Dependability of COTS Microkernel-Based Systems

Jean Arlat, *Member*, *IEEE*, Jean-Charles Fabre, *Member*, *IEEE Computer Society*, Manuel Rodríguez, and Frédéric Salles

Abstract—The commercial offer concerning microkernel technology constitutes an attractive alternative for developing operating systems to suit a wide range of application domains. However, the integration of COTS microkernels into critical embedded computer systems is a problem for system developers, in particular due to the lack of objective data concerning their behavior in the presence of faults. This paper addresses this issue by describing a prototype environment (MAFALDA: Microkernel Assessment by Fault injection AnaLysis and Design Aid) that is aimed at providing objective failure data on a candidate microkernel and also improving its error detection capabilities. The paper first presents the overall architecture of MAFALDA. Then, a case study carried out on an instance of the Chorus microkernel is used to illustrate the benefits that can be obtained with MAFALDA both from the dependability assessment and design-aid viewpoints. Implementation issues are also addressed that account for the specific API of the target microkernel. Some overall insights and lessons learned, gained during the various studies conducted on both Chorus and another target microkernel (LynxOS), are then depicted and discussed. Finally, we conclude the paper by summarizing the main features of the work presented and by identifying future research.

Index Terms—COTS microkernels, dependability characterization, fault injection, error confinement, wrapping.

1 Introduction

Microkernel technology is very appealing today in the design and implementation of either general purpose operating systems (OSs) or specific ones targeted at a given application domain. This technology provides basic components ("componentized" architecture) on which or from which OS layers can be easily implemented. These software components can be obtained "off-the-shelf" since today many vendors provide microkernels. It is up to the system developer integrating such COTS components to develop dedicated upper layer software. This is the current trend in OS development, including systems with stringent dependability requirements such as embedded systems aiming at critical applications (e.g., railway, space, avionics, etc.).

The design issues involved when considering to incorporate a microkernel for developing a given system depend, of course, on several considerations. They include: the functionalities offered by the components, the flexibility to customize the microkernel, the performance of the interaction between the application processes and the microkernel, the various addressing modes and related protection mechanisms, etc. Still, the decision to use COTS microkernels in critical systems raises additional concerns related to dependability issues. Accordingly, the assessment of the behavior of these components in the presence of faults is of high interest since upper software layers have to deal with the failure modes of the selected microkernel.

Clearly, the ability to get insights on their failure modes is a major data entry, not only by contributing to the decision whether to select a given microkernel, but mainly for guiding the subsequent design decisions, including the incorporation of additional protection mechanisms.

Accordingly, we have developed a methodology, supported by a tool MAFALDA (*Microkernel Assessment by Fault injection AnaLysis and Design Aid*), whose aim is twofold:

1. To provide objective quantitative data regarding the failure modes of a candidate microkernel. Using fault injection at various levels, MAFALDA enables to establish a better understanding of the microkernel behavior in the presence of faults.
2. To improve the failure modes depicted by the assessment of the faulty behavior. Such an improvement relates to the addition of error detection mechanisms to the microkernel. This involves using *wrappers* for checking the proper behavior of the functional components of the microkernel.

Although these goals and the techniques used by MAFALDA are not innovative by themselves, to the best of our knowledge, there are at least three main features that should be pointed out that make MAFALDA an original and unique experimental environment. These features encompass the provisions of: 1) a coherent support for implementing both fault injection and wrapping capabilities, 2) fault injection mechanisms for corrupting both the kernel calls (robustness testing) and the memory segments of the target microkernel, 3) the analysis of error propagation among the target functional components of the target microkernel.

- *J. Arlat, J.-C. Fabre, and M. Rodríguez are with LAAS-CNRS, 7, Avenue du Colonel Roche, 31077 Toulouse Cedex 4, France. E-mail: {arlat, fabre, rodriguez}@laas.fr.*
- *F. Salles was at LAAS-CNRS. He is now with Sun MicroSystems, Inc., Mail Stop UMPK 16-204, 16 Network Circle, Menlo Park, CA 94025. E-mail: frederic.salles@sun.com.*

Manuscript received 15 Oct. 2000; revised 16 May 2001; accepted 19 June 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 114408.

This paper proposes a comprehensive exposition of the various facets of the methodology supported by MAFALDA: failure mode assessment [1], enhancement of error detection [2], implementation issues [3], and usage and lessons learned [4]. It also covers additional features concerning, for example, the statistical monitoring of the conduct of a fault injection campaign, and some further insights gained from the work carried out on the two target microkernels that were considered, namely Chorus and LynxOS.

The organization of the paper is as follows: Section 2 motivates the work and introduces the overall methodology. Then, Section 3 briefly describes the main features that support both the assessment and design-aid facets of MAFALDA. An instance of the Chorus microkernel[1] is used to illustrate the benefits that can be obtained from the fault injection (Section 4) and wrapping (Section 5) facilities offered by MAFALDA. Some implementation issues are addressed in Section 6. Section 7 provides some further insights and lessons learned concerning the fault injection experiments carried out on another target microkernel (LynxOs). Finally, Section 8 draws conclusions and introduces some on-going work.

2 Problem Statement and Overall Methodology

When considering whether to use COTS OSs in critical systems, developers are faced with serious certification problems. It is noteworthy that recent specific efforts for developing the OSE real-time operating system [5] have succeeded in its certification according to the IEC 61508 [6]. Although certification aspects are beyond the scope of this paper, we claim that the framework we propose provides a complementary and promising approach for the assessment and improvement of the behavior of microkernel-based software executives in the presence of faults. We advocate that this framework bridges the gap between the stringent "full visibility" requirement currently imposed by most standard documents (including for COTS software) and the usual lack or poor observability and controllability attached to COTS components. Indeed, it can objectively contribute to the certification process and the evolution of standardization documents.

Because COTS components are not usually designed according to a development methodology recommended for critical software, their behavior may be weak from a dependability viewpoint. This is not surprising at all since their normal usage is often general. Because their size is to be kept small and they are a basic support for OS layers, many design options (in particular concerning error signaling) are left open and the error confinement capabilities must be enhanced at upper layers. Furthermore, OSs are used in different target environments with different application profiles, so some classes of elementary functions may seldom be activated or even not activated at all. Accordingly, some error propagation channels could remain unknown. Finally, because software versions usually have a very short lifetime, it is difficult to get enough confidence only on the basis of statistics from field experience.

In the sequel of this section, we first discuss the main issues attached to the microkernel technology for developing specialized and efficient OSs, then we introduce the main features of the overall methodology.

2.1 Microkernel Technology

Microkernels contribute attractive design solutions for developing OS layers tailored to fit the functional requirements of various application domains. These executive layers can provide the application programmers with either a standard API (Application Programming Interface), such as POSIX, or a proprietary interface, as in the ELEKTRA railway signaling system (in particular, for being able to run legacy software) [7]. Moreover, the provision of a "componentized" architecture is typical of the most recent generation of microkernels. They feature basic components, each providing elementary services: synchronization, scheduling, memory, communication, etc. Examples of such microkernels include Chorus [8], LynxOS [9], and VxWorks [10].

One important distinctive dimension when considering the microkernels is the way application and middleware layers interact with the microkernel software. Two main cases can be identified:

1. The microkernel software is provided as a set of *library* functions to be combined at compile or load time to the application/middleware software. In this case, the resulting standalone software executes in a single address space, normally in supervisor mode. For instance, this was the case for the early version of VxWorks.
2. The microkernel software is running in a separate address space (usually in supervisor mode), while the application/middleware software is running in user mode. The interaction between the application processes and the microkernel software, i.e., the calls to the microkernel, is often implemented by a *trap* mechanism. This is the case for Chorus and LynxOS.

The main distinction between microkernels implemented only as a library of basic services or implemented as a true kernel whose internal functions can only be accessed through a trap mechanism is depicted in Fig. 1. However, the actual distinction is not always that clear since, in the latter case, kernel calls are also provided to upper layers as libraries. In such a case, part of the call is executed in the user address space and the trap mechanisms invoke internal functions in the supervisor address space. The invocation of the internal functions is accompanied by a change of the addressing/running mode. Clearly, this approach has an impact on the performance of the kernel calls (context switch vs. simple jump).

This distinctive feature has a strong impact as far as dependability is concerned. Clearly, the library-based interaction does not ensure that a deviant application process will not corrupt other application processes and even the microkernel internal functions shared between all application processes. Thanks to the clear distinction

1. The version used in the experiments carried out is no longer distributed by the provider.

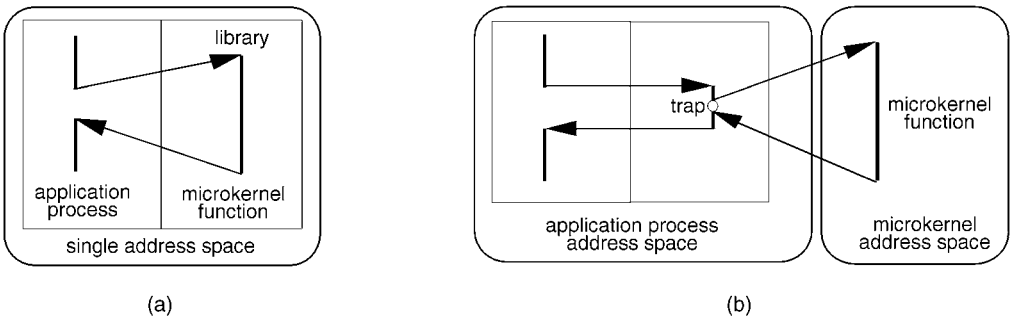

Fig. 1. (a) Library-based microkernels vs. (b) trap-based microkernels.

between kernel space and application address spaces, the trap-based interaction basically can provide enhanced error detection and confinement, by simply using hardware-based mechanisms—mainly, the memory management unit (MMU). Another aspect deals with the control of the invocation of the microkernel functions: The trap-based microkernels have a single entry point into the microkernel address space that can be used for both fault injection and wrapping of the microkernel components.

2.2 Microkernel Fault Pathology

The confidence one can have in a microkernel depends on the impact of its behavior in the presence of faults on the upper layers. The origin of a fault can be physical or due to design or implementation deficiencies.

As for any software component, an activated fault that affects the microkernel can propagate and lead to various types of failures (according to the causal chain $fault \rightarrow error \rightarrow failure$ [11]) which may temporarily or permanently impair the delivered services. Three main outcomes can be identified, whatever the perturbation affecting the microkernel (either an erroneous kernel call or an internal fault):

1. *Detected error*: The perturbation is successfully identified and reported by the return of a kernel call or at least an exception is raised to the application layer.
2. *Kernel crash*: The microkernel "hangs," e.g., the microkernel enters an infinite loop or waits for a fictitious event.
3. *Propagated error*: Neither the internal error detection mechanisms nor the intrinsic behavior of the microkernel can ensure that the error will be confined within the executive software layer and, thus, its effects may propagate to the application layer.

Of course, the impact of these outcomes on the application layer depends on the specification of the system and on the level of redundancy implemented at this layer. When no service interruption can be afforded, then all three outcomes may well lead to a *failure*. However, whenever error processing can be carried out at this upper layer, it is likely that the ability to detect errors will be of great help. The impact of a kernel crash is also very much application dependent. Besides, it can be related to a "fail-silent" behavior of the microkernel, which exhibits some advantages from a dependability viewpoint [12]; still, such crashes need to be handled by external mechanisms, i.e., fault tolerance strategies based on (preferably diversified) replication (e.g., see [13]). Concerning errors propagating to the application layer, in practice, this outcome is usually further refined by distinguishing two cases: application hang and application failure, where the latter is of the highest severity level.

Obtaining objective information on the failure modes of a specific microkernel candidate is definitely of main interest for the system developers to support their design decision whether to integrate the microkernel. Also, getting objective and detailed insights on the faulty behavior of the microkernel is a form of input that is of main interest for the microkernel provider. An example of such insights concerns the ability to identify and evaluate possible error propagation channels between the functional components of the microkernel. Indeed, a call to the microkernel from an application software process involves many interactions between the microkernel components and, thus, the error can propagate to several application processes.

2.3 Fault Injection-Based Assessment

As it allows for a detailed analysis of faulty behaviors, fault injection has long been recognized as a pragmatic and efficient means to assess the behavior of complex fault-tolerant computer systems (e.g., see [14], [15], [16]).

It is important to point out that what really matters when probing the behavior of a target system in the presence of faults is much less the faults themselves than the consequences provoked/observed. Indeed, in practice, similar error patterns often originate from various distinct causes (faults). Such a matter of fact is one main driver for the emergence of the so-called *software implemented fault-injection* technique (SWIFI, in short).

SWIFI basically implements the "bit-flip" fault model at the level of memory words or registers, as well as at the level of data exchanged between software components. The question of the representativity of this fault model, compared to the pathology induced by the occurrence of real faults, is still not fully established. However, it is worth noting that what really matters here is much less the actual match between injected faults and real faults, but rather the match of the error patterns provoked by these respective faults.

