Exploiting Mobile Agents for Structured Distributed Software-Implemented Fault Injection

Thomas M. Galla
DECOMSYS
Dependable Computer Systems GmbH
Vienna, Austria
Email: thomas.m.galla@decomsys.com

Karin A. Hummel
Institute of Distributed and Multimedia
Systems, University of Vienna
Vienna, Austria
Email: karin.hummel@univie.ac.at

Burkhard Peer
DECOMSYS
Dependable Computer Systems GmbH
Vienna, Austria
Email: burkhard.peer@decomsys.com

Abstract

Embedded distributed real-time systems are traditionally used in safety-critical application areas such as avionics, healthcare, and the automotive sector. Assuring dependability under faulty conditions by means of fault tolerance mechanisms is a major concern in safety-critical systems. From a validation perspective, Software-Implemented Fault Injection (SWIFI) is an approved means for testing fault tolerance mechanisms.

In recent work, we have introduced the concept of using mobile agents for distributed SWIFI in time-driven real-time systems. This paper presents a prototypical implementation of the agent platform for the OSEKtime real-time operating system and the FlexRay communication system. It is further shown, how to implement fault injection experiments by means of mobile agents in a structured manner following a classification of faults in terms of domain, persistence, and perception. Based on experiments conducted on ARM-based platforms, selected results are described in detail to demonstrate the potential of mobile agent based fault injection.

1. Introduction and Related Work

Software-Implemented Fault Injection (SWIFI) is a well established means for evaluating fault tolerance mechanisms due to flexibility and cost-efficiency reasons [1], [2]. Fuchs showed that using SWIFI it is possible to achieve similar failure scenarios as by means of hardware implemented fault injection [1] since hardware failures caused by environmental phenomena affect the system software and, thus, can be detected and simulated at the software level. In addition to injecting specific heuristic faults, SWIFI is further used to inject faults based on random processes to provide a required test coverage [3].

In *distributed* fault-tolerant systems, which are usually based on redundancy concepts, the consequences of node failures as perceived by other nodes are of major interest. In such systems faults are perceived as different kinds of messaging failures. From the perspective of distributed computing, the first challenge for a successful SWIFI approach is targeted to manipulating communication, that is, interfering with the system's middleware layer. The second challenge arises from the need for high-quality SWIFI frameworks and approaches. In detail, any SWIFI approach has to assure the following major requirements: (i) after the testing phase, the fault injection software has to be removed safely from the system in order to prevent accidental invocation in the production system, (ii) probe effects have to be avoided, and (iii) traditional non-functional software quality criteria have to be met, such as maintainability, adaptability, and reliability.

To allow for controlling of distributed inter-process communication, SWIFI is thus implemented either as an additional middleware layer, by accessing main middleware functions, or by dedicated middleware technologies. Dawson [4] proposes a SWIFI middleware layer for fault injection into TCP/IP based protocols. Similarly, Koopman et al. [5] introduce such a layer for the automotive *Controller Area Network (CAN)* [6], [7] communication protocol.

In contrast to special purpose fault injection implementations for evaluating specific fault-tolerant real-time systems, a few related works describe new methods for implementing SWIFI tools from a software engineering perspective. These works address the second challenge of providing high-quality SWIFI necessary for achieving useful test results. Martins et al. [8] describe an approach introducing reflection and object oriented programming to SWIFI. This approach allows to separate the functional aspects of SWIFI, that are, the specification of the particular faults to be injected, from the non-functional aspects related to code instrumentation necessary to inject the faults. For high-level fault injection, *GOOFI* [9] presents a generic object oriented fault injection framework based on Java and Structured Query Language (SQL).

In [10] we have introduced the concept of using *mobile agents* as a means for distributed SWIFI in time-driven real-time systems. Mobile agents [11], [12] are programs which decide when and where to migrate to in an autonomous manner. The agents are supported by agent platforms executing the agents. By migrating, mobile agents may invoke procedures locally and, thus, reduce network communication overhead in distributed systems [13]. We exploit mobile agents in order to:

- assure *easy deployment and removal* of the fault injection code, which is realized by the migration concept of mobile agents,
- assure *adaptability* by encapsulating the fault injection logic in the agent, and

- realize *distributed fault injection* by either deploying multiple agents or by using agent migrating from one node to another.

Since our approach is targeted to time-driven real-time systems we assume the existence of a global time and the possibility to access the application's memory area. The latter condition is usually met by current real-time system architectures and can be exploited to interfere with the system's middleware layer.

This paper enhances our former works by demonstrating the feasibility of our approach. We present a prototypical implementation of the concept including the agent implementation language based on the time-driven operating system OSEKtime [14] and the time-driven communication protocol FlexRay [15] which is used for safety-critical automotive applications. We further show, how mobile agents can be implemented for structured fault injection based on a commonly used classification of failures and faults in terms of domain, persistence, and perception. Finally, we use an example application to demonstrate the results achievable by fault injection experiments.

