Software Implemented Fault Injection for Safety-Critical Distributed Systems by Means of Mobile Agents

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

Karin A. Hummel
Department of Computer Science
and Business Informatics
University of Vienna, Austria
Email: karin.hummel@univie.ac.at

Roman Pallierer
DECOMSYS
Dependable Computer Systems GmbH
Vienna, Austria
Email: roman.pallierer@decomsys.com

***Abstract*— The availability of inexpensive powerful microprocessors leads to increasing deployment of those electronic devices in ever new application areas. Currently, the automotive industry considers the replacement of mechanical or hydraulic implementations of safety-critical automotive systems (e.g., braking, steering) by electronic counterparts (so-called "by-wire systems") for safety, comfort, and cost reasons.**

In order to remain operational in the presence of faults, these kinds of systems are built as fault-tolerant distributed real-time systems consisting of interconnected control units. To assure the correct operation of the fault tolerance mechanisms, software implemented fault injection provides low cost and easy to control techniques to test the system under faulty conditions.

In this paper we propose a distributed software implemented fault injection framework based on the mobile agent approach. Software agents are designed to utilize the real-time system's global time and messages to trigger the fault injection experiments. We introduce a lightweight agent implementation language to model the fault injection and the concerned system resources, agent migration and logging of the fault injection experiments.

We argue the feasibility of our approach by applying it to a "drive-by-wire" application. Here, incorporating the fault injection software in a mobile agent provides a clear separation of the fault injection software from the application software. Thus, the fault injection code is easily exchangeable – providing means to inject different faults – and can be removed smoothly from the production system after the test run is completed.

I. Introduction and Related Work

Distributed fault-tolerant computer systems are increasingly applied in the automotive domain to implement safety-critical applications. In the next few years, by-wire systems improving vehicle performance and safety features are planned to replace mechanical and hydraulic braking and steering systems [1], [2]. These automotive applications impose stringent requirements not only in trustworthiness attributes, such as reliability and survivability, but also in real-time.

Because of these stringent demands rather strict guidelines (e.g., MISRA C guidelines [3]) have to be adhered to when developing automotive software. The sole adherence to programming guidelines however is not even closely sufficient to ensure that the whole distributed system operates correctly in the presence of faults. Therefore the fault tolerance properties of the distributed system have to be evaluated as well using appropriate methods.

Among the numerous approaches established to evaluate fault tolerance properties, fault injection plays an important role. In particular software implemented fault injection (SWIFI) has gained popularity due to its flexibility and cost-efficiency [4]. Introducing a middleware layer above the communication layer provides effective fault injection in distributed systems. Dawson [5] proposes such a middleware layer for fault injection in TCP/IP protocols. Koopman [6] introduces such a layer for the automotive communication protocol CAN.

This paper proposes an innovative approach extending the software implemented fault injection approach by introducing mobile agents in the middleware layer [7]–[9]. For Internet applications, mobile agents are known for the following advantages [10]: (i) reduced bandwidth consumption because they can move computation to the data, (ii) flexibility because they do not require specific code on a remote node, (iii) suitability and reliability for mobile computing because they do not require continuous network connections. We are going to use the mobile agent approach to target the following goals:

a) Easy removal of the fault injection code: A clear separation between application code and fault injection code allows the removal of the fault injection code after completion of the test run.

b) Easy adaptation of the fault injection code: By encapsulating the fault injection code into a mobile agent, the code can be clearly separated and adapted in a flexible manner.

c) No explicit communication between the agents is required: The roaming concept allows autonomous operation of the mobile agents and the coordination of the distributed fault injection actions without message exchange between the agents.

In order to provide meaningful test results, testing issues like determinism, reproducibility and avoiding probe effects must be considered in the fault injection approach. In [11] an approach is proposed for the testing of distributed real-

time system which considers scheduling information as well as worst case execution times to avoid probe effects. Our approach addresses these problems in a similar manner. We assume that our target applications are implemented on time-driven operating systems (e.g., OSEKtime [12]) with a time-driven communication protocol like FlexRay [13] for safety-critical automotive applications or TTP [14], which is considered for use in the avionics sector.