It is clear that this issue is still a matter of current research. Nevertheless, several published studies have investigated and evidenced the pertinence of SWIFI-induced bit flips

with respect to real faults, e.g., see [17], [18], [19], [20], [21]. In particular, the SWIFI technique is well suited to simulate errors induced by space radiations (also known as *single event upsets*) [22]. In addition to the suitability for simulating such transient hardware faults, a recent study showed that bit flips are also able to simulate (to some extent) the consequences of software faults [23]. A comprehensive discussion on the pros and cons of the SWIFI technique can be found in [24].

Fault injection is indeed very much suited to helping characterize the faulty behavior of COTS executives. The work reported in [25] compares the robustness of 15 off-the-shelf POSIX-compliant OSs, using the Ballista tool. The basic idea is to corrupt kernel call parameters (using predefined wrong values) and analyze the consequences according to the so-called "CRASH" scale.[2] FINE [26] and its distributed variant DEFINE [27] were developed for assessing the dependability properties of the SunOS 4.1.2 kernel with respect to both software and hardware faults. It is based on the injection in the memory space of the target OS.

These two forms of corruption are actually complementary. Accordingly, in our work, we have retained the following two forms of fault injection to analyze the behavior of a microkernel in the presence of faults: 1) injection on the parameters associated with microkernel *primitives*[3] and 2) injection on the microkernel address space, focusing either on code or data segments. Another dimension is related to the way the parameters and memory locations to be faulted are selected, either deterministically or randomly. Here again, we have kept the two options open for the MAFALDA environment. In particular, we claim that random fault selection offers a pragmatic and easily portable injection approach.

Accordingly, the objectives of the fault injection-based assessment are twofold:

1. Statistics obtained can be used to evaluate the coverage of the error handling mechanisms.
2. Deficiencies can be revealed in the microkernel (including in the error handling mechanisms), thus providing guidance for design improvements; this also encompasses the definition of an appropriate fault tolerance upper layer.

In order to improve the failure modes of a COTS microkernel, the MAFALDA framework enables the development of additional error detection and confinement mechanisms based on wrapping techniques.

2.4 Wrapping-Based Error Detection

Trying to detect all possible errors leading to the corruption of upper layer software is a difficult issue and certainly almost impossible as far as COTS software is considered. Preventing such faults from impacting the upper layers can only be achieved by external mechanisms. This claim leads to the notion of *wrapper*, originally introduced to deal with security concerns [28].

2. The CRASH acronym stands for these five robustness failures: Catastrophic, Restart, Abort, Silent, and Hindering.

3. A *primitive* designates an elementary service accessible via the microkernel API that is invoked during a *kernel call*.

The incorporation of defense through wrapping in order to check component's functionality, was described in [29] regarding off-the-shelf application software. The author distinguishes input and output wrappers, the former preventing certain inputs from reaching the component, the latter verifying constraints on the output before releasing it. Input wrappers can be seen as filters of some syntactically incorrect invocation pattern, while output wrappers rely on the semantics of the expected behavior.

The wrappers we are advocating in this paper are based on the well-established notion of executable assertions (e.g., see [30], [31], [32], [33]). The semantic verification is based on a (formal) model of the expected behavior of the microkernel functional components. Inputs, outputs, as well as the internal state of the component, impact the behavioral model on which predicates are verified at runtime.

The possibility of establishing in practice such a model depends on: 1) the complexity of the target component and 2) the level of abstraction needed to obtain a realistic model. Clearly, modeling all functional classes is almost impossible for general-purpose COTS OSs. However, this objective is actually reachable in the case of most COTS microkernels, in particular thanks to their componentized architecture. These functional components (basically: synchronization, scheduling, memory, and communication) feature specifications that are easy to understand and, thus, amenable to a formal expression of their expected behavior.

In practice, the verification of the assertions goes beyond mere input and output filtering. This requires better observability of the internal state/behavior of the microkernel support error detection. In turn, these enhanced detection capabilities may lead to some corrective action that would require increased controllability of the COTS microkernel. One original dimension of the development of the wrappers we propose is related to their efficient implementation using the notion of *behavioral reflection* [34] to facilitate both the observability and controllability over the microkernel components.

3 The MAFALDA Environment

After a short exposition of the objectives that have led to the design of MAFALDA, we present the main assumptions considered for selecting the fault models supported by the tool. The modules that compose the architecture of MAFALDA are then described. Finally, the main steps of conducting a fault injection campaign with MAFALDA are briefly identified.

3.1 Design Guidelines and Objectives

The first objective is the characterization and assessment of the behavioral profile of a given COTS microkernel instance in the presence of faults. This objective is twofold: 1) evaluation of the robustness of its interface with respect to external faults and 2) evaluation of the coverage of its internal error detection mechanisms. The first aspect is related to the aptitude of the microkernel to cope with an error propagating from the application level of the system through its interface. The second aspect is related to the assessment of its own error detection mechanisms.

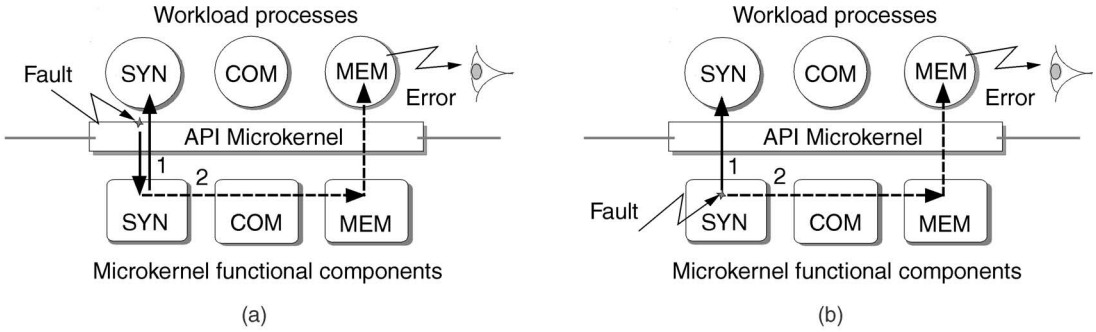

Fig. 2. Modular activation and observation of errors. (a) External injection. (b) Internal injection.

Accordingly, two forms of fault injection are being considered: 1) corruption of the input parameters during the invocation of a microkernel primitive and 2) corruption of the code and data segments of the microkernel. We map these corruptions to the functional decomposition of the internal architecture of the microkernel. This allows for two original forms of assessments, concerning:

- the robustness and error detection for each functional component of the microkernel,
- the error propagation between the functional components.

MAFALDA thus provides a modular workload made up of a set of application processes, each targeting the primitives supported by a single basic component: synchronization (SYN), communication (COM), memory management (MEM), etc. (e.g., see Fig. 2). The analysis of the behavior and data produced by the workload processes provide specific insights for the corresponding microkernel components. This scheme also facilitates the observation of the error propagation between the components.

Fig. 2 illustrates the possible error propagation that may occur when a fault is injected either at the level of the microkernel API, targeting the parameters of the kernel calls from a workload process to a specific microkernel component (Fig. 2a), or in the memory space of the functional component (Fig. 2b). In both cases, it is more likely that the error primarily affects the corresponding workload process (Case 1 in either portion of the figure). The observation of a failure on a distinct workload process than the one corresponding to the microkernel component targeted by the injection is then interpreted as the consequence of the propagation of an internal error within the microkernel (Case 2 on either portion of the figure).

As already identified, the second major objective is to improve the behavior of the microkernel from a dependability point of view. This objective builds up on the results of the behavioral assessment for providing guidance and support to contain the effect of both external and internal faults that may affect the behavior of the target microkernel. This has been achieved by the definition and implementation of enhanced wrappers (see Section 2.4).

3.2 Fault Assumptions and Fault/Error Model

Two forms of injection can be associated with each of the fault models identified to support the two forms of assessment of the target microkernels:

1. The evaluation of the robustness of microkernel's API corresponds to the analysis of the propagation of an error from the application level to the executive level. This approach consists in the corruption on a (pseudo)randomly selected byte, among the byte chain that constitutes the set of parameters passed to the microkernel during the invocation of one of its primitives. Only kernel calls to the selected functional component tested are susceptible to corruption, while the other calls remain unchanged.
2. The second form of injection has for an objective the simulation of the occurrence of a physical fault (consecutive to the failure of the physical support of the system) or a software fault (design or development) into the microkernel. Here again, the injection process consists in the corruption of a (pseudo)randomly selected byte within the address space of the targeted functional component, in its code segment, as well as in its data segment.

For both forms of injections, the applied binary operation consists of performing one (or several) bit flip(s) on the designated byte. It is important to note that bit-flips are only maintained until their activation, much like an error provoked by a transient fault. This is in accordance with the model supported by most SWIFI tools. Indeed, although the tool also supports the injection of permanent faults, we are focusing on transient faults, which are reportedly more likely.

3.3 Architecture

Fig. 3 illustrates the general architecture and the different modules of MAFALDA. Two main sets of machines can be identified:

- a rack of 10 *Target Machines* (Intel Pentium-based PC boards) running the target microkernel (the 10 machines are intended to speed up the fault injection campaigns),
- a *Host Machine* (a Sun Workstation) that defines, executes, controls the experiments carried out on the target machines, and, finally, analyzes the results.

Here, we simply present the main attributes that characterize MAFALDA; a more comprehensive description, in particular of the GUI, can be found in [4].

On each *Target Machine*, *Workload processes* (Wpi) are responsible for focusing the activity on a specific

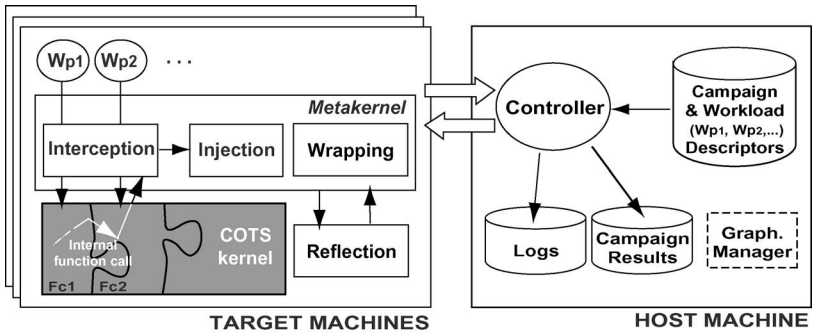

Fig. 3. MAFALDA architecture.

microkernel functional components (Fci). Four software modules are also embedded: the *Interception*, *Injection*, *Wrapping*, and *Reflection* modules. The first three are encapsulated by the *Metakernel*. The metakernel relies on the *Reflection* module for providing the suitable level of observability (resp. controllability) from (resp. upon) the target microkernel.

Both the *Interception* and *Injection* modules contribute to the fault injection process. The *Interception* module traps the relevant events (kernel calls though the kernel API and internal service calls among the kernel functional components). These modules control, respectively, the injection on kernel call parameters (robustness testing of the microkernel) and on memory locations storing the code and data of the functional components of the target microkernel. The *Injection* module implements two forms of corruptions (bit-flips on either kernel call parameters or kernel code and data segments).

In addition to the characterization of the faulty behavior and failure modes of the target microkernel, MAFALDA also supports the implementation of wrappers that are meant to enhance the error detection capabilities of the target microkernel. This is achieved by means of the *Wrapping* module. This module constitutes a generic support for the implementation of error detection and confinement wrappers via the runtime verification of executables assertions characterizing the expected behavior of the functional components of the target microkernel.

The *Reflection* module is meant to both monitor and act upon the pertinent variables of the target microkernel whose access is necessary for an efficient implementation of both the fault injection and error confinement processes. For example, the invocation of microkernel primitives plays the role of injection trigger in the case of fault injection on the parameters of kernel calls; accordingly, such an event must be systematically intercepted. In addition, wrapping requires that all events prone to inducing the update or the verification of an executable assertion (e.g., microkernel primitive invocation, return from invocation, and, even, call to some internal function) be monitored. The enhanced controllability provided by the *Reflection* module can also facilitate the implementation of error recovery actions that can be undertaken after a wrapper signals that an assertion has been violated.

On the *Host Machine*, the *Controller* is devoted to the execution and coordination of the fault injection campaigns. The parameters characterizing the campaign are stored in two files: 1) the *Campaign Descriptor* that defines the target functional components (Fci) and the fault model to be considered for the fault injection campaign and 2) the *Workload Descriptor* that contains the code for the workload processes (Wpi) executed during the experiments.

The *Controller* interprets these files and extracts all the necessary information to 1) set up the *Interception* and *Injection* modules and 2) install the workload processes on the Target machines. The experiment readouts and measures are stored into files: the *Logs* files and the *Campaign results* files.