The remainder of this paper is structured as follows: Section 2 summarizes the concept of SWIFI based on mobile agents, while Section 3 introduces structured SWIFI based on a commonly used fault classification. Section 4 details the implementation and Section 5 describes the experiments conducted and results achieved by applying the agent prototype to an example application. Section6 concludes this work.

2. Mobile Agent Based SWIFI

In this section, we describe the concept of using mobile agents for SWIFI in brief by introducing the mobile agent paradigm and the proposed system architecture.

2.1. On Mobile Agents

Mobile agents [11], [12], [16], [17] have been introduced as a new communication paradigm during the last decade. Instead of sending messages from one process to another or transferring passive information, autonomous migration of code is proposed. In order to support mobile agents, an agent platform on each node has to be provided to en/decapsulate and to send/receive the agents in a reliable way. Hence, large databases may be searched after migration to the database and inter-process communication may take place locally. From a performance perspective, migration of code is preferred when the migration costs are lower than the message transfer costs in terms of network load.

In case mobile agents migrate statefully, that is, the code and execution status is transferred, the term *strong migration* is used, while *weak migration* is used for stateless migration. Note, that strong migration requires more sophisticated techniques. The sequence of locations (or migrations) can be used for ordering of activities, in particular when strong migration is performed. In our mobile agent based SWIFI approach, we address both migration types.

2.2. Hardware and Software Architecture

The overall architecture is depicted in Figure 1. In addition to the fault injection agents *(FI agents)*, our fault injection approach consists of one dedicated central fault injection development and evaluation unit, a gateway node, and several computing nodes.

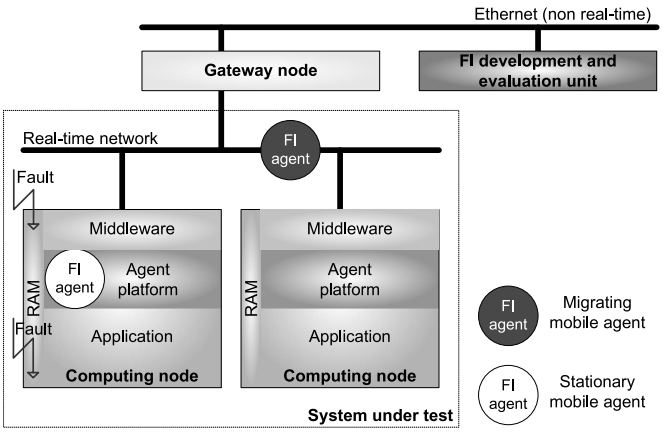


Figure 1. Fault injection architecture

The *computing nodes* (i.e., the nodes under test) execute the distributed real-time application and are connected via a real-time communication network which is used for message exchange according to a communication protocol with real-time capabilities supported by the middleware. The execution of this communication protocol is done by a dedicated communication controller, whereas the main CPU of the computing nodes executes the application tasks under the control of a real-time operating system. These application tasks exchange information by means of messages, which are either delivered locally or sent over the communication network in case the tasks are located on different computing nodes.

The *fault injection (FI) development and evaluation unit* is used for the development of the fault injection agents, for the triggering of the initial migration of the fault injection agents and for the collection, evaluation, and presentation of the results of the fault injection actions. Furthermore, it may send control messages, like a termination request which forces the agents to terminate immediately, to the mobile agents by means of message broadcasting. This fault injection development and evaluation unit is connected to a non real-time network.

The *gateway node* acts as an entry point for the mobile agents and relays control messages from the fault injection development and evaluation unit to the real-time network of the computing nodes. For this purpose the gateway is connected to both the non real-time network and the real-time network.

2.3. Principle of Mobile Agent Based Fault Injection

On each computing node, an instance of the agent platform which interprets and executes the fault injection agent is installed prior to the start of the fault injection experiment.

After successful installation of the agent platform, the fault injection agents are allowed to migrate to these nodes and to perform their fault injection actions in an autonomous fashion.

Multiple similar or different, dependent or independent faults are injected by exploiting the mobile agents' ability to roam between the nodes and to decide autonomously which faults to inject. Hereby, the fault injection can either be triggered by time or events, like (control) message reception. In order to inject the required faults in parallel, one mobile agent is cloned and roams to each node under test.

Additionally, it is possible to model fault chains, i.e. a sequence of faults on one node, and – in principle – also on different nodes. In contrast to agent systems using messages for coordination issues, in our approach the mobile agents do not exchange messages with one another. The reasons for this restriction are the limited bandwidth available on the real-time network and the determinism of the fault injection experiment, which allows to decide upon coordination issues before runtime by means of a common notion of time provided by the real-time network. However, for the communication with the fault injection development and evaluation unit, bandwidth is statically allocated for the exchange of control messages. Thus, this messaging is non-intrusive to the communication of the real-time application under test.