Another main challenge is to implement an agent platform in the target field of bus devices since these devices have very limited resources in terms of CPU power and available memory. A Java-based agent platform addressing consumer devices has been introduced with LEAP (Lightweight Extensible Agent Platform based on JADE [15], [16]), and several lightweight Java virtual machines (JVM) are available for these devices, while Jbed [17] addresses the real-time field embedding the operating system in the JVM. Penya [18] proposes a new platform for embedded systems which is based on Java as well. In contrast, we propose to implement our own agent platform which is especially tailored to fault injection in by-wire systems. The main reasons are:

- There is no JVM available for our target system.
- There is only a limited functionality of a JVM required.
- Fault injection code requires specific means to address the above described requirements of determinism, reproducibility and avoiding probe effects.

The paper is structured as follows: Section 2 argues our concepts and approach in detail. Section 3 explains implementation issues and presents the implementation architecture and the agent implementation language. Section 4 shows an example application where this agent-based fault injection can be applied and discusses possible benefits and drawbacks.

II. Concepts and Approach

Using software implemented fault injection [4], we aim at testing the reliability and survivability attributes of the fault tolerance mechanisms implemented in automotive safety-critical distributed systems. The approach assumes that hardware failures caused by environmental phenomena effect the system software and thus can be detected at the software level. Table I shows both the fault hypothesis and the mapping between a possible cause of a fault and the perception on hardware and software layers.

The proposed fault injection approach is capable of injecting symmetric and asymmetric message loss and message mutilation, and bitflips in application data of the nodes' RAM. Bitflips in ROM – mainly resulting in fatal operating system errors – are not covered. The fault injection software manipulates messages and application data to inject the proposed faults. Hereby, access to the application's memory is required. This is usually possible in automotive embedded systems since the used devices do not provide memory protection. Furthermore, means to lookup the right address in memory are needed. Commonly, this information is provided by a so-called "MAP file" which is produced during compilation.

Typical Environmental Phenomena	**Hardware Failures**	**Failure Perception**
Physical Disconnection, Local Electro-Magnetic Disturbance, Short Circuit	Incoming Link Failure	Asymmetric Message Loss/Mutilation
Physical Disconnection, Local Electro-Magnetic Disturbance, Short Circuit	Outgoing Link Failure	Symmetric Message Loss/Mutilation
Physical Disconnection, Electro-Magnetic Disturbance, Short Circuit	Channel Failure	Multiple, Simultaneous Symmetric or Asymmetric Message Loss/Mutilation
Local Electro-Magnetic Disturbance	ROM Bitflips	Corrupted Application Data Structures, Arbitrary Application Behavior, Fatal Operating System Failures
Local Electro-Magnetic Disturbance	RAM Bitflips	Corrupted Application Data and Data Structures, Arbitrary Application Behavior, Fatal Operating System Failures

TABLE I
FAULT HYPOTHESIS

The fault injection software should be clearly separated from the system under test to allow easy removal after the test runs are completed. Encapsulating the fault injection experiment within mobile agents [7], [8] ensures that there is no code remaining on the node under test after the mobile agent has terminated or left the node. 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. In our approach we use one special mobile agent as an interpreter agent, which migrates to the node first and is used to execute the second type of mobile agents, the fault injection and monitoring agents. The results of the fault injection experiments are sent to an external evaluation platform.