During an experiment, the *Logs* file collects the raw outcomes produced by the modules located on each target machine to carry out the campaign:

- the reports of injection returned by the *Injection* module (e.g., injection time, location, fault type, fault activation),
- the events notified by the error detection mechanisms of the hardware platform or of the microkernel (e.g., exceptions, kernel debugger, etc.),
- the error detection notification by the *Wrapping* module (assertion violation, error propagation),
- the outputs produced during the execution of the *Workload processes*, including normal (results of computation) and abnormal (notification of an application failure) outcomes.

At the end of the campaign, an analysis of the data stored in the *Logs* files can be carried out, in particular to diagnose the possible failure of a *Workload process*. This information, as well as a short synthesis of the most significant campaign attributes (e.g., error latencies, error messages), is then stored in the *Campaign Results* files. These experimental measures can be displayed by a spreadsheet *Graph Manager* (e.g., *Microsoft Excel*). It is also worth pointing out that all logged information can be used later for more elaborate custom processing and analyses.

3.4 Fault Injection Campaign

Fig. 4 summarizes the main steps of a fault injection campaign.

First, the set of workload processes is executed in the absence of a fault so as to collect a trace of their behavior as well as the results produced under normal operation. In addition to output data, this trace includes internal values and process scheduling. These data constitute the *Oracle* used a posteriori to diagnose the propagation of an error to

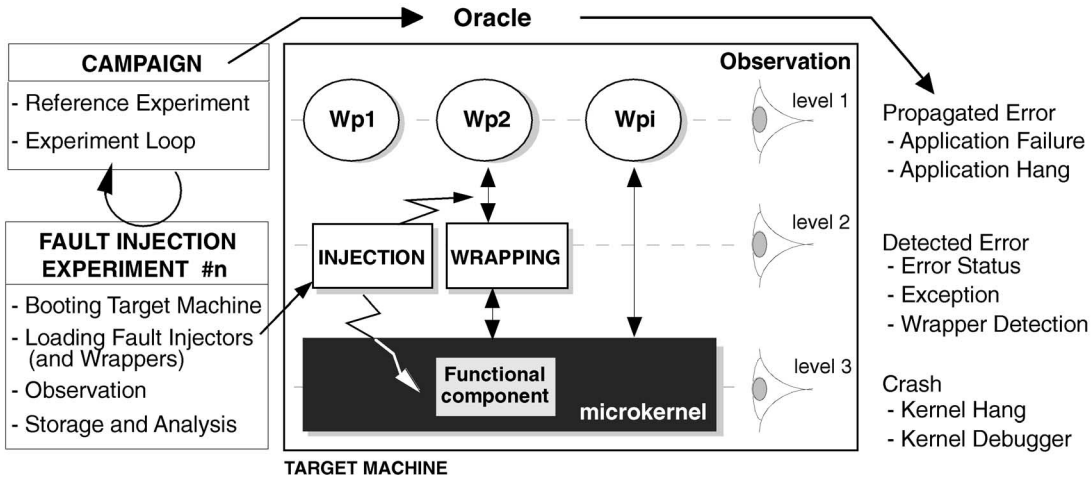

Fig. 4. Scheduling of a fault injection compaign.

the application level. After this initial step, the *Controller* executes the series of fault injection experiments as specified in the *Campaign Descriptor* file. For each experiment, the *Target Machine* is rebooted, thus avoiding any perturbation due to the potential presence of a residual erroneous state from the previous experiment. Then, the various workload processes (Wpi) responsible for the activation of the microkernel components are executed concurrently.

Depending on the type of injection, a single transient fault is injected, that affects the targeted functional component, through either its interface (corruption of an incoming parameter from the corresponding workload process) or its internal state (corruption of a memory location).

The last step consists of the observation of the target system behavior. The various outcomes and traces produced are collected for the duration of the experiment.

The different failure modes of the candidate microkernel are grouped according to three distinct observation levels:

1. The highest level (*application level*) concerns the observation of the symptoms of a propagated error:
   - *Application failure* corresponds to erroneous behavior or the production of erroneous data. Diagnosing such situations is carried out by an a posteriori analysis of the collected traces for each workload process against the *Oracle.*
   - *Application hang* is diagnosed when the workload has stopped running without having produced an erroneous output. The criterion for this diagnosis can be customized (e.g., between one workload process up to all processes).
2. The second level of observation (*interface level*) corresponds to the notification of a detected error. When issued from the detection mechanisms of the microkernel, the notification typically takes three forms. Either an *Error status* is notified by the workload process that issued the kernel call, an *Exception* is raised by the system, or a *Wrapper* returns a specific extended error status.
3. The lower level of observation (*kernel level*) concerns microkernel crashes. Such failures include:
   - *Kernel hang*: This diagnosis is established by testing the microkernel activity at the end of the experiment (e.g., nonactivation of a "witness" process and no reply to a "ping" command). More complex tests can be performed according to user needs.
   - *Kernel debugger*: The control is passed to the microkernel's debugger mode. This means that the microkernel has detected a critical error and deliberately put the system into a safe state by freezing the microkernel. This mode can be used by an operator to elaborate a diagnosis and to issue a simple corrective action.

Finally, MAFALDA explicitly addresses the cases when no failure or error report is observed for a fault injection experiment. Such a form of "no reply" was identified to mark a singular feature of SWIFI with respect to physical fault injection (e.g., pin-level[4]); this is mainly due to the difficulty of ensuring that the injected faults are really activated and was exhibited in many other related SWIFI experiments [17]. For instance, in FERRARI, nonactivated faults (labeled "no error") were a priori excluded from the statistics. In FINE, a similar issue has been observed: 68 percent of the faults injected have not been activated (labeled "very long latency") [26].

Nevertheless, several explanations can be given for such a high percentage of "no replies" for SWIFI experiments:

- The activation of the component under test does not lead to an internal control flow accessing the injected location;
- The latency is very long and the resulting error cannot be observed during the duration of the experiment;

4. In the case of pin-level fault injection, the activation rate commonly quoted was higher than 90 percent [14].

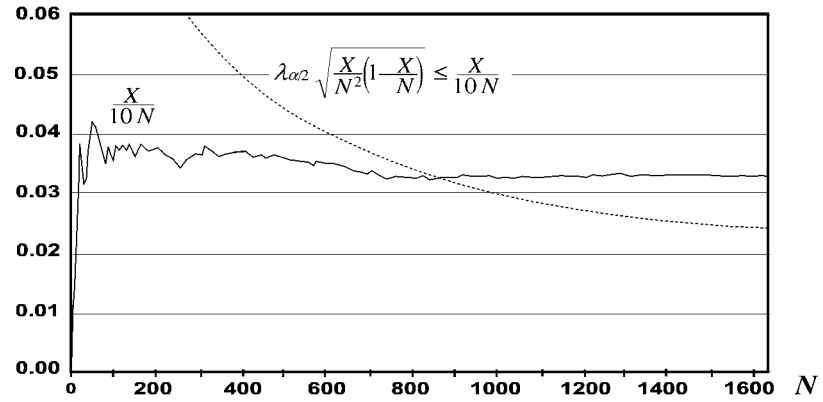

Fig. 5. Example of application of the stopping criteria for a campaign (*N* experiments).

- The injection applies to unused fields in data structures or in instruction format.

Because of the fault injection technique used in MAFALDA (processor debug registers and handlers), we can be sure whether the fault was activated or not. This allows the first case to be readily identified at runtime. Still, it is difficult to eliminate or make the distinction between the two latter, which is a classical concern in fault injection experiments. Accordingly, these two cases are aggregated into a single category, called "No observation" (fault activated, but no observed effect).

In summary, the results provided by MAFALDA are a synthesis of the events observed during the fault injection campaign, including:

1. error distribution,
2. error propagation,
3. notification distribution (exception, error status), and
4. the corresponding latencies (e.g., for the exceptions raised).

Clearly, the set of events synthesized can be microkernel-dependent and also relies heavily on the workload processes being used.

3.5 Experiment Selection and Campaign Duration

Although this is consistent with all related studies, the preliminary experiments carried out with MAFALDA revealed a high rate of nonactivated faults, especially in the case of microkernel injection.[5] It is worth noting that both 1) the principle of random selection of bit flip locations that governs the definition of the fault injection campaign conducted with MAFALDA and 2) the consideration of focused workload processes (that only partially activates the microkernel) have a direct impact on this high rate. Accordingly, in practice, one would like to see such an outcome be minimized as such an experiment can be considered a "nonsignificant test" in the assessment context we are considering here.

5. As an example, for the synchronization (SYN) component, from the 16,000 randomly selected locations in the code segment, 66 percent were not accessed.

Toward this ends, an a priori log of the control flow activated by the workload processes is performed.[6] This log is then used to confirm whether or not the fault injection experiment related to the memory word selected by random sampling in the kernel address space specified actually needs to be carried out. If the selected address is not part of this log, then the corresponding experiment can simply be skipped (as the outcome is already known).

Still, the substantial gain in the duration of the campaign resulting from the application of this procedure has a counterpart with respect to the accuracy of the experimental results obtained. Accordingly, some specific estimation needs to be carried out to specify the number of experiments to be performed so as to guarantee that the campaign is statistically significant.

In practice, it is usually assumed this will be achieved when the value of the half-confidence interval computed is lower than (or equal to) 10 percent of the empirical mean value. The application of this principle to the fault injection experiments conducted leads to the following formula that relates the number of experiments to be carried out (N), the current number of significant events (X) observed so far—the number of activated faults in our case—and the percentile of the underlying approximation to the binomial distribution ($\lambda_{\alpha/2}$):

$$\lambda_{\alpha/2}\sqrt{\frac{X}{N^2}\left(1-\frac{X}{N}\right)} \le \frac{1}{10}\frac{X}{N}.$$

In the application of this formula, we considered a confidence coefficient $\alpha = 95\%$. Fig. 5 shows a typical graph.

The practical interpretation is straightforward: The value on the horizontal axis when both curves cross indicates the minimum number of experiments that need to be carried out for the interpretation of the results of the campaign to be statistically significant. More details on the whole procedure can be found in [35].

6. Such an analysis is facilitated when injecting into code segments. Indeed, the segment implementation table (address of component primitives) is available at generation time and MAFALDA is able to identify the control flow within the components (instruction tracing). However, it is more difficult to determine a priori the access to data locations because it is very dependent on the control flow activated within the microkernel by the workload process.

TABLE 1
Configuration of the Chorus Instance

Component	Functionality	Exhibited interface	
CORE	Handling of actors and threads (creation, destruction, activation suspension)	Actors Threads	4 primitives 10 primitives
SYN	Standard handling of semaphores (initialization, acquisition, release)	Semaphores	3 primitives
MEM	Handling of the virtual protected address space, Region sharing (copy or mapping) Supervisor and user execution modes	Regions	5 primitives
COM	Synchronous or blocking communication (IPC) Remote procedure calls (RPC) Management of messages, ports and groups of ports	Ports Groups Service modes	10 primitives 3 primitives 9 primitives
SCH	Priority-based FIFO preemptive scheduling policy	One single primitive for dynamic modification of priority	

4 Failure Mode Assessment: A Case Study

The fault injection experiments reported in this section have been performed using an early version of the Chorus ClassiX r3 microkernel as the target system. It is worth noting that the main objective of this section is not to assess this microkernel but rather to illustrate the types of results that can be obtained by using MAFALDA on a candidate microkernel. Furthermore, it is important to stress that the results presented cannot be generalized; in particular, they are very dependent on the workload processes being used and many other factors (see Section 7 for a related discussion). Nevertheless, they explicitly show what types of objective data and insights the system integrator can obtain from using MAFALDA.

4.1 The Chorus ClassiX r3 Microkernel

The Chorus ClassiX microkernel [8] was an open version of previous versions of this microkernel [36]. It is composed of various internal components following the categories given hereafter, some providing primitives at the microkernel API:

- *Core (CORE)*: Set of primitives for the management of threads and actors (Chorus multithreaded processes) and hardware related basic functions (e.g., interrupts, timers, traps, MMU, etc.), some of which do not belong to the microkernel API.
- *Synchronization (SYN)*: Set of primitives for the management and the use of semaphores, mutexes, event flags, etc.
- *Memory Management (MEM)*: Set of primitives for the management and the use of memory segments including functions for flat memory, protected address spaces management policies, address space sharing, etc.
- *Scheduling (SCH)*: Various off-the-shelf schedulers are available, including priority-based FIFO preemptive scheduling or with fixed quanta, Unix Time Sharing, etc. This module does not directly provide visible primitives, although it handles running threads and actors and is thus indirectly activated by the creation of threads and actors.
- *Communication (COM)*: Set of primitives for local and remote communication.

A customized microkernel can be generated from the available off-the-shelf instances of the above internal components. Also, some of them can be user-defined (a specific scheduler, for instance) and can be used during the generation process, thus providing the users with various options to specialize the microkernel for a given application context. As the Chorus is "trap-based" (see Section 2.1), the microkernel and the application processes have distinct address spaces; the microkernel is running in supervisor mode, while application processes are (normally) running in user mode.