To avoid probe effects caused by the fault injection software, mobile agents are scheduled in a way to prevent temporal interference with the application tasks. Since the targeted real-time system uses static scheduling, it is possible to predict and use just the free time slots for the execution of the fault injection software. As a consequence, the assumed time granularity for the fault injection experiment is adjusted depending on the calculated CPU load before runtime. In a similar manner the bandwidth required for the migration of the agents is statically allocated and thus exclusively reserved for this purpose.

The effects of the injected faults are perceived and logged by the agent on the local computing node in a non-volatile memory. After the completion of the fault injection experiment the logged data is retrieved from the non-volatile memory and transferred back to the fault injection development and evaluation unit.

3. Structured Fault Injection Framework

This section introduces the faults considered for structured fault injection, describes how the different faults are injected based on the mobile agent approach, and summarizes the development process necessary for implementing the fault injection code.

3.1. On Faults and Failures

Regarding to failures, we use the definition provided in [18], where Avižienis et al. define a *failure* as

> *an event that occurs when the delivered service [of a system] deviates from the correct service.*

Since a service is a sequence of the system's external states, a (service) failure means that at least one external state of the system deviates from the correct system state. Such a deviation in an external state is called an *error*. The hypothesized cause of the error is termed a *fault*.

Avižienis et al. provide a classification of failures according to the following dimensions:

- **Domain:** Here, content (value domain) vs. timing (time domain) failures are distinguished. For content failures, the service data deviates from the service specification. In case of timing failures, a service is delivered at a point in time deviating from the service specification.
- **Detectability:** Depending on whether service failures are reported to the user a distinction between signaled and unsignaled failures is made.
- **Consistency or Perception:** In case a service failure is perceived in a similar manner by all system components, the term consistent (in this paper called symmetric) failure is used. Otherwise, the term inconsistent (in this paper called asymmetric or byzantine) failure is used.
- **Consequences:** Due to the environmental consequences, minor failures are distinguished from catastrophic failures.

As far as failures are concerned, the distinction between signaled and unsignaled failures and the differentiation between minor and catastrophic failures does not have any impact on the fault injection setup. Avižienis et al. further provide a detailed classification of faults. From a fault injection setup's perspective however, the occurrence of the fault is the only attribute that has to be considered. Here, transient, intermittent, and permanent faults are distinguished. Transient faults are defined as faults occurring only once, intermittent faults may happen repeatedly, and permanent faults remain in the system.

As a consequence, in this paper we focus on an approach capable of injecting transient, intermittent, and permanent faults that cause consistent and inconsistent failures in both the time and the value domain by means of SWIFI. Formally, we define each fault as a 3-tuple c as follows:

$$c =< c_1, c_2, c_3 >, \quad \text{where} \tag{1}$$

$$c_1 \in \{value, time\} \tag{2}$$

$$c_2 \in \{transient, intermittent, permanent\} \tag{3}$$

$$c_3 \in \{symmetric, asymmetric\} \tag{4}$$

Hereby, c_1 denotes the *domain* of the corresponding failure, c_2 denotes the *persistence* of the fault, and c_3 denotes the *perception* of the failure. Figure 2 illustrates and summarizes the resulting 12 fault classes in a geometrical manner.

3.2. Injecting Faults

In order to inject these faults the fault injection software manipulates received and/or transmitted messages and application data. Hereby, access to the application's memory is required. This is usually possible in embedded systems, for example, in the automotive industry, since the used devices do not provide memory protection mostly for cost reasons. Additionally, means to lookup the right memory address for the desired message or for the application data are needed.

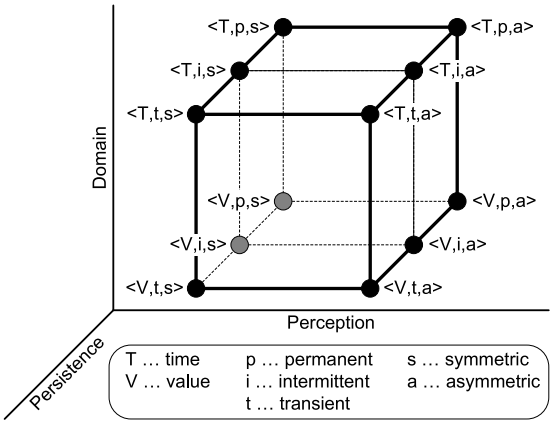


Figure 2. Fault classification

According to the domain the fault injection software determines what type of fault to inject as described by Table 1. While for the value domain the message content is altered, in the time domain the reception or transmission of a new value is postponed which is modeled by using the last (old) value.

Table 1. Injecting faults in the time and value domain

c_1	*value*	*time*
FI function	alter message content	use old value

Due to persistence (Table 2), these fault injection actions are defined in terms of frequency and occurrence of the fault injection function. For the intermittent faults, the periods are configurable and may change during runtime.