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

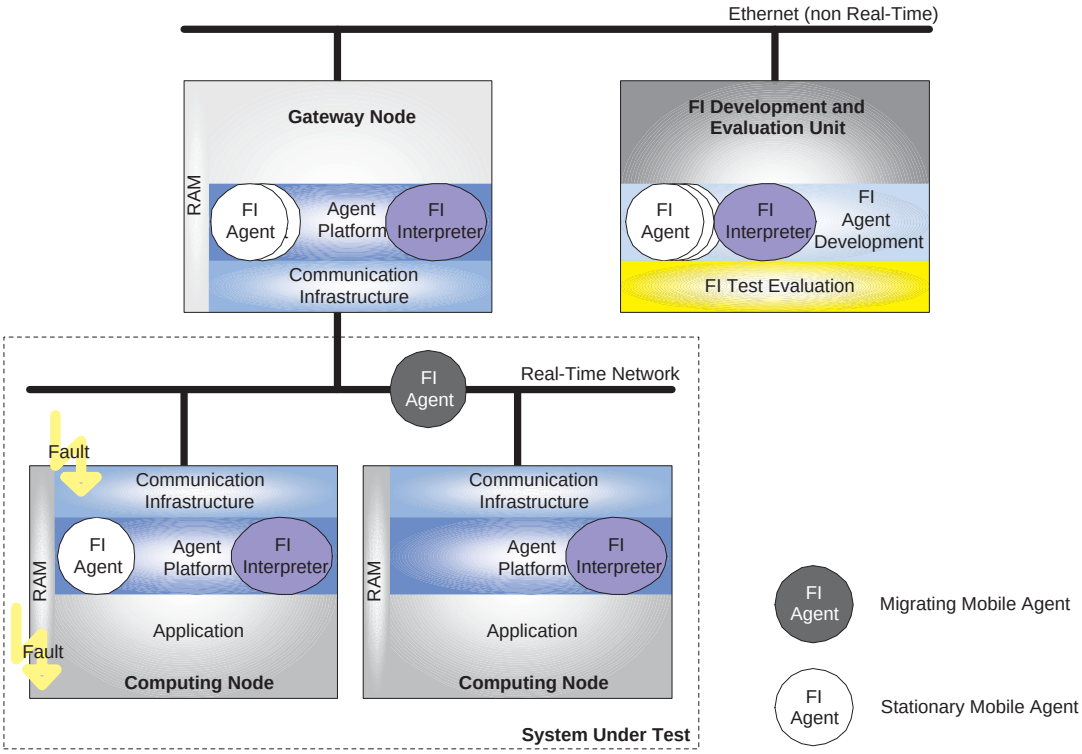

Fig. 1. Fault Injection Setup

real-time network and the determinism of the fault injection experiment, which allows to decide upon coordination issues before runtime.

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.

The targeted hardware platforms are characterized by limited processor power and limited memory capacity. Although they are comparable with consumer devices, like PDAs or mobile cell phones, they are not targeted by mainstream distributed systems development focusing on script and byte-code language support like e.g. Java Virtual Machines. The JVM size ranges from about 40 kB (KJava) up to 2.3 MB (fat version of Jeode) targeting hardware architectures like ARM, StrongARM, SH3/4 and similar ones [19]. Many platforms offer mobility support, like implementing J2ME MIDP [20], garbage collection and a set of Java classes, e.g. Java AWT for GUI. The features, however, are not needed for our small fault injection setup, and garbage collection makes it difficult to implement deterministic execution in terms of time. Following the requirement of keeping the software needed for fault injection as small as possible, we decided to develop a new fault injection language compiler and a lightweight interpreter capable of interpreting the byte-code of mobile agents.

III. Implementation of the Approach

In the following section the implementation approach is presented. After a short description of the hardware architecture and the development process with respect to the mobile agents, the principle of operation of the fault injection based on mobile agents is illustrated, and the agent implementation language is presented.

A. Hardware/Software Architecture

The overall architecture (as depicted in Figure 1) of our approach consists of one dedicated central fault injection development and evaluation unit, a gateway node, and several computing nodes.