Table 1 summarizes the main features (functional classes and exhibited primitives) characterizing the configuration of the targeted Chorus instance. Among the functional components shown in Table 1, only CORE, SYN, MEM, and COM feature a real API. Such components make it readily possible to define workload processes dedicated to the activation of their respective primitives, thus complying with the requirements for functional separation of the (modular) workload expressed in Section 3.1. This is not the case, however, for SCH. Accordingly, although the SCH component was also submitted to fault injection experiments, it was not considered in the study of error propagation channels.

It is worth noting that a large part of the instance is composed of a common substrate (the CORE) implementing basic primitives used by these four components. The CORE component primitives are essentially activated to initialize the threads and actors used in the workload processes and were not target of the injection in the campaigns carried out. The full size of the microkernel instance was about 1 Mbyte.

Table 2 summarizes the respective size for the code and data segments of the four target microkernel functional components, independently from the basic CORE primitives used by each component.

4.2 Workload Processes

The SYN workload is made up of a set of threads that are competing for a shared resource. The acquisition of this resource is fully compliant with a strict mutual exclusion

TABLE 2
Code and Data Sizes of the Target Components

Component → ↓ Segment	SYN	MEM	COM	SCH
Code (kbytes)	2.0	19.2	47.4	2.7
Data (kbytes)	0.046	4.9	17.5	0.018

scheme controlled by a semaphore object. Based on the FIFO preemptive scheduling policy, the SYN workload ensures that the order of combination of the competing processes is deterministic, which is essential for the definition of an *Oracle* (cf. Section 3.4).

In order to stress the primitives supported by the MEM component, the dedicated workload features concurrent actors that make intensive use of matrix transposition and dynamic memory allocation. It also includes a check of data integrity at the matrix line level. This workload also induces deterministic behavior.

The COM workload implements a fairly complex communication protocol. In substance, it features exchanges between two sets of actors, each featuring signed IPC (Inter-Process Communication). This self-checking workload imposes a deterministic scenario to the actors involved.

The SCH workload is devoted to trigger activity for the various queues in the microkernel (e.g., ready-to-run queue, waiting queue, stopped queue, etc.) in order to stress the scheduling mechanisms. To do so, threads are constantly created, executed, stopped, resumed, delayed, awakened, changed of priority, etc. Incidentally, the SCH workload uses some features provided by the CORE component. The sequence of transitions to be performed is determined beforehand, i.e., the scenario developed is fully deterministic.

Thanks to the determinism of these workloads, a detailed analysis of the impact of the fault injection at the application level could be carried out. For example, corrupting the execution of synchronization primitives (either using microkernel or parameter fault injection) may have an impact on threads scheduling. Such occurrences have been reported in the experiments conducted. Errors like *thread hang* or *deadlock* correspond to *application hang* in the figures, errors like *illegal access to shared resources* and *wrong threads sequencing* leading to incorrect results correspond to the *application failure* case.

In the sequel to this section, we report some of the most relevant results obtained. In particular, for the sake of conciseness, we concentrate on the fault injection campaigns carried out either on the code segment of the *memory* space (Section 4.3) or on the *parameters* of the calls to the targeted microkernel functional components (Section 4.4). More detailed results, especially those concerning campaigns focusing injection on the data segment of the memory of the microkernel, can be found in [1]. Here, we only provide some selected results from these experiments.

4.3 Memory Fault Injection

It is worth noting that, in the case of the injection into the code segment, it was possible to eliminate a priori nonsignificant experiments from the randomly selected set of addresses by means of a preanalysis[7] of the execution flow activated within the microkernel by the workload process (see Section 3.5). Being able to know a priori which experiments are significant (and thus to execute only these) has a strong impact on the duration of a campaign.[8] This is essential for speeding up the campaign.

4.3.1 Distribution of Errors

We present here the results obtained by injecting into the code segment of various microkernel components of the target Chorus instance.

Fig. 6 shows the distributions observed for each component for about 3,000 activated faults. This means that the flipped code word was actually executed by the control flow. As already quoted (see footnote 5), the 66 percent "activation ratio" observed for the SYN component in this case means that the 3,010 experiments actually carried out are part of a larger set of about 16,000 randomly selected target words. It is also worth pointing out that the activation ratios observed for each target component revealed a strong relationship with the respective sizes of the target memory segments (see Table 2).

Fig. 6a depicts the results obtained for the SYN component. Among these errors, 9 percent have led the application to fail, i.e., the results obtained were different from the reference results and no detection signal was previously raised. At the other end, 28.5 percent of the activated faults fall into the *No observation* category. Similar experiments were performed on the MEM and COM microkernel components (see Fig. 6b and Fig. 6c). Slightly different results have been observed that are more satisfactory from the dependability and fault tolerance viewpoints: In particular, more exceptions have been raised and less application failures were reported.

4.3.2 Error Propagation

MAFALDA also supports the characterization of error propagation channels within the microkernel. Indeed, some errors propagate from the injected component to some companion component. For instance, when SYN is the injected component, the propagation may affect the

7. It is worth noting that such a procedure would be more difficult to apply in the case when fault injection focuses on the data segments. Accordingly, the number of significant experiments that can be carried out in practice on a specific component is usually substantially lower for the data segment than for the code segment.

8. Each experiment lasted six minutes on average. This delay includes: intialization, fault injection, data collection, and reboot of the target system.

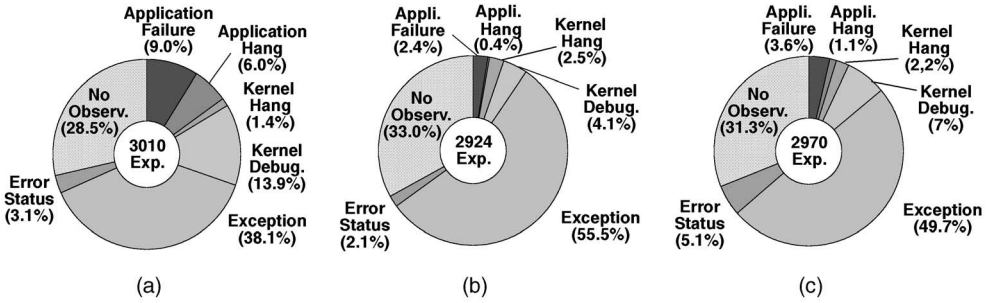

Fig. 6. Distribution of errors—code segment. (a) SYN component. (b) MEM component. (c) COM component.

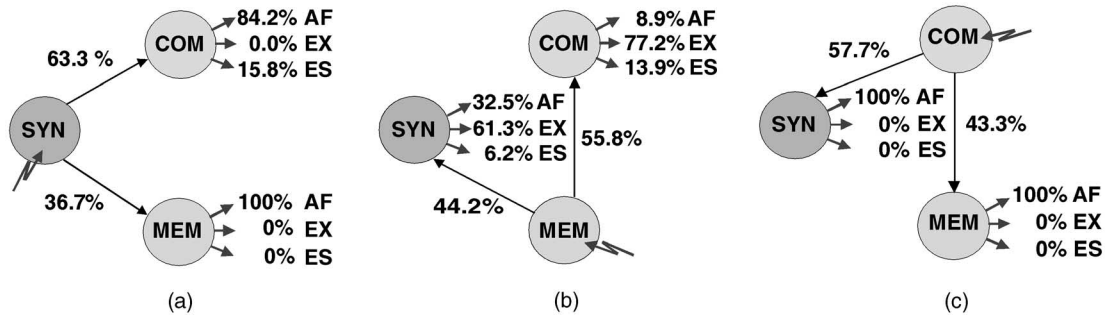

Fig. 7. Evaluation of error propagation channels—code segment (# propagated errors) (AF: Application Failure; EX: Exceptions; ES: Error Status). (a) SYN component (30). (b) MEM component (181). (c) COM module (26).

MEM and the COM components. Similarly, when MEM and COM are the injected components, some errors propagate to the two other companion components. It is important to note that the events that are prone to induce a priori propagated errors are essentially: application failures, exceptions, and error status notifications (see Fig. 6). This corresponds to: 1,511 events (~ 50.2%) for SYN, 1,725 ~ 59% for MEM, and 1,734 ~ 58.4% for COM.

Fig. 7 provides the number of propagated errors observed and the propagation distributions.

The results show that a fault injected into a given microkernel component may provoke an error that propagates to a different component as revealed by 1) the occurrence of error signaling events when this other workload process is being executed (e.g., error status notification, exception raised) or 2) the diagnosis of an application failure with respect to the workload related to this other component. In other words, these events have been observed first for a workload process different from the workload process focusing on the injected component. For instance, among the faults that propagated when MEM was the target component, 44.2 percent affected SYN and 55.8 percent affected COM (Fig. 7b).

It is also worth mentioning that a more significant ratio of errors is propagated when the injection is performed into the MEM component (actual propagation ratio = 181/1,725 ~ 10%) when compared to the two other ones (< 2%). This singular behavior is mainly due to the frequent solicitation of the low-level dynamic memory allocation services. Moreover, besides more errors being propagated, it is worth noting that their consequences are better caught, in particular, by the exception handling mechanisms (SYN: 61.3 percent, COM: 77.2 percent) and, accordingly, lead to a lower proportion of application failures. All these insights are of main interest, both for the system developers and the microkernel providers.

4.3.3 Exception Coverage and Latency

In all of the experiments, most of the exceptions raised (actually, 83 percent) led to a segment violation (*segFault* exception) when the fault was injected in the code segment of the microkernel component. The other exceptions are invalid instruction code (*InvOpCode*: 10 percent) and coprocessor error (*copError*: 7 percent). The same distribution was obtained with all three components since exception mechanisms are common to all of them.

Timing measurements carried out by MAFALDA allow for error latencies to be obtained. Both *Exceptions* and *Kernel Debugger* events are considered. In fact, both events presented the same latency distribution. These noted similarities are partly explained by the fact that the *Kernel Debugger* event is raised by a microkernel-defined exception. Also, the distributions were identical for all three components tested. As an example, Fig. 8 illustrates the *Exception* latency distribution observed for the SYN component.

The distribution shows that a large proportion of the exceptions—about 30 percent (340/1,135)— exhibited a null latency, i.e., they occurred during the instruction being executed when the fault was injected. If not raised immediately, 80 percent of the remaining exceptions are raised with a latency of less than 4 μs. Finally, we can also observe from these experimental results that few (142, i.e., 12 percent) are raised with a latency beyond 10 μs. Among

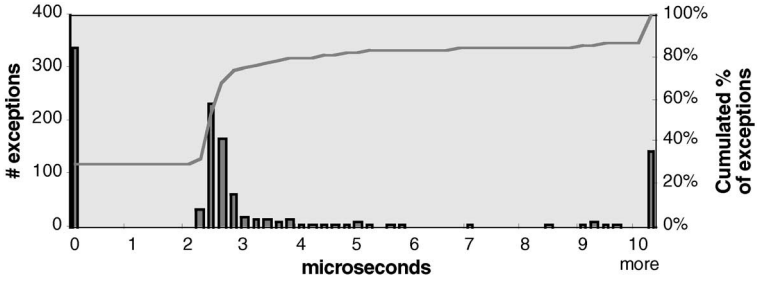

Fig. 8. Exception latency—code segment (case of SYN component).

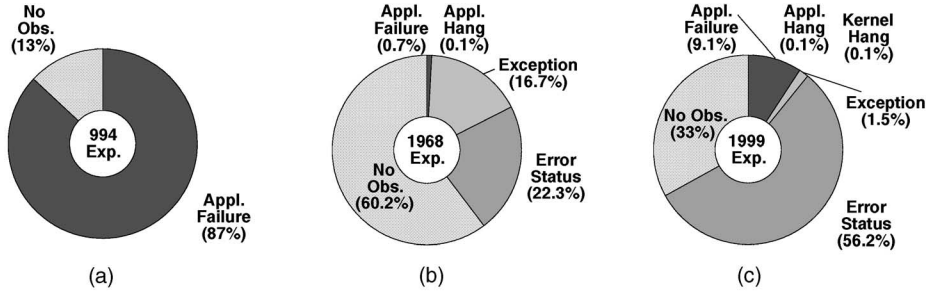

Fig. 9. Distribution of errors—parameter fault injection. (a) SYN component. (b) MEM component. (c) COM component.

them, 42 were raised before 30 μs, while the remaining 100 occurred beyond 4.5 ms. These results depend very much on the control flow activated by the considered workload processes. Nevertheless, obtaining temporal information on the faulty behavior (such as exception latency) is of high interest for real-time systems/applications.

One of the main differences exhibited by the series of experiments carried out on the data segment concerned the latency distribution: No immediate exceptions are observed in this case. This is not surprising since a wrong data value seldom has an immediate effect on the processing being performed or on the control flow of the program being executed (see [1] for more details).

4.4 Parameter Fault Injection

Flipping a single bit in a parameter used by a kernel call should enable the control checks performed by the microkernel on parameters delivered during the call to be evaluated. It is worth pointing out that the notion of nonactivated fault has no meaning in this case: Indeed, the bit-flip on the target parameter occurs when the primitive is invoked.