Table 2. Configuring fault injection due to persistence

c_2	*transient*	*intermittent*	*permanent*
FI configuration	once	periodically	permanently

Finally, the perception of the failures caused by the injected faults can be configured as described in Table 3. Symmetric failures can be easily implemented by manipulating messaging at the sender because all nodes will perceive the alteration in a similar manner. Otherwise, manipulating messaging at a subset of the receiving nodes allows for injecting faults causing asymmetric failures. Hence, the 12 different fault classes listed in Figure 2 can be implemented by looking up the appropriate dimension value in each table.

Table 3. Configuring fault injection due to perception

c_3	*symmetric*	*asymmetric*
FI configuration	alter send buffer	alter receive buffer

3.3. Development Process

In the field of embedded systems many actions performed by the system are planned and fixed at system design time. Figure 3 depicts the application development process extended by the fault injection implementation steps.

Commonly the design information (e.g., the application tasks executed on each node as well as the messages transmitted and received by each computing node) is stored in a so-called *design database*. Based on the information stored in this database, *middleware code* (the middleware layer) and *operating system (OS) configurations*, like task dispatching tables are generated in an automatic fashion. The middleware layer hides communication details from the application by means of a standardized application programming interface and allocates memory regions for the local storage of the message data (so-called "message buffers").

These automatically generated operating system configuration and middleware layer files are compiled and linked with the *application source code*. This produces an *executable* file for the computing node and the *map-file*, which contains a mapping of symbolic information (i.e., variable names and function names) to the corresponding memory locations on the computing nodes.

The fault injection agents are implemented in a dedicated *FI agent implementation language*, which is translated into a byte-code via the *FI agent compiler*. This compile step is performed on the fault injection development and evaluation unit prior to the actual run-time. The symbolic information (i.e., message and variable names) is transformed into physical memory addresses using the contents of the design database and the map-file.

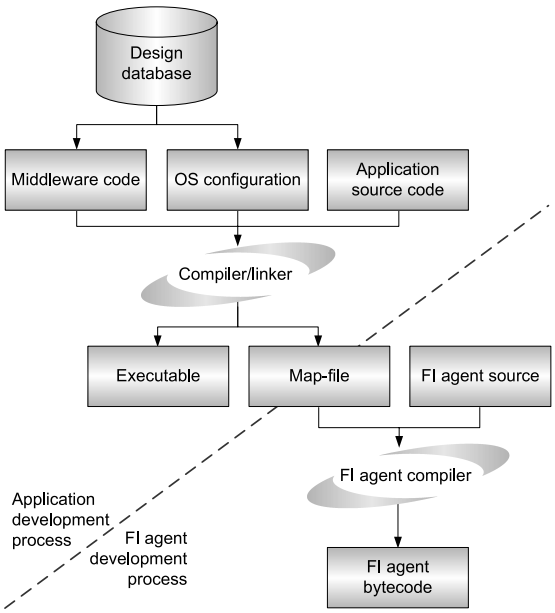
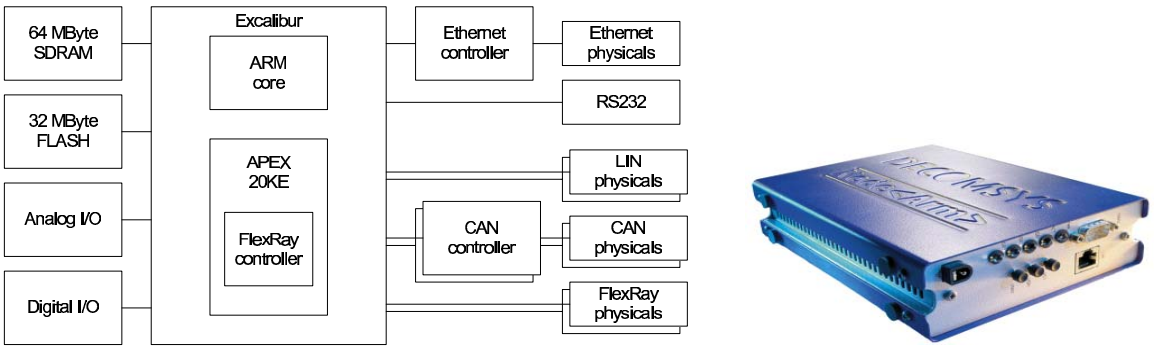


Figure 3. Development process

4. Prototypical Implementation

The prototypical implementation is targeted at time-driven real-time systems which do not provide memory protection. As a consequence, we can make use of the global time provided



(a) (b)

Figure 4. NodeARM prototype platform (a) schematic overview, (b) snapshot

by the real-time network and inject faults by manipulating the RAM (including the communication buffers) of a node. The approach supports both strong, that is, stateful, and weak, that is stateless, agent migration.