The *computing nodes* (i.e., the nodes under test) execute the distributed real-time application and are interconnected via a real-time communication network which is used for message exchange according to a communication protocol with real-time capabilities. 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 development and evaluation unit* is used for the development of the fault injection agents, for the trig-

```
fault_scenario  ⟶  entry +
entry           ⟶  ON node trigger action +
node            ⟶  NODE number
trigger         ⟶  AT_TIME number | ON_MESSAGE message_name
action          ⟶  fi_action | migration | log_action | TERMINATE

fi_action       ⟶  message_loss | message_mutilation | memory_mutilation
message_loss    ⟶  LOSE_MESSAGE message_name
message_mutilation ⟶ CHANGE_MESSAGE_CONTENT message_name TO data
memory_mutilation ⟶ CHANGE_MEMORY_CONTENT location TO data

migration       ⟶  MIGRATE_TO node

log_action      ⟶  message_log | memory_log
message_log     ⟶  LOG_MESSAGE_CONTENT message_name
memory_log      ⟶  LOG_MEMORY_CONTENT location

location        ⟶  address | variable_name
address         ⟶  number

variable_name   ⟶  /* identifier */
message_name    ⟶  /* arbitrary character sequence */
data            ⟶  /* byte sequence */
number          ⟶  /* sequence of digits */
```

Fig. 2. Definition of the Agent Implementation Language

gering 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.

In the current implementation Ethernet with TCP/IP as communication protocol is used as non real-time network. FlexRay together with the FlexRay communication protocol is used as real-time network. As far as operating systems are concerned, Win2k is used on the fault injection development and evaluation unit, Linux with RTAI [21] real-time extension is used on the gateway node and the real-time operating system OSEKtime [12] is used on the computing nodes.

B. Development Process

In the field of safety-critical systems the actions performed by the system are planned and fixed at system design time. Commonly this 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 layers for message exchange including memory regions for the local storage of the message data (so-called "message buffers") and operating system configuration, like task dispatching tables, are generated in an automatic fashion.

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 *fault injection implementation language*, which is translated into a bytecode via the *fault injection language compiler*. This compile step is performed on the fault injection development and evaluation unit prior to the actual run-time. Hereby 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.

C. Principle of Operation

Prior to run-time instances of a special agent – the *interpreter agent* – migrate to each computing node in the real-time network. This interpreter agent performs the execution of the fault injection agents and incorporates an interpreter for the agents' bytecode. Since the interpreter agent itself cannot use an agent platform for its own execution, it has to be present in binary form and must be compiled for each different hardware platform of the computing nodes.

Once an instance of the interpreter agent has migrated to each computing node, the actual fault injection agents are allowed to migrate to these nodes and to perform their fault injection actions in an autonomous fashion.

The effects of the injected faults are perceived and logged by the agent on the local computing node. Since it is common practice in the automotive industry to use non-volatile memory (e.g., EPROM) for the storage of the loggged data, we decided to make use of this apporach as well. After the completion

```
ON NODE 0x1 AT_TIME 100 CHANGE_MESSAGE_CONTENT wheel_angle TO 0xAC
ON NODE 0x1 ON_MESSAGE failure LOSE_MESSAGE break_pressure
ON NODE 0x1 AT_TIME 300 CHANGE_MEMORY_CONTENT system_status TO 0x42
ON NODE 0x1 AT_TIME 2400 LOG_MESSAGE_CONTENT break_pressure
ON NODE 0x1 AT_TIME 2500 LOG_MEMORY_CONTENT 0x0815
ON NODE 0x1 AT_TIME 3000 MIGRATE_TO NODE 0x30
```

Fig. 3. Example Usage of the Agent Implementation Language

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.

For the migration of the agents and for the transfer of the logged data the ISO transport protocol [22] 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.

D. Agent Implementation Language

The structure of the agent implementation language follows an event-action scheme, where actions are triggered by predefined events. In addition, the mobile agents use node location information to control the context of the fault injection experiments. Figure 2 shows the lightweight grammar used for the mobile agents.

Triggered by time or on appearance of control messages the mobile agent decides on the action to perform. The action is either a migration to another node, the manipulation of messages or the mutilation of memory. Furthermore, the agent may log results of the fault injection experiment or terminate.