4.4.1 Distribution of Errors

Fig. 9 shows the results obtained for the three microkernel components considered.

The detection of an error affecting a primitive parameter can be very difficult to achieve as it requires semantics checks which are not provided at all by a standard microkernel. For instance, for the SYN component, although only a small ratio of the experiments (13 percent) had no observable effect, they also revealed a very high application failure ratio (87 percent). These abnormal behaviors include: illegal access to a critical section, resources no longer accessed, thread hang, etc. The reason for this singular behavior can be explained by the fact that any parameter value is accepted by the synchronization module.[9] For instance, a wrong semaphore queue identifier simply leads to the creation of a new semaphore queue data structure in the microkernel. Also, when the number of tokens requested by a semaphore is corrupted, the value of the semaphore is just changed without any additional control.

We would like to stress here that these results were obtained when using semaphores in user mode with Chorus ClassiX r3 and can be very different with another candidate microkernel.[10] We also believe that this behavior is very implementation dependent. In particular, the use of semaphores in supervisor mode leads to very distinct results (see Fig. 10).

Actually, the control flow of the primitives in supervisor mode is very different. Input parameters are not processed in the same way by the primitives. For instance, while in the user mode, a corrupted identifier of a semaphore queue (unknown queue id) leads to the creation of a new queue; when running in supervisor mode, this same event is usually detected (either *Exception*, *Kernel Debugger*, or *Kernel Hang*). Nevertheless, the proportion of undetected failures is still very high, thus strengthening the need for additional checks.

4.4.2 Error Status, Error Propagation, and Exception Coverage and Latency

The most important outcome provided by the microkernel regarding parameter fault injection is an error status returned to the user application, such as *K_INVAL*

9. Roughly speaking, the parameter of several primitives of the SYN component is a simple data structure composed of three fields (`queue_key`, `blocked_threads`, `semaphore_count`) plus some unused bytes.

10. The experiments conducted on an instance of LynxOS featuring a POSIX API showed totally different results for a similar set of experiments (see Section 7.1).

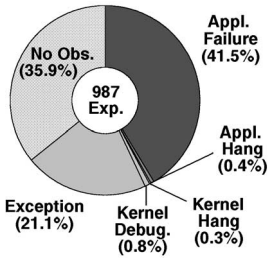

Fig. 10. Distribution of errors—SYN primitives in supervisor mode.

(inconsistent process id) or *K_UNKNOWN* (process not reachable).

Also, very few errors propagated from one component to another; most propagated errors have been observed when injecting on primitives related to the MEM component, which is central to the microkernel as it is often used by internal companion components by means of internal functional calls (e.g., dynamic memory allocation).

A single type of exception was raised during these experiments: *segFault* (illegal memory access). For what concerns exception latency, as the kernel calls are not processed instantaneously, the distributions observed featured no zero-delay occurrences (similar to the ones observed for data segment injection). In particular, the Exception latency for the MEM component showed a singular (multimodal) distribution (Fig. 11). Three populations can be observed centered at 3, 5, and 7 μs, encompassing 80 percent of all exceptions. They are due to the way the microkernel handles the parameters of the different MEM primitives. For example, exceptions centered at 5 μs are triggered when a corrupted region structure is eventually used to allocate dynamic memory, while exceptions at 7 μs appear because of a corrupted actor descriptor is handled by the kernel. Since the exceptions are only activated when the injected parameter is finally handled, some of them have a much higher latency (up to some milliseconds in our experiments).

4.5 Discussion

In this section, we have summarized the main results obtained when submitting a Chorus instance to fault injection experiments using the facilities offered by MAFALDA. In particular, the focused analysis at the component level allows for very useful insights to be obtained concerning: 1) the specific failure modes that characterize a functional component and 2) the error propagation channels between these components. This capability is very much adapted to analyzing the current generation of microkernels that feature a componentized architecture.

Of course, the numerical values obtained are very much dependent on the workload being used, on the instance being analyzed, and also on the target microkernel. However, MAFALDA provides a framework that is very much suited to supporting the derivation of objective data on the faulty behavior of the target kernel. As such, these insights are useful for both the system developers to help them assess a candidate microkernel and the microkernel suppliers to complement the standard benchmarking and conformance testing activities. It is important to stress that the analysis of the detailed information being logged provides useful insights for 1) understanding the failure cases and the propagation channels, 2) identifying potential implementation flaws and lack of internal checks, and 3) suggesting some specific architectural improvements.

One interesting observation related to the results we obtained is that fault injection into data segments was found much less stressful than the other two forms of fault injection. Although it was clearly expected that injecting into code segment would be more error prone than injecting data segments, this was much less obvious for what concerns parameter fault injection. These results were confirmed by the analysis of the related error logs which revealed a larger proportion of errors affecting the control flow. From a practical point of view, such an observation provides another rationale for choosing rather to focus the tests on the code segments in the case of fault injection into the memory of the microkernel.

The high rates of application failures or even crashes that were reported in this section for the Chorus instance clearly reveal the sensitivity of such a type of the COTS components[11] to faults, be they internal or external.

These results fully justify the need for improving the error detection and confinement mechanisms offered by a standard microkernel in order to help the developers make their decision whether to actually integrate such COTS components into systems featuring strict dependability requirements. This is the subject of the next section.

5 Error Detection and Confinement: A Case Study

Based on the results obtained on the Chorus instance submitted to fault injection experiments, the SYN component was definitely a candidate for wrapping. Moreover, the detailed analysis of the application failures identified for this component revealed several interactions with the SCH component. This significantly motivated the consideration of these two components in the development of error detection and confinement wrappers. In this section, we first present the principle of defining predicates on the expected behavior of basic primitives of the microkernel functional components to derive error detection wrappers. Two examples of the application of this approach are given that concerns two components (namely, SYN and SCH) of the same Chorus instance. Experimental results are then presented that illustrate the efficiency of these wrappers.

5.1 Definition of Predicates

Predicates are defined to assert the correct behavior of a single functional component. This consists of, whenever possible, defining, in a formal way, the correct behavior of the functional component, invoked by some primitives of the executive software. It is worth noting that, although independent from a semantics viewpoint, the components are linked together and interact. This was actually illustrated by the error propagation results

11. This was corroborated by the results obtained on an instance of the LynxOS microkernel (see Section 7.1).

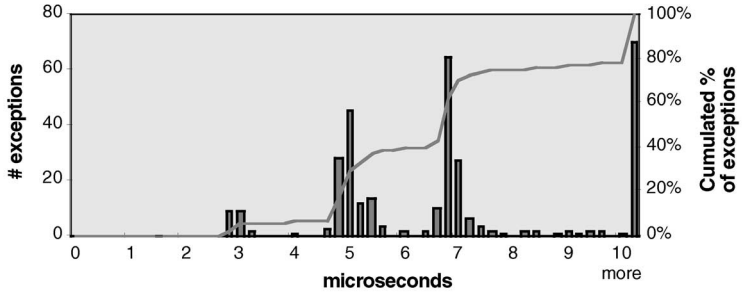

Fig. 11. Exception latency—parameter fault injection (MEM component).

reported in Section 4. For instance, a thread suspended when requesting a semaphore handled by the SYN component leads to the activation of a different thread by the SCH component. Accordingly, the activation of one primitive may thus have an impact on the behavior of a companion component. Thus, for the above reason, and although predicates can be defined independently, they must be verified altogether at runtime.

Predicates are complementary to the error detection mechanisms already provided by COTS microkernels. A predicate characterizes the consistency of the results produced (output parameters) and of the modified internal data according to the given input parameters and the internal state before the invocation. The violation of a predicate is thus interpreted as a faulty behavior of the corresponding component.

Three major criteria have to be accounted for in the definition of predicates: 1) ability to define (model) the semantics of the functional class considered, 2) appropriate abstract level of their definition to master the complexity, and 3) observability—to a limited extent—of the internal state of the executive. Some microkernel mechanisms can be analyzed on the basis of their specifications, i.e., a logical expression or a model of their correct behavior. In most cases, access to internal data is needed. These two aspects are illustrated in the next subsections.

5.2 Synchronization

We illustrate here, by considering basic synchronization mechanisms (semaphores), how predicates can be derived from the specifications of the class of objects handled by the executive software. Using Dijsktra semaphore specification (e.g., see [37]), a semaphore s corresponds to an integer value *s.val* and a process queue denoted *s.queue*. Three main primitives are provided to the applications: Init $I(s)$, Get $P(s)$, and Release $V(s)$.

The semaphore s is initialized by a first value denoted *init_value* (i.e., the initial number of available tokens). Let $\#P(s)$ (resp. $\#V(s)$) be the total number of calls to the $P(s)$ and $V(s)$ primitives. The invariant property that can be derived from the definition of $P(s)$ and $V(s)$ is:

$$[s.val = ini\_value - \#P(s) + \#V(\mathrm{s})].$$

This property is obviously true at initialization time (since *s.val* = *init_value*) and must be kept true for any number of invocations to $P(s)$ and $V(s)$. Then, denoting #*Suspended*(s) the number of threads in *s.queue*:

$$[\#Suspended(s) = max(0, -s.val)].$$

The interpretation of this relation is as follows: If *s.val* is negative, then there are -*s.val* threads suspended in *s.queue*. When $s.val \geq 0$, then *s.queue* is empty. Accordingly, a predicate can be defined as follows:

$$\begin{aligned}&[s.val = init\_value - \#P(s) + \#V(\mathrm{s})]\wedge\\ &[\#Suspended(s) = max(0, -s.val)].\end{aligned} \quad (1)$$

This predicate can be assessed after the execution of the $P(s)$ and $V(s)$ primitives, provided the number of suspended threads is accessible. For example, in the case of the Chorus microkernel, some semaphores data structures are located in the application address space (user space) even though the *s.queue* is located in the microkernel address space. This means that the size of the *s.queue*—denoted #*Suspended*(s), must be made visible to execute the predicate of (1).

5.3 Scheduling

The problem of modeling a scheduling policy or of defining formal specifications of a scheduling policy to support the design process of real-time microkernels and systems is an important issue [38], [39], [40], [41]. Nevertheless, this focus is out of the scope of this paper: Our main objective is limited to the detection of any violation of the scheduling of the tasks that may be caused by an error. Accordingly, the models we are considering here are voluntarily simple. They are mainly meant to illustrate the principles underlying the definition and development of online error detection wrappers.

5.3.1 Behavioral Model

Task scheduling can be expressed informally in the following way: The scheduler is responsible for electing the first task (thread) among existing ready-to-run tasks, according to a given policy. Depending on the policy, the first ready-to-run task is the task with the highest frequency (e.g., rate monotonic scheduling) or the task with the earliest deadline (e.g., the EDF policy), etc. Such rules can be easily described as predicates, provided that some internal data of the scheduler component are accessible. The example considered in this subsection is based on the priority-based FIFO preemptive scheduling policy (e.g., see [37]). The behavior of the corresponding scheduler can be modeled as illustrated by Fig. 12.

Any task newly created can be put either in the **Stopped** state or in the **Ready-to-run** state. A stopped task becomes ready to run when the **Start** operation is executed by

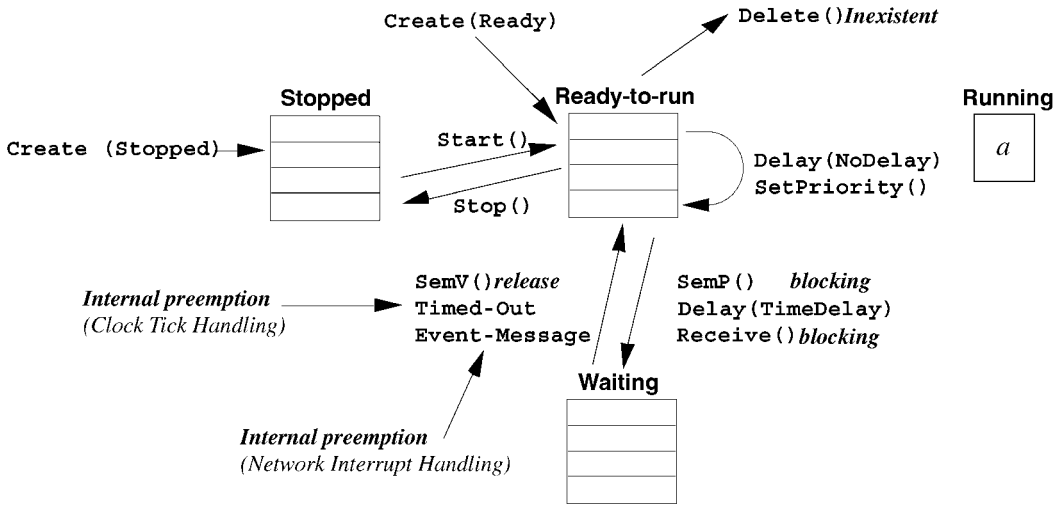

Fig. 12. A simple scheduler model.

another task. A task returns to the **Stopped** state when either it (or a different task) executes the **Stop** operation. The task selected to become the running task (denoted *a* in Fig. 12) is the task in the **Ready-to-run** state with the highest priority. All tasks with the same priority level are scheduled on a FIFO basis. This running task can change the priority of a ready-to-run task or even delay it. This delay operation can be used to release the CPU for another task at the same priority level (**NoDelay** in this case). Alternatively, a task can be put in the **Waiting** state. Such a transition may be due to a blocking operation on a semaphore object, a voluntary suspension for a given delay, or a blocking receive. Conversely, a task resumes to the **Ready-to-run** state when either: 1) A release operation is performed on a semaphore object, 2) the delay has expired, or 3) an event related to the reception of a message has been triggered. The last two cases correspond to an asynchronous event not related to the activation of a primitive, but due to an external input.