In this section, we describe the prototype hardware used for the implementation as well as the core functions of the implementation, that are, the FI agent byte code generator based on the FI agent implementation language specification, and the FI agent platform including the interpreter and the agent structure. The FI agent implementation language defines a small set of operations for the fault injection experiments in a batch-oriented manner. Caused by this small set of operations, we decided not to use any existing byte code generator, like Java language compilers, but to implement a lightweight and specific language.

4.1. Prototype Hardware

The implementation was done on an embedded automotive prototyping platform based on an Altera "Excalibur" device, which integrates an industry-standard 166 MHz ARM922T processor with an APEX 20KE FPGA device-like architecture. Figure 4(a) provides a simplified schematic overview of the prototyping platform, whereas Figure 4(b) depicts a snapshot of the prototyping platform.

The platform provides 64 MByte of SDRAM, 32 MByte of FLASH, and multiple on-board interfaces (2 CAN interfaces, 2 LIN interfaces, 1 Ethernet interface, 24 digital I/O lines, 4 analog I/O lines). Within the APEX 20KE FPGA device the design for a FlexRay communication controller is located. The prototyping platform makes use of the OSEKtime real-time operating system [14] and the FlexRay communication system [15], [19] as real-time network.

4.2. FI Agent Implementation Language

The structure of the FI agent implementation language follows an event-action scheme, where actions are triggered by pre-defined events. In addition, the mobile agents use node location information to control the context of the fault injection experiments.

Triggered by time or on appearance of control messages the mobile agent decides on the action to perform. The temporal trigger conditions are based on the global time (in units of microseconds) provided by the real-time network. The action is either a migration to another node, the manipulation of messages or the mutilation of memory areas. Furthermore, the agent may log results of the fault injection experiment and terminate.

Based on a simple grammar (for details see [10]), fault injection scenarios can be described. By means of the prototypical implementation we validated the major parts of the grammar. The instruction set of the byte code supported by the FI agent platform is defined in an interface declaration file. Hence, the capabilities and the memory footprint of the FI agent platform can be scaled by means of this interface declaration file. In principle, the agent byte code supports the following operation types:

- basic operations (e.g., `killme()` used to terminate the agent),
- stack and arithmetic operations (e.g., `push()`),
- control flow operations (e.g., `return()`),
- register operations, which are used to inject the faults in the value and time domain (e.g., `rset()` used to consume a value from the stack and write it to a register), and
- agent migration operations (e.g., `sendstfl()` used for stateful migration of the agent).

In Table 4, three examples for byte code operations are listed, that are, the `push()` operation which pushes a value onto the stack, the `memset<x>()` operation (for `<x>` $\in 8, 16, 32$) which writes to a specific memory address, and the `sndstfl()` operation which invokes stateful agent migration. Figure 5 illustrates the mapping of a simple agents's source code to the corresponding byte code operations.

Table 4. Agent bytecode operation examples

Instruction	Operand_1		Operand_2		...
	Name	Type	Name	Type	
`push()`	`value`	`int32`	-	-	...
`memset<x>()`	`regnr`	`int8`	-	-	...
`sndstfl()`	`hostid`	`uint8`	-	-	...

```
\\agent source                                    \\byte code operation

ON NODE 0x1 AT_TIME 100                           \\controlled by agent platform
   MIGRATE_TO NODE 0x0A                           sndstfl 0x0A
ON NODE 0x1 AT_TIME 200
   CHANGE_MESSAGE_CONTENT wheel_angle TO 0xAC     push 0x00944101 \\push memory address to stack
                                                  pop \\move memory address to register 0
                                                  mget16 0 \\read memory address to stack
                                                  push 0xAC \\push value 0xAC to stack
                                                  iadd \\write sum of last two stack values to stack
                                                  mset16 0 \\write sum to memory address stored in register 0
ON NODE 0x1 AT_TIME 300
   MIGRATE_TO NODE 0x0B                           sndstfl 0x0B
```

Figure 5. Mapping of an example agent source to the byte code operations

4.3. FI Agent Structure and Platform

Mobile agents migrate from one FI agent platform to another. On each FI agent platform, an interpreter task executes, which is a stack-based program written in the ANSI-C programming language. The interpreter executes the agent's code and tracks the operations executed. The invocation time of an agent is determined by the task schedule of the agent platform and by the agent code itself by means of the agent's ability to read the current global time of the real-time network.

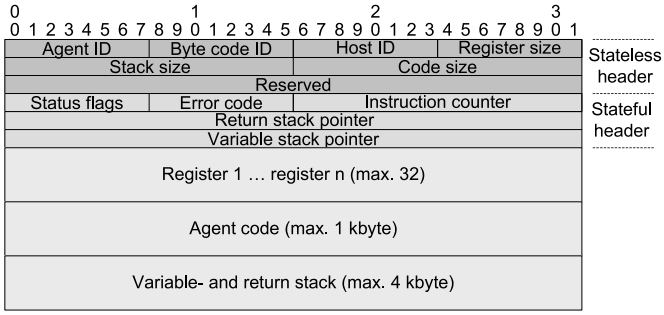


Figure 6. Agent structure

Figure 6 depicts the structure of an agent and the information used by the agent platform to execute the agent. In case the agent code proposes migration to another node, a *move status bit* is set in the agent's *status flags* and the program execution is stopped. Then, the agent platform prepares the agent for migration by performing the following steps:

1) The agent ID, version, source and destination address, register size, stack size, and code size are inserted into the *stateless header*. In case stateless migration is used, goto step 4).
2) In case of stateful migration, the instruction pointer, return stack pointer, variable stack pointer, and status or error bytes are inserted into the *stateful header*.
3) The return stack and the variable stack are compressed and copied to the corresponding areas of the agent body.
4) The agent byte code is compressed and copied to the agent body.