Hereby, for messages sent via the real-time network, the message instance name or the message name are unique symbolic names. These symbolic names are stored in the design database of the distributed system. After reception or prior to transmission the messages, as well as the message instances, are stored in the message buffers on the transmitting and receiving computing nodes. A unique mapping to the memory addresses of these message buffers is possible via the MAP file. The variable name is the name of a program variable, which again can be translated into a memory address using the MAP file.

The temporal trigger conditions are based on the global time (in units of microseconds) provided by the communication system. In case the global time reaches the time specified in a temporal trigger condition, the corresponding actions are invoked.

Figure 3 illustrates the agent implementation language via an example. Here the data of the message `wheel_angle` is changed to `0xAC` at time `100` on node `0x1`, afterwards the fault injection software simulates a loss of the message `break_pressure` upon the reception of the control message `failure`, and furthermore changes the value of the variable `system_status` to `0x42` at time `300`. In the following, at time `2400` the contents of the message `break_pressure` and at time `2500` the contents of the memory cell at address `0x0815` are logged. Finally, the agent is instructed to migrate to node `0x30` at time `3000`.

IV. Evaluation

In this section an evaluation of the presented approach is made based on an example application from the automotive domain.

A. Example Application

For the evaluation of the presented framework, a distributed real-time system for an automotive application – the *drive-by-wire application* – is used. This distributed system used by this application consists of 15 computing nodes interconnected via the FlexRay communication system. Four of these 15 nodes – the so-called *steering nodes* – take care of aligning the road wheels according to the computed steering angle. This steering angle is derived from the driver's input, which is obtained via a side stick by four so-called *stick nodes* and contains the desired change of the vehicle velocity (accelerate, decelerate) as well as requested direction of movement (left, right). Four *brake nodes* are responsible for applying the desired change in vehicle velocity to the road wheels. For the supervision and the management of the different power sources (i.e., battery and alternator) three *power nodes* are used. – Figure 4 illustrates the architecture of the drive-by-wire application.

For the implementation of the nodes of the drive-by-wire application, ES1600 boards developed by ETAS are used. These boards are of single Europa size and can be plugged into a VME backplane for reasons of power-supply and communication.[1] Each of these boards hosts an MPC555 microcontroller clocked at 40 MHz, providing 2 MB of external SRAM and 4 MB of external flash EEPROM.

Each node hosts two FlexRay communication controllers, which are located on a dedicated adaption board of the ES1600. Hereby one of the two FlexRay communication controllers is connected to two FlexRay communication channels, whereas the second FlexRay communication controller is only connected to a single communication channel. Thus this setup provides a total of three communication channels.

The communication via the communication channels is performed in a static cyclic fashion using the FlexRay communication protocol. The duration of a communication cycle[2] is 8 milliseconds.

B. Fault Hypothesis and Degree of Fault Tolerance

Both the computing nodes as well as the communication channels in the drive-by-wire application are assumed to

[1]The drive-by-wire application does not make use of the VME bus. The VME backplane is solely used for means of power-supply.

[2]During a communication cycle all messages are exchanged between the different nodes.