All the transitions in this model are related either to a primitive of the scheduler itself or a primitive of a companion component within the microkernel. From an object-oriented viewpoint, several objects are responsible for the evolution of the task queues handled by the microkernel. The **scheduler** object provides the following external primitives to handle threads: **Create**, **Start**, **Stop**, **Delay**, **PrioritySet**, and **Delete**. Some internal functions are also available to control the thread behavior that may impact task scheduling: **Wait**, **Signal**, and **PrioritySet**. A **semaphore** object is controlled by two primitives, **SemP** and **SemV**. The **MessageQueue** object has two primitives, **Send** and **Receive**. A **TimeOut** object has a **TimeoutSet** internal function to initialize a timer and the **OneClockTick** activated by the real-time clock and responsible for the management of the timers.

5.3.2 Predicates

The considered predicates perform consistency checks on the scheduler behavior. They are based on the same model since, in most cases, a transition corresponds to a task t leaving a given queue and entering another one.

Let us denote *PSched* (•) as the predicate defining the selection process of the current running task among all the candidate tasks. In the case of the FIFO preemptive policy, such a predicate expresses the fact that the running task *a* is always the first one of the highest priority level in the *Ready-to-run* queue denoted R:

$$\forall(t \in R) : \text{prior}(t) \leq \text{prior}(a). \tag{2}$$

Clearly, (2) must always hold right after a transition occurs. For the sake of conciseness, in the sequel, we will simply denote it *PSched*(*R*, *a*).

Regarding the microkernel behavior, the predicates verify that the activation of a primitive or a given event leads to the appropriate transition in the scheduling state diagram. As an example, the postcondition after a **SemV(*s*)** that releases a task t can be summarized as follows:

$$t \notin W(s) \wedge t \in R \wedge PSched(R, a), \tag{3}$$

where *W(s)* is the queue of tasks waiting on semaphore s and t denotes the first task blocked on s. Expression (3) means that task t exits the **Waiting** queue for the semaphore s $[t \notin W(\text{s})]$ and becomes ready $[t \in R]$ and, as a result of this new task entering the **Ready** queue, the predicate *PSched*(*R*, *a*) must hold.

Similarly, the expiration of a delay involves the verification of the following predicate:

$$t \notin W_d \wedge t \in R \wedge PSched(R, a), \tag{4}$$

where W_d is the queue of delayed tasks and t is the task whose delay has just expired.

Similar predicates are defined for all primitives and events identified in the scheduling state diagram of Fig. 12. It is also worth noting that the implementation of these predicates assumes an access to the descriptor of the running task and to several queues used in the microkernel. Table 3 summarizes the various predicates defined for the considered scheduling policy. More details can be found in [35].

However, it is clear that checks based on such predicates cannot ensure that the behavior of the tasks handled by the

TABLE 3
Primitives and Related Predicates

Primitive	Predicate
Create(t, Stopped)	$t \in W_S$
Create(t, Ready)	$t \in R \wedge PSched(R, a)$
Start(t)	$t \notin W_S \wedge t \in R \wedge PSched(R, a)$
Stop(t)	$t \in W_S \wedge t \notin R \wedge PSched(R, a)$
Delay(t, d)	$t \in W_d \wedge t \notin R \wedge PSched(R, a)$
OneClockTick	$t \notin W_d \wedge t \in R \wedge PSched(R, a)$
Delay(NoDelay)	$PSched(R, a)$
SetPriority(t, pr)	$PSched(R, a)$
SemP(s)	$t \in W(s) \wedge t \notin R \wedge PSched(R, a)$
SemV(s)	$t \in W(s) \wedge t \notin R \wedge PSched(R, a)$
MessageEvent	$t \notin W(mq) \wedge t \in R \wedge PSched(R, a)$
Receive(mq)	$t \in W(mq) \wedge t \notin R \wedge PSched(R, a)$

- $PSched(R, a)$: $\forall(t \in R) : prior(t) \leq prior(a)$
  a: running task, R: Ready-to-run queue, pr: priority level;
- $W(s)$: queue of tasks waiting on semaphore s;
- $W(mq)$: queue of mq objects (Message Queue type);
- W_d: queue of suspended tasks waiting on a delay;
- W_S: queue of non activated tasks (Stopped state).

microkernel is correct. Several cases can be identified. For instance, a given task may block forever on a semaphore or wait for a message that never arrives. Furthermore, in this model, the verification of the transition from the **Waiting** state to the **Ready-to-run** state is possible only when a message is received by a task or when a semaphore is released. Another example is a task in the **Stopped** state that is never reactivated. Such situations relate to either a fault in a companion component (e.g., message loss) or a wrong design of the application. Handling these situations would require modeling and simulating the whole application to be able to reveal the corresponding design faults. This would involve verifying safety and liveliness properties that are definitely application-dependent.

5.4 Discussion

The definition of predicates depends on the ability to formally specify the behavior of objects managed by the microkernel. We showed how this was achievable for semaphores and scheduling policies. Clearly, a cost/efficiency trade-off exists between predicates corresponding to a very detailed modeling, but that would be very difficult to develop than simpler predicates corresponding to a higher modeling level, but that might not be as efficient.

However, for many other functional components, such a formal specification can seldom be obtained. This is the case in particular for the MEM and COM components. Two options are possible. One may want to define a model of the memory management policy or of the basic communication protocols. This activity may be very tedious and the resulting model questionable. The second option is more pragmatic. It is based on operational consistency checks such as acceptance or validity checks. For instance, an allocated memory segment must be entirely accessible (i.e., no bus error observed when reading or writing any of its bytes) as a result of the corresponding primitive invocation. Another example is given by the fact that a released memory block should be readily available for reuse by another task. Similarly, regarding IPC, any message sent should be received and correctly acknowledged to the sender (e.g., TCP/IP).

We have currently extended this work by providing formal specifications using a temporal logic describing the required properties of real-time kernels. These specifications are the basis for the implementation of a new class of wrappers [42].

5.5 Evaluation of Wrappers

We now present and discuss the results obtained when applying fault injection on the wrapped Chorus ClassiX instance. The main goal is to assess the improvement of the error detection coverage provided by the two wrapping mechanisms introduced in Sections 5.2 and 5.3. This comparison is based on results we previously presented in Section 4 when analyzing the faulty behavior of the microkernel without wrapping mechanisms. In the sequel, we present in turn the results obtained for the synchronization and task scheduling components.

5.5.1 Synchronization Component

For the sake of readability, Fig. 13a recalls the results obtained when subjecting the microkernel to faulty synchronization input parameters (which was the most severe case). Proceeding to the same fault injection campaign, but with the synchronization wrapper defined in Section 5.2, we now observe very distinct results: All propagated errors are detected before any application failure is observed (Fig. 13b).

Fig. 14 depicts the behavior of the microkernel when subjected to transient faults affecting the code segment of the synchronization module.

Fig. 14a recalls the results obtained for the standard component. The results obtained with the SYN wrapper are shown in Fig. 14b. The SYN wrapper significantly reduces the percentage of application failures (from 9.0 percent to 2.2 percent). Although important, this result shows that the

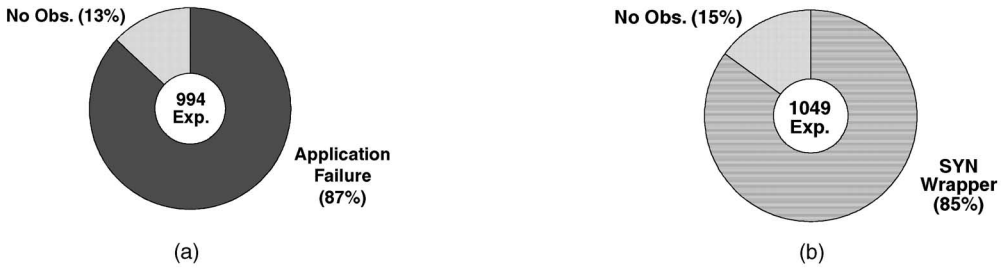

Fig. 13. Distribution of errors—parameter injection in the SYN component. (a) Standard component. (b) With SYN wrapper.

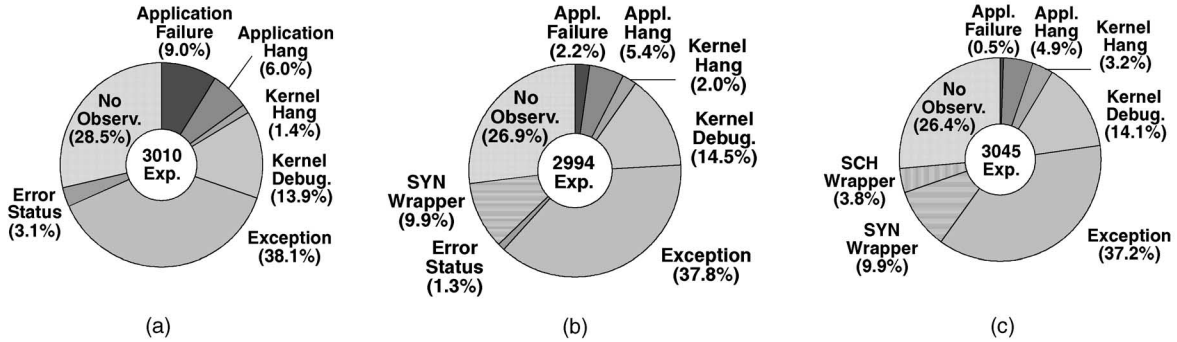

Fig. 14. Distribution of errors—code segment injection in the SYN component. (a) Standard component. (b) With SYN wrapper. (c) With SYN and SCH wrappers.

SYN wrapper is unable to prevent all the errors from propagating to the application level. A closer examination of these 66 (2.2 percent) application failures showed that most of them were related to scheduling problems, such as threads mixed up in an incorrect way. From (1), it is clear that the SYN wrapper is not designed to handle such a kind of faulty behavior. So, we decided to carry out the same fault injection campaign using now both the SYN and SCH wrappers. The corresponding results are shown in Fig. 14c: Only 0.5 percent of the injected errors remain undetected leading to an application failure.

As shown by Fig. 13b, the predicate implemented by the SYN wrapper is efficient enough to keep applications from misbehaving when faulty parameters are delivered to the microkernel. This is why no further experiments were carried out using both wrappers in this case. However, as exemplified by the injection experiments carried out in the code segment (Fig. 14b), synchronization requests may impact the scheduling: The release of a *mutex* leads queued threads to be activated by the scheduler. This provides a means for errors affecting the synchronization module to propagate to the scheduling module. The results obtained when combining the use of both SYN and SCH wrappers support this analysis.

Nevertheless, it is worth noting that the restriction to a smaller number (15, i.e., 0.5 percent) of application failures makes a detailed analysis of these residual cases readily feasible. Such an analysis provides useful insights according to several complementary viewpoints: 1) on the design and implementation of the microkernel (microkernel provider viewpoint), 2) on the design and the implementation of upper layers (system developer viewpoint), and 3) on further development of the wrappers (our viewpoint).

5.5.2 Task Scheduling Component

We now turn to the analysis of the scheduling component subjected to transient faults injected into the code segment (Fig. 15). The results concerning the standard component are shown in Fig. 15a. About 4.9 percent of the errors led to an application failure, 14.6 percent to an application hang, 2.5 percent provoked the kernel to hang, and about 52 percent of the remaining errors were successfully detected. It is important to point out that most errors are detected by mechanisms internal to the microkernel rather than processor-level mechanisms: Indeed, 43.4 percent activate the Kernel Debugger, while only 7.4 percent raise an Exception.

A similar fault injection campaign conducted with the scheduling wrapper gives the results of Fig. 15b. The wrapper thus reduces the percentage of application failures to 2.4 percent. In addition, application and kernel hangs are reduced to 3.9 percent and 1.1 percent, respectively. As an illustration of the role of the SCH wrapper, let us indicate the following examples of faulty situations it was able to successfully handle and notify:

- A thread newly created, awoken, or changing priority has not been correctly inserted (or not inserted at all) in the ready-to-run queue.
- The running thread is not preempted either 1) upon reception of a message by a waiting thread with higher priority or 2) upon expiration of a time-out that should wake-up a delayed thread with higher priority.
- A thread performing a blocking action on a mutex semaphore is not removed from the ready-to-run queue.
- The elected thread does not have the highest priority of all threads in the ready-to-run queue.