After these preparation actions, the agent is transmitted by means of the transport layer. For the migration of the agents and for the transfer of the logged data a variation of the ISO/DIS 15765-2.2 transport protocol [20] is used. This protocol was originally developed for CAN networks for the exchange of data packets which are larger than the maximum transfer unit (MTU) of the underlying network.

5. Experimental Proof of Concept

For demonstration purpose we conduct distributed fault injection experiments using simple *sine* and *ramp* test functions. Hereby, four NodeARM nodes are used for the experiments as depicted in Figure 7. The *sine node* calculates the original sine function. The *ramp node* generates a periodic ramp with different gradients (note, that a gradient of 0 denotes that the value should be constant). The *combine node* adds the two values received and generates an altered sine curve which is then sent to the *extract node*. By subtracting the ramp value from the value received from the combine node, the extract node is able to restore the original sine function.

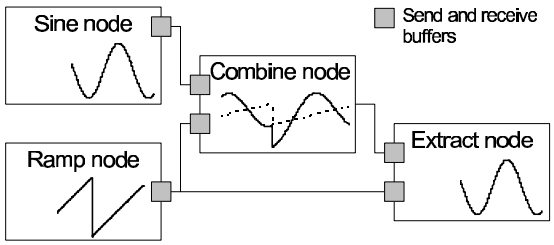


Figure 7. Experimental setup: sine-ramp

For the fault injection experiments, mobile agents migrate to a specific node and inject faults by altering the *send and receive buffers* of the respective nodes. Send buffers are altered to inject symmetric faults, whereas receive buffers of different nodes are altered in different ways to inject asymmetric faults. The output of each node is tracked by means of log files which are evaluated in a post fault injection manner. The results of the fault injection experiments are visualized as deviations from the original sine function.

The fault injection experiments are structured according to the fault classification presented in Section 3 (Figure 2). We will describe three major representatives selected from these classes which show the most interesting results. Additionally, we will present a case where multiple faults are injected. All

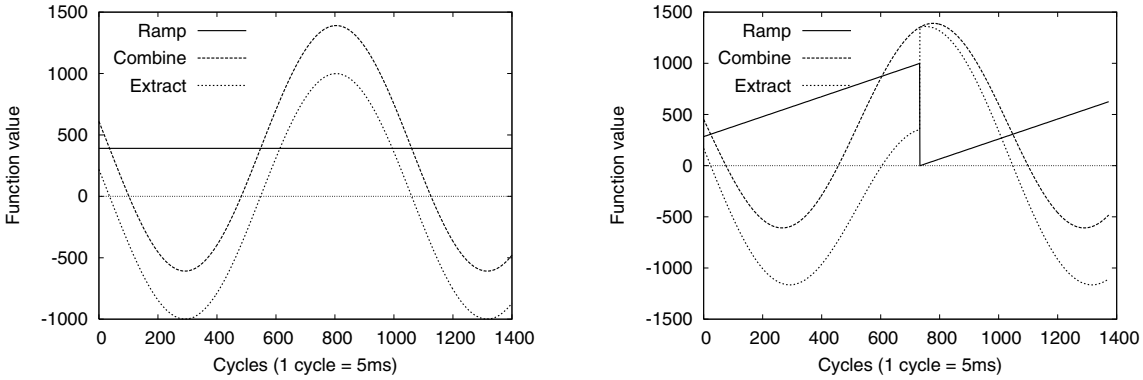


Figure 8. Experiment (a) $< value, permanent, symmetric >$, (b) $< value, permanent, asymmetric >$

results represent the system state as perceived by the extract node over time measured in cycles (1 cycle = 5ms). The curves show the values received from the ramp and the combine node, as well as the results calculated by the extract node respectively.

5.1. Fault Injection in the Value Domain

Figure 8(a) depicts a permanent fault ($c =< value, permanent, symmetric >$) injected by a mobile agent which has migrated to the ramp node. Here, the agent resets the ramp function value in the node's send buffer to the constant value $v = 391$ (assuming gradient 0 of the ramp). Thus, the combine node generates a function which simply adds the constant value of 391 to the original sine value. Since the change is consistently perceived by the extract node, this node subtracts the received (faulty) offset from the combine node's value. Consequently, the curve for the extract host depicts the original sine function. In the system under operation, such a situation may occur in case of a crash failure (or a propagation of a crash failure) and the application logic is responsible for tolerating this fault (similarly to the extract node in the example).