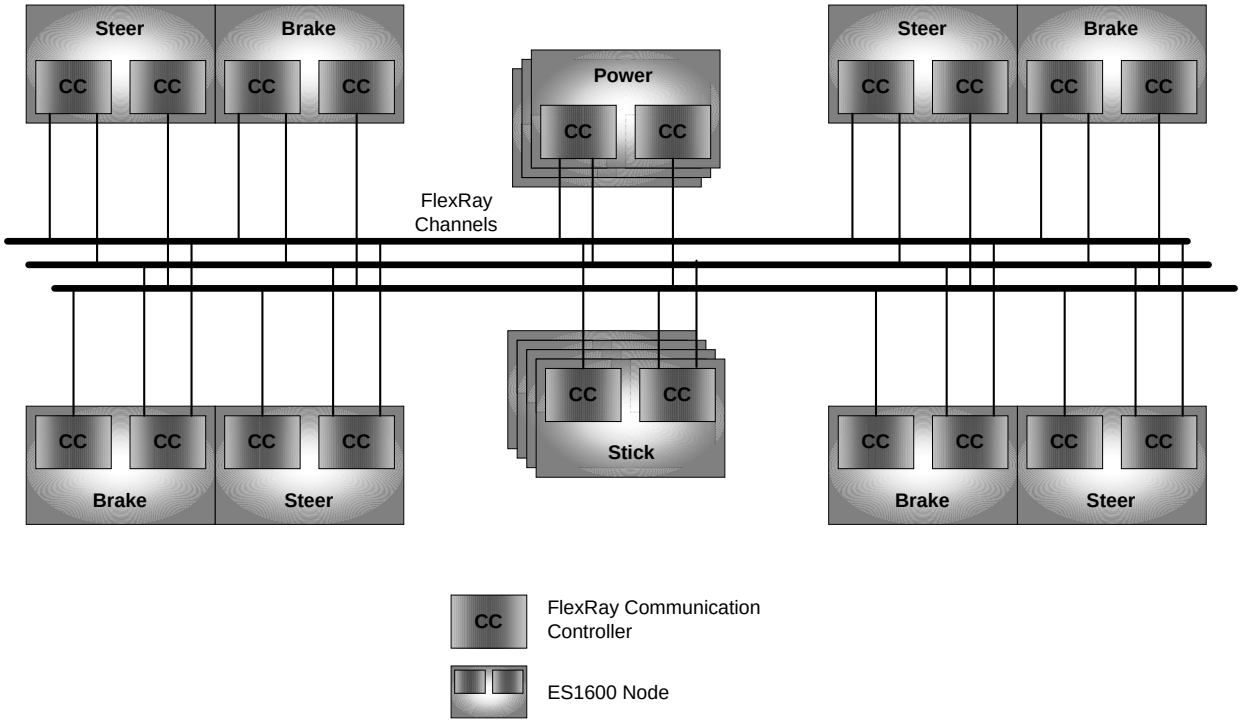

Fig. 4. Architecture of the Drive-By-Wire Application

exhibit only one failure mode: fail silent. The components either produce correct results, detectable invalid results or no results at all.

As far as fault tolerance is concerned, the drive-by-wire application is designed to be able to tolerate the failure of two arbitrary computing nodes, two arbitrary communication channels, and two arbitrary communication links.

C. Applying the Agent-Based SWIFI Approach

The inherent safety-criticality of the drive-by-wire application requires the testing of its fault tolerance mechanisms and redundancies in the presence of faults. Injecting these fault via physical fault injection is hard to control. In particular the injection of asymmetric faults is hard to do by means of physical fault injection. Using software implemented fault injection however provides a much more controlled means to perform the fault injection. In particular the previously presented mobile agent-based approach is well-suited for this task.

With the presented approach, asymmetric faults can be injected via multiple agents located on different computing nodes by mutilating or losing the messages in the message buffers of a subset of the receiving nodes.

If symmetric faults are to be injected, message loss or message mutilation is either injected at the sending node or at all receiving nodes.

The coordination of the mobile agents is mainly done by utilizing the global time provided by the FlexRay communication system (i.e., each agent injects the faults at pre-defined points in time). Only the external fault injection development and evaluation unit may send additional control messages to the mobile agents. In this way the communication overhead is reduced and a temporal agent coordination in the microsecond range is achieved. This is a fact which is important in the drive-by-wire application due to its rather small cycle time.

After completion of the test runs, the agents can smoothly be removed from the volatile SRAM of the ES1600 by performing a simple power cycle (power down followed by power up) of the respective node.

V. Conclusion

In this paper we introduced a novel method of a software implemented fault injection approach that exploits the concepts of mobile agents for testing reliability and survivability attributes of distributed fault-tolerant real-time systems. Therefore we proposed a customized agent-based platform that can be applied under the extreme limited resource capabilities of automotive by-wire systems. The mobile agent-based platform resides within the middleware layer between communication and application supporting deterministic fault injection simultaneously on different nodes of the distributed system. To achieve an efficient implementation, we introduced a specific fault injection language that meets the restricted resource capabilities of the automotive target systems and allows access to system resources utilizing the mapping information generated at compile time. We applied our approach to a drive-by-wire system to argue its feasibility.

