. (1996), Prototype Implementation and Evaluation of TTP/C, Master's thesis, Institut für Technische Informatik, Technische Universität Wien, Vienna, Austria.
- Segall, Z., Vrsalovic, D., Siewiorek, D., Yaskin, D., Kownacki, J., Barton, J., Dancy, R., Robinson, A. & Lin, T. (1988), FIAT - Fault Injection based Automated Testing environment, in 'Proc. 18th Int. Symposium on Fault-Tolerant Computing', IEEE Computer Society, Tokyo, Japan, pp. 102–107.
- Shin, K. G. (1991), 'HARTS: a distributed real-time architecture', *IEEE Computer* **24**(5), 25–35.
- Stott, D. T., Floering, B., Burke, D., Kalbarczyk, Z. & Iyer, R. K. (2000), NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors, in 'Proceedings of the 4th IEEE International Computer Performance and Dependability Symposium', Chicago, IL, pp. 91–100.
- Stott, D. T., Hsueh, H.-C., Ries, G. L. & Iyer, R. K. (1997), Dependability analysis of a commercial high-speed network, in 'Proc. 27th Int. Symposium on Fault-Tolerant Computing', IEEE Computer Society, Seattle, Washington, USA, pp. 248–257.