A similar case is depicted in Figure 8(b) for permanent asymmetric fault injection in the value domain ($c =< value, permanent, asymmetric >$). In contrast to the symmetric case, the mobile agent agent migrates to the combine host and changes the ramp node messages to the constant value $v = 391$ by altering the receive buffer. The extract node perceives the original ramp function values as depicted by the ramp curve. The extract node is not able to reconstruct the original sine value because it is not aware of the ramp fault at the combine host. The remaining offset at cycle 711 is given as: $offset = sine() + v - ramp()$. During the next cycle, the ramp function is set to 0 and the extract node calculates the same value as the combine node. Such failures are typical byzantine failures, which might occur, for example, in case of incoming link failures of a single node.

For transient and intermittent failures, the results observed by the mobile agent based experiments are similar. However, differences in the consequences due to the period or the point of occurrence in time can be observed.

5.2. Fault Injection in the Time Domain

In addition to value domain fault injection, we have further conducted experiments targeting timing failures. Here, the sending of a new value is delayed by resending another (old) value instead. For $c =< time, permanent, asymmetric >$, Figure 9(a) depicts a scenario where a mobile agent migrated to the extract node and injected a permanent temporal delay of 2 cycles. The effects of this fault are marginal, except at cycle 747. Here, this injected fault causes a significant spike at the sine value reconstructed by the extract node. Timing failures might occur due to increased process execution times caused by, for example, cache misses, pipeline stalls, or complex exception handling.

5.3. Virtual Fault Injection Chains

Distributed fault injection is enabled by the ability of the agents to migrate to arbitrary nodes. This distributed fault injection can either be realized by migrating several agents before the experiment is conducted or by including agent migration as a means to model fault causalities as fault chains.

Figure 9(b) depicts a virtual fault injection chain, that is, a sequence of faults which model causality by means of event ordering. Due to the availablility of a global time base provided by the real-time network, logical ordering of fault injections can easily be achieved by invoking these experiments at different, subsequent points in time. Here, two transient value failures are injected in an asymmetric manner. One mobile agent migrated to the extract node and injected a fault (a temporary different constant value) into the receive buffer for the ramp node at cycle 551. Since the combine node does not perceive this fault, this fault does not influence the combine value.

Additionally, another mobile agent migrated to the combine node and injected another fault (again a temporary different constant value) into the receive buffer for messages from the ramp node at cycle 627. The curve representing the value calculated by the combine node shows the effect. Since the

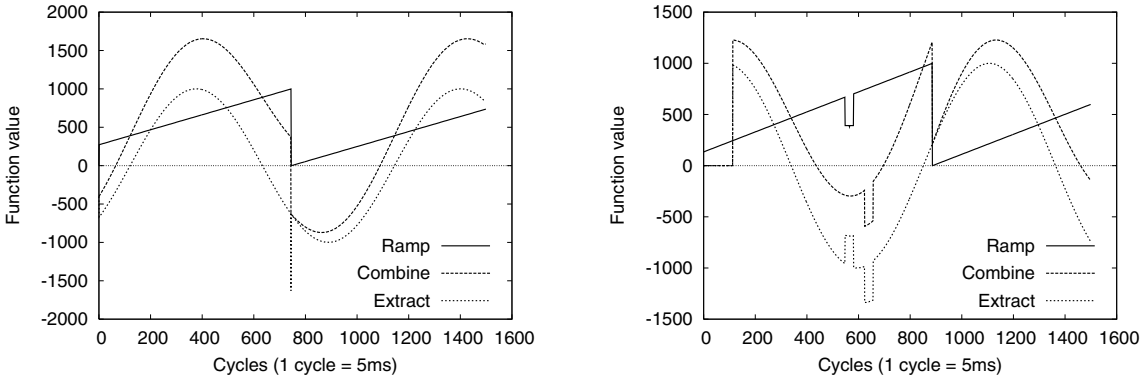


Figure 9. Experiment (a) $< time, permanent, asymmetric >$ and (b) virtual fault chain $< value, transient, asymmetric >$

extract node perceives a different ramp value, this node cannot reconstruct the original sine signal. Thus, this asymmetric virtual fault chain is perceived as two failures at the extract host when compared to the original sine function.

6. Conclusion

In this paper we showed experimentally that mobile agents are a feasible means for deterministic and reliable software-implemented fault injection. We presented the implementation details of the lightweight fault injection agent platform based on the time-driven real-time operating system OSEKtime [14] and real-time communication protocol FlexRay [15]. Both technologies are in particular suitable for automotive systems. The mobile agents make use of system resources utilizing the mapping information generated at compile time, of the global time provided by the FlexRay communication protocol, and of the communication primitives provided by the FlexRay transport layer.