We see the benefits of this testing technique – encapsulating the whole fault injection code into a mobile agent – also applicable for other trustworthy systems: no explicit message

passing is required during test experiments, the fault injection code can easily be removed after the test experiments, and a high flexibility for the test experiments is achieved due to easy adaptation of the mobile agent code.

REFERENCES

[1] G. Leen and D. Hefferman, "Expanding Automotive Electronic Systems," *IEEE Computer*, pp. 88–93, Jan. 2002.

[2] X.-B.-W. Consortium, "X-By-Wire: Safety Related Fault Tolerant Systems in Vehicles," X-By-Wire Consortium, Stuttgart, Germany, Tech. Rep. XByWire-DB-6/6-25, 1998.

[3] P. Edwards, S. Fischer, G. McCall, D. Newman, F. O'Neill, R. Pearman, and R. Rivett, "Guidelines for the Use of the C Language in Vehicle Based Software," The Motor Industry Software Reliability Association, Tech. Rep., Apr. 1998.

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

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

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

[7] 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.

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

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

[10] G. Cabri, L. Leonardi, and F. Zambonelli, "MARS: A Programmable Coordination Architecture for Mobile Agents," *IEEE Internet Computing*, vol. 4, no. 5, pp. 26–35, July 2000.

[11] H. Thane and H. Hansson, "Testing Distributed Real-Time Systems," *Journal of Microprocessors and Microsystems*, vol. 24, pp. 463–478, 2001.

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

[13] 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 *Proc. SAE 2001 World Congress*. Detroit, MI, USA: Society of Automotive Engineers, Mar. 2001.

[14] H. Kopetz and G. Grünsteidl, "TTP – A Protocol for Fault-Tolerant Real-Time Systems," *IEEE Computer*, pp. 14–23, Jan. 1994.

[15] F. Bergenti and A. Poggi., "LEAP: A FIPA Platform for Handheld and Mobile Devices," in *Proc. Agent Theories, Architectures, and Languages Conference (ATAL-2001)*, Aug. 2001, pp. 436–446. [Online]. Available: http://leap.crm-paris.com/

[16] F. Bellifemine, A. Poggi, and G. Rimassa, "JADE – A FIPA-Compliant Agent Framework," Centro Studi e Laboratori Telecomunicazioni S.p.A., via G. Reiss Romoli 274, 10148 Torino, Italy, Tech. Rep., 1999. [Online]. Available: http://jade.cselt.it/papers/PAAM.pdf

[17] esmertec inc., "Jbed ME – Java Technology for Small Handheld Devices," esmertec, inc., Lagerstrasse 14, CH-8600 Duebendorf, Switzerland, Tech. Rep., 2002. [Online]. Available: http://www.esmertec.com/download/pdf/Jbed_ME_White_Paper.pdf

[18] Y. K. Penya, S. Mahlknecht, and P. Rössler, "Lightweight Agent Platform for High-Performance Fieldbus Nodes," in *Proc. 4th IEEE International Workshop on Factory Communication Systems*, 2002. [Online]. Available: http://www.mrtc.mdh.se/WFCS2002/

[19] J. Backus and N. Snijder, "Java Gets Real for Embedded Software Development," *Real-Time Magazine*, vol. 3, pp. 40–44, July 1999. [Online]. Available: http://www.realtime-info.be/magazine/99q3/index993.htm

[20] JSR 118 Expert Group, "Mobile Information Device Profile for Java 2 Micro Edition," Sun Microsystems, Inc., Tech. Rep., Nov. 2002. [Online]. Available: http://java.sun.com/j2me/

[21] E. Bianchi, L. Dozio, and P. Mantegazza, *DIAPM-RTAI – A Hard Real Time Support for LINUX*, Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano, Campus Bovisa Sud, via La Masa 34, 20156 Milano, Italy.

[22] ISO (International Organization for Standardization), "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.