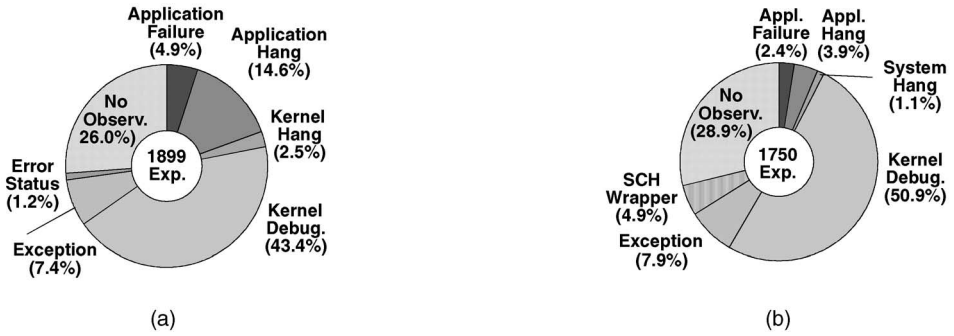

Fig. 15. Distribution of errors—code segment injection in the SCH component. (a) Standard component. (b) With SCH wrapper.

Further analysis of the 41 (2.4 percent) residual application failures clearly showed that the corresponding erroneous situations were definitely out of reach of the proposed SCH wrapper, i.e., they provoked errors significantly departing from a mere incorrect ordering of the threads. A detailed analysis of these error situations made it possible to categorize them into four main classes. Interestingly, one class gathers about half of them, while the others are almost evenly distributed among the three remaining classes. The distribution of the residual application failures within these four classes, along with a brief description of the corresponding error situation, are given below:

1. Twenty-one failures are provoked by wrong priority information returned by the microkernel, leading the requesting applications to no longer behave correctly.
2. Seven failures are related to asynchronous events (e.g., arrival of a message or the expiration of a time-out) which are not correctly handled by the microkernel.
3. Seven failures correspond to the hang of all the threads attached to the same application, without either any previous violation of the ordering of the threads or failure of the application.
4. Six failures result from the modification of the execution flow of the applications, without any violation of any ordering property.

These observations set new grounds for the development of improved wrappers. In particular, a scheduling wrapper tailored to deal with the real-time specification of the microkernel services should be able to avoid failures of class 3. Improving the trapping of asynchronous events should enable failures of class 2 to be avoided, while wrappers specially designed to control the application behavior are mandatory to avoid failures of classes 1 and 2.

6 Implementation Issues

One very important concern when considering the assessment and error detection methods described in the preceding sections is to make these readily applicable, in particular in an industrial context. Toward these ends, the development of a supporting tool (MAFALDA) and the associated implementation issues play a major role. This is especially true with respect to the nonintrusiveness of the fault injection, the performance and efficiency of the wrappers, as well as the portability of the overall approach to various microkernel candidates.

6.1 Memory Fault Injection

Fault injection into the microkernel address space carried out by MAFALDA takes advantage of the debugging hardware features existing in most of the modern microprocessors so as to inject faults by software and monitor the activation of the faults in the same manner as Xception [24]. Fig. 16 illustrates the fault injection technique used in MAFALDA.

As shown in Fig. 16a, MAFALDA provides an exception handler, activated by both temporal and spatial triggers, which is responsible for injecting either a permanent or a transient fault in the code and data segments of the microkernel. For example, Fig. 16b describes how MAFALDA carries out the injection of a transient fault into the code segment of the microkernel. When the temporal trigger expires and the target assembly instruction is reached by the processor program counter, the exception handler is activated. The handler signals the activation of the fault and corrupts the assembly instruction, applying a bit-flip mask to it. In order to simulate a transient fault, a trace mode is then set up and execution is allowed to resume. Now, the faulty instruction is executed. If no failure occurs, the handler takes over the CPU again, resets the trace mode, recovers the faulted bit, and resumes execution.

6.2 Parameter Fault Injection

Techniques concerning parameter fault injection are different whether library-based microkernels or trap-based microkernels are considered (see Section 2.1). Besides the tool has been designed to accommodate both types of microkernels, as both target microkernels (Chorus and LynxOS) are trap-based, we emphasize on this category here. *Trap-based* microkernels feature a single entry point for kernel calls to the microkernel. Fault injection is carried out by intercepting target kernel calls before they enter the microkernel, as shown in Fig. 17.

The simplest and more efficient way to transparently carry out this interception is by *trapping* the kernel call entry point. For that, MAFALDA uses the debugging hardware features of the microprocessor to program a hardware breakpoint so that an exception is automatically raised whenever the kernel call entry point of the microkernel is reached by the processor program counter. The handler for this exception is responsible for corrupting the input

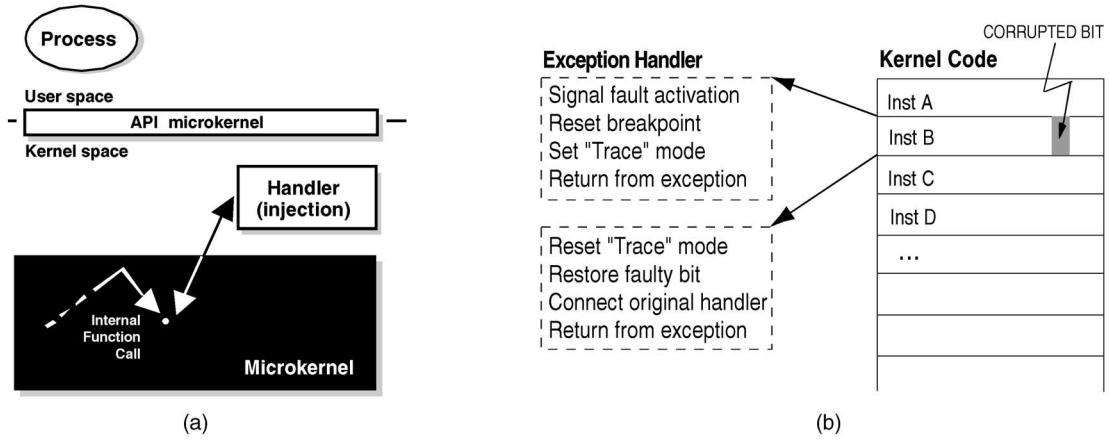

Fig. 16. Memory fault injection. (a) Overview. (b) Transient fault in code segment.

parameters. Once parameters have been injected, the handler lets the call proceed into the kernel. The advantages of this approach are twofold: 1) The source code of application processes is not needed and 2) user defined workloads with specific activation profiles (i.e., matching a particular application domain) can be readily used.

In the case of a *library-based* microkernel, a fault injection library is provided on top of the standard library. A detailed presentation can be found in [3].

6.3 Error Detection and Confinement Wrappers

The wrappers we propose for *trap-based* microkernels rely on a *metakernel protocol*. This protocol implements behavioral reflection on the target microkernel. Essentially, it provides interception of the kernel calls and enhanced observation of its internal behavior. This behavior can be analyzed and customized from outside the target component, i.e., at its *metalevel*. From the metalevel of the kernel (thus called the *metakernel*), some actions can be undertaken to modify the observed behavior. This allows for every kernel call and dedicated internal function call within the microkernel to be intercepted. Interception is achieved by programming the internal hardware debugger. Some internal data structures within the microkernel can be exhibited to the upper layers of the system. This allows for the internal state of the microkernel to be both observed and controlled. The observation and control facilities are provided by the reflection module (see Fig. 3) that implements a side-API (the so-called *metainterface* in a reflective context).

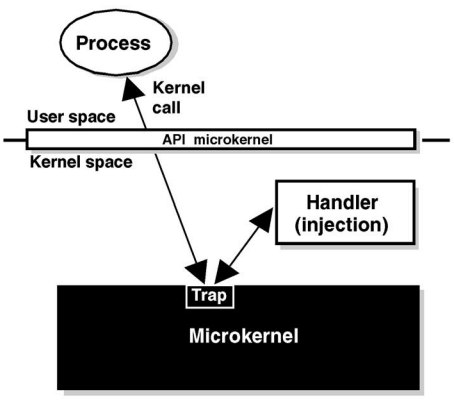

Fig. 17. Parameter fault injection (trap-based microkernel).

Fig. 18 illustrates the metakernel protocol of MAFALDA. Every kernel call or internal function call is trapped by a handler, which diverts the control to the metakernel. The latter carries out checks during the *preactions* and *postactions*, taking advantage of the metainterface to access the information needed. The actual execution of the primitive takes place between such pre and postactions, but it can be skipped or eventually replaced by another primitive. Finally, normal execution flow is resumed by returning to either the user application or the microkernel, depending on whether the call originated from the user layer or from a microkernel component.

Wrappers for *library-based* microkernels are provided as a second additional library that is inserted between the fault injection and the standard libraries (see [3] for details).

7 Overall Insights

This section summarizes the overall insights obtained from the experiments carried out with MAFALDA. In addition to the insights gained from the experiments on the Chorus instance (see Section 4), these also concern the fault injection experiments carried out on LynxOS (r3.0.1) [9] with the same hardware platform. Besides the improvements that have been made to the tool (portability, ease of use, off-line data analysis, etc.), carrying experiments on an alternative target microkernel was really interesting to further assess the benefits that can be obtained with the evaluation methodology supported by MAFALDA. We describe first some results of the experiments with LynxOS. The main lessons learned are reported in Section 7.2.

7.1 Targeting a Different Microkernel

The objective here is not to carry out a detailed comparison of the two target microkernels. Indeed, many parameters (e.g., target instance) and other aspects of the experimental context (e.g., workloads, "faultloads") impact on the results of the experiments; careful attention must be paid when considering using them for comparison. In addition, ethical

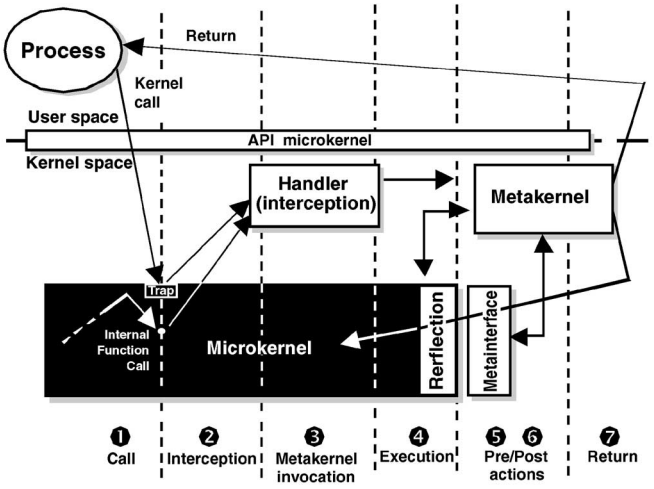

Fig. 18. Error detection and confinement wrappers (trap-based microkernel: metakernel protocol).

reasons prevent us from publishing a full account of such a form of comparison. Our aim is, rather, to focus on the main distinctive aspects that were exhibited and to discuss the various interpretations of the results.

The experiments targeting LynxOS have been carried out with a major company in the aircraft industry, this COTS microkernel being one possible real-time kernel candidate for future developments. Very intensive campaigns have been carried out with LynxOS and a detailed account of results reported to the company.

The new target microkernel has been analyzed on the same grounds as for the Chorus and divided into semantically similar functional components, namely: synchronization (SYN), memory management (MEM), scheduling (SCH), and communication management (COM). While the source code was available to us in the case of Chorus, this was not the case for LynxOS. Of course, this step is made easier when the source code is available; nevertheless, this could be readily achieved in the case of LynxOS.

The same modular workload was used to carry out both campaigns. Although identical from a specification viewpoint, the workloads used differed slightly to accommodate the basic differences between each target instance. In particular, the interface of the Chorus microkernel standalone instance was proprietary, while the interface of the targeted LynxOS instance partially referred to the POSIX interface. Nevertheless, some of the original predefined workload processes could be ported easily by just translating a given kernel call into a respective one. Some others were redeveloped on the new target from the same specification. The set of kernel calls used in this new implementation was thus different.

In the sequel, we illustrate some of the very distinctive results exhibited by similar functional components. Let us consider first the results concerning the respective synchronization components. Both components support similar functionality, basically, mutex semaphores.

As an example, Fig. 19 shows the results obtained for Chorus and LynxOS when injecting faults into the code segments of the respective synchronization components (SYN).

Although some observations are similar, some are totally different. These discrepancies reveal contrasted design and implementation decisions of the target microkernels that may have a significant impact on the architectural solutions available for integrating the respective microkernels. For instance, in the case of LynxOS, kernel hangs dominate and fewer exceptions are raised; furthermore, the results show that a slightly larger percentage of "Error Status" is returned for LynxOS. The prior discrepancy definitely provides a significant insight in the decision of selecting a microkernel.