We presented a framework for structured fault injection based on a commonly used classification of faults and failures and showed the feasibility of using mobile agents for the respective fault injection experiments. We conducted fault injection experiments for an example sine and ramp application for both value and time domain failures with different degrees of persistence. Additionally, faults causing both symmetric and asymmetric failures have been injected and virtual fault chains have been experimentally evaluated.

Hence, we have demonstrated, that mobile agents are well suited for structured fault injection. In addition to the benefits achieved by encapsulating the fault injection logic into a mobile agent in terms of adaptability and ease of deployment and removal, we further see the strength of the approach for modeling the distributed nature of faults and fault chains.

7. References

[1] E. Fuchs, "Software Implemented Fault Injection," Ph.D. dissertation, Institute for Computer Engineering, Vienna University of Technology, Treitlstrasse 3/182-1, A-1040 Vienna, Austria, Apr. 1996.

[2] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber, "Comparison of Physical and Software-Implemented Fault Injection Techniques," *IEEE Transactions on Computers*, vol. 52, no. 9, pp. 1115–1133, 2003.

[3] R. R. Some, W. S. Kim, G. Khanoyan, L. Callum, A. Agrawal, and J. J. Beahan, "Software-Implemented Fault Injection Methodology for Design and Validation of System Fault Tolerance," in *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE Computer Society, 2001, p. 0501.

[4] S. Dawson, F. Jahanian, T. Mitton, and T.-L. Tung, "Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection," in *Proceedings of the 26th Annual Symposium on Fault-Tolerant Computing*, 1996, pp. 404–414.

[5] P. Koopman, E. Tran, and G. Hendrey, "Toward Middleware Fault Injection for Automotive Networks," in *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing*, 1998. [Online]. Available: http://www.chillarege.com/fastabstracts/ftcs98/359.html

[6] ISO, "Road Vehicles – Controller Area Network (CAN) – Part 1: Data Link Layer and Physical Signalling," ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, Tech. Rep. ISO/DIS 11898-1, 2003.

[7] ISO, "Road Vehicles – Controller Area Network (CAN) – Part 2: High-Speed Medium Access Unit," ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, Tech. Rep. ISO/DIS 11898-2, 2003.

[8] W. Martins and A. C. A. Rosa, "A Fault Injection Approach Based on Reflective Programming," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2000, p. 407.

[9] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "GOOFI: Generic Object-Oriented Fault Injection Tool," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2001, p. 0083.

[10] T. M. Galla, K. A. Hummel, and R. Pallierer, "Software-Implemented Fault Injection for Safety-Critical Distributed Systems by Means of Mobile Agents," in *Proceedings of the 37th Annual Hawaii International Conference on System Sciences*. IEEE Computer Society, 2004, p. 90302a.

[11] D. Chess, C. Harrison, and A. Kershenbaum, "Mobile Agents: Are They a Good Idea?" IBM, Tech. Rep. IBM Research Report Rc 19887, Dec. 1994.

[12] J. E. White, "Mobile Agents," in *Software Agents*, J. M. Bradshaw, Ed. MIT Press, 1997, pp. 437–472.

[13] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems - Concepts and Design*, 3rd ed. Pearson Education, 2001.

[14] OSEK, "OSEK/VDX – Time-Triggered Operating System," OSEK, Tech. Rep., July 2001. [Online]. Available: http://www.osek-vdx.org/mirror/ttos10.pdf

[15] R. Mores, G. Hay, R. Belschner, J. Berwanger, C. Ebner, S. Fluhrer, E. Fuchs, B. Hedenetz, W. Kuffner, A. Krüger, P. Lohrmann, D. Millinger, M. Peller, J. Ruh, A. Schedl, and M. Sprachmann, "FlexRay – The Communication System for Advanced Automotive Control Systems," in *Proceedings of the SAE 2001 World Congress*. Detroit, MI, USA: Society of Automotive Engineers, Mar. 2001.

[16] A. Fuggetta, G. P. Picco, and G. Vigna., "Understanding Code Mobility," *IEEE Transactions on Software Engineering*, vol. 24, no. 5, pp. 342–361, May 1998.

[17] D. S. Milojicic, W. LaForge, and D. Chauhan, "Mobile Objects and Agents (MOA)," in *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*. Santa Fe, NM, USA: USENIX, Apr. 1998.

[18] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[19] T. Führer, F. Hartwich, R. Hugel, and H. Weiler, "FlexRay – The Communication System for Future Control Systems in Vehicles," in *Proceedings of the SAE 2003 World Congress & Exhibition*. Detroit, MI, USA: Society of Automotive Engineers, Mar. 2003.

[20] ISO, "Road Vehicles – Diagnostics on Controller Area Networks (CAN) – Part 2: Network Layer Services," ISO (International Organization for Standardization), 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, Tech. Rep. ISO/DIS 15765-2.2, April 2003.