When reactive applications are the main focus of the system developer (integrator), then synchronization between threads is a major worry. More exceptions raised allow for local recovery actions to be undertaken (provided the faulty situation can be diagnosed and the erroneous state identified). Conversely, more kernel hangs means that the recovery actions must rely on replication strategies, which has a strong impact on the architectural solution for the overall system. It is worth noting that very distinct results were obtained for the other microkernel components.

Very different behaviors were also observed when injecting faults at the interface (Fig. 20). Parameter fault injection not only reveals weaknesses of the implementation of the target functional component, but also provides significant insights regarding its design and implementation strategies.

As was already mentioned (see Section 4.4.1), the results observed for Chorus reveal that no check of input parameters was implemented as part of the basic synchronization facilities. This design choice is meant to favor

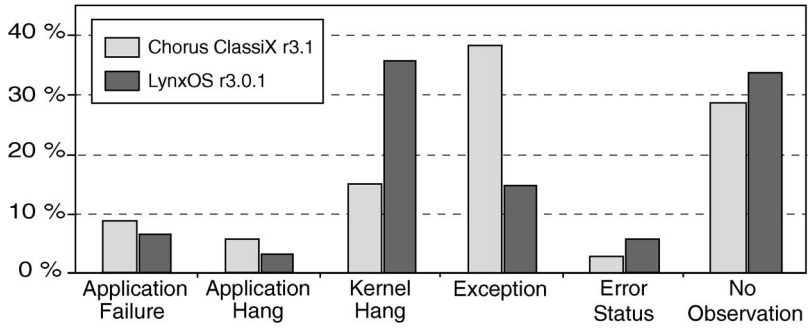

Fig. 19. Injection into SYN component code segment—Chorus vs. LynxOS.

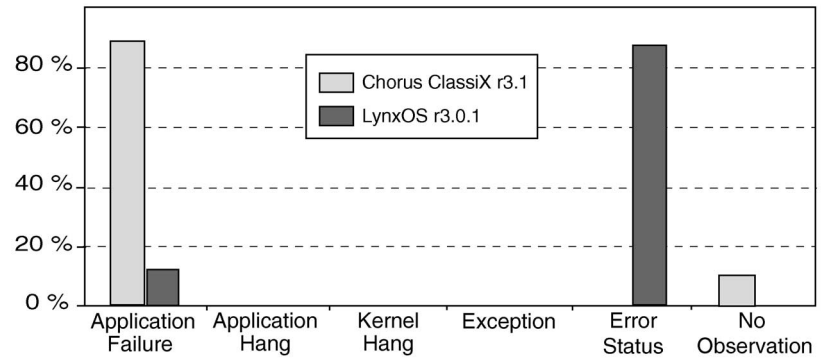

Fig. 20. Injection into SYN component call parameters—Chorus vs. LynxOS.

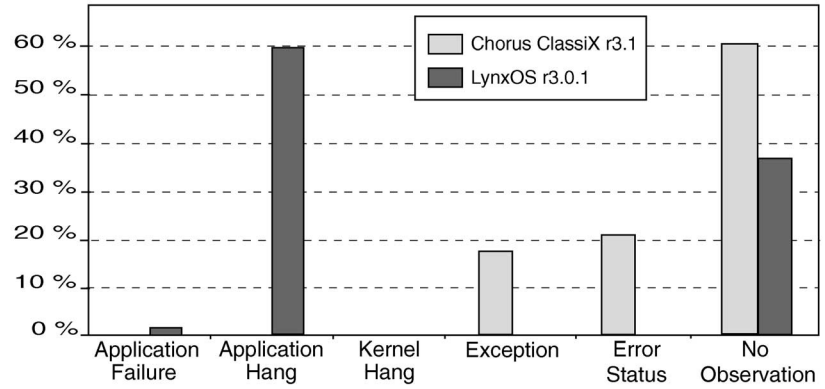

Fig. 21. Injection into MEM component call parameters—Chorus vs. LynxOS.

performance, while it leaves the decision for enhanced error handling (e.g., executable assertions and checks) to the system integrator. The design decision is opposite for LynxOS; the safer behavior exhibited reflects the fact that checks and other assertions are part of the implementation. This shows that using a standalone microkernel (such as the target Chorus instance) into embedded systems with dependability requirements calls for fault containment wrappers or necessitates that the observed behavior in presence of faults be taken into account in the design of the upper layers (i.e., middleware). This is a significant input to system integrators for designing the overall architecture, including the precise definition of error handling mechanisms within middleware layers.

As another example of significant insight provided to system integrators, let us now consider the experiments targeting the memory management component (MEM) by means of parameter fault injection (Fig. 21). The large percentage of application hangs clearly reveals some vulnerability of this component or KPI (Kernel Plug-In) as denoted in the case of LynxOS, at least for the instance we have used.[12]

Conversely, some other components exhibit very similar behaviors in the presence of faults. This is the case for example of the communication component (COM) when subjected to parameter fault injection (Fig. 22). One interesting result of these campaigns is that two experiments (shown as "*" in Fig. 22) resulted in hangs that were observed with Chorus (one application hang and one kernel hang). The reason for these is subtle. In each case, the target parameter was found to be an actor capability, i.e., a raw actor's identifier that is delivered to the kernel as an input parameter. Such a parameter is used *"as is"* (without check) by several internal functions of the kernel. When this identifier is "bit-flipped," then the kernel jumps into some different part of the kernel code, thus resulting in a hang. This explicitly shows that the interface of a kernel and, in particular, the type of parameters of the kernel calls may cause problems when incorrectly used. Direct use of such parameters (or of pointers to user address space data structures as well) is thus likely to lead the kernel or the application to fail.

7.2 Some Lessons Learned

We summarize in this section the various additional insights that have been obtained during the campaigns carried out on the two target microkernels.

7.2.1 Definition of the Workload and the Oracle

Clearly, the definition of the workload processes is a major dimension for a system integrator for conducting some assessment work. They must adequately reflect the use of microkernel services for a given application. For instance, highly reactive applications strongly request synchronization and scheduling services. Dynamic memory services can thus be less important for this kind of application. Conversely, one can consider an application as composed

12. This observed weakness seems to concur with the warnings concerning the vulnerability of the Shared Memory KPI that are explicitly stated in the online user manual.

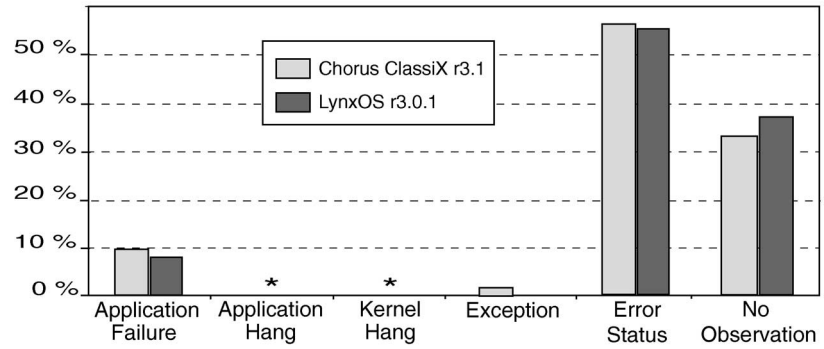

Fig. 22. Injection into COM component call parameters—Chorus vs. LynxOS.

of very simple periodic monothreaded processes responsible for data acquisition, thus highly demanding on dynamic memory for buffer management. In this case, memory management features are very significant and synchronization much less. These examples illustrate the variability of the possible application profiles.

Also, when selecting the workload processes, one must keep in mind that application failures are detected using an application Oracle. When the workload processes are complex, an appropriate abstraction level must be determined to define this Oracle. The latter should include at least a trace of some key features of the application behavior and its major output results. Today, MAFALDA provides some facilities to help the user in defining the Oracle (interception and tracing of library functions and system calls).

7.2.2 Fault Injection

In the reported experiments, target parameters in kernel calls and memory locations within the microkernel address space are selected randomly by MAFALDA. This approach is complementary to the static selection of *invalid* input parameters, as in [25], or the *mutation* (e.g., see [43]) of a given instruction into the microkernel code.

Nevertheless, the tool also enables some specific addresses to be a priori selected as targets for the injection. Such experiments can be used to complement the statistical analyses carried out using random fault selection. For instance, relevant locations (e.g., key addresses within devices drivers) can be used to test some specific portions of the microkernel code. The activation profile must then be defined accordingly to trigger the target code (e.g., see [44]).

Although SWIFI is a flexible fault injection technique that is easy to apply and can simulate a large spectrum of faults, low intrusiveness is another property that is required for fault injection. To minimize intrusion, MAFALDA uses exception handlers separate from the microkernel code (see Sections 6.1 and 6.2) to implement the interception and fault injection modules. Temporal intrusiveness is reduced by the use of the debugging and monitoring facilities offered by modern processors. Nevertheless, temporal intrusiveness remains a concern when hard real-time applications are considered. We are conducting further work to cope with this issue [45]. The approach extends MAFALDA in MAFALDA-RT that forms a testbed where hardware devices (e.g., the Intel 8254 Interval Timer) that handle the temporal events governing the evolution of the real-time tasks can be controlled (stopped/resumed) whenever a fault is to be injected. Of course, an environment simulator is being used as part of the testbed and is preempted accordingly. This way the temporal intrusiveness related to fault injection can be virtually eliminated.

7.2.3 Assertions and Wrappers

The development of assertions is quite complex. It first requires formalizing (formal expression or model) the expected behavior of the microkernel functional component. The implementation of the wrapper can then be derived from this expression or model.

In order to formalize the definition of wrappers and also facilitate their implementation, we have recently developed a framework for the specification of real-time microkernels using a temporal logic [42]. These expressions can be interpreted online by a runtime checker. From the preliminary set of experiments we have carried out, the efficiency is improved and the performance overhead is reasonable.

Besides the improvement of the error detection and confinement capabilities (see Section 5.5), an important practical issue concerns the overhead induced by the wrappers. The first dimension concerns the memory size of the software implementing the wrappers. While the SYN wrapper was only about 100 assembly code lines, the size of the scheduling wrapper was approximately 32Kbytes.[13] Thus, we observed that the wrappers induced very different performance overheads on the executed workloads. This was negligible for the SYN wrapper (the observed overhead in execution time of the executed workloads was less than 10^{-4}). Conversely, a significant increase (about 60 percent) was induced by the SCH wrapper on these workloads. It is worth noting that such an overhead is also very much dependent on the frequency of execution of the wrappers (events due to internal preemption or kernel calls) while the workloads are applied. Accordingly, as the workload processes used in the experiments were tailored to intensively activate the target functional components, it seems reasonable to consider these overheads as upper bounds; the overhead should be lower for a real application program.

13. These figures should be related to the size of the full microkernel instance (1 Mbyte).

7.2.4 Raw Data Analysis

Beyond global statistical results, the detailed information logged by MAFALDA during the experiments enable the user (microkernel supplier or system integrator) to analyze design/programming flaws. Clearly, the analysis of raw data is required when some singular situation has been observed. This analysis leads to a complex job for which the tool already provides some *scripts*, essentially to filter the log files. Examples of results of such analyses were briefly presented in Section 5.5.2.

However, it is sometimes necessary to "dive" into the logged information as for any debugging phase. It is up to the user to develop its own scripts to filter some specific information item and/or to relate different items together. Different criteria can also be defined for the various failure modes observed. Because of the complexity of this job, a database system has recently been added in order to facilitate the organization of the input and output data but also the a posteriori analysis of the logged raw data.

7.2.5 Interpretation of Results

It is important to stress again that the results one can obtain depend very much on the activation profile and on the configuration of the target instance. Accordingly, global results may vary significantly for distinct application domains, e.g., using different instances of the same microkernel, but with slightly different configurations and hardware underlying platforms. This means that the results obtained must be analyzed first in isolation and that the comparison between several candidates must account for these issues.

Indeed, when the API provided is different for several candidates (as was the case for the Chorus and LynxOS instances considered in Section 7.1), the application of similar workload processes might be difficult. In the worst case, the workload requires a different implementation from the same specification. Regarding the configuration of the microkernel itself, the type of API provided to the user may have a strong impact on the results. The same microkernel equipped or not with a POSIX interface, for instance, shows a different behavior, in particular because of the additional checks implemented within this standard interface.

7.2.6 Target System Evolution

MAFALDA enables some evolution of the runtime support software layer to be analyzed and controlled. This is an important issue since the development of any COTS-based system implies taking advantage of the evolution of the COTS components (new releases and versions). Building a new version of the system integrating a new release of the COTS microkernel needs some assessments to be carried out again. Such a campaign corresponds to a form of regression testing that enables the integrator to verify that the new instance is acceptable in its system from the point of view of its failure modes. It is worth noting that such information is very important with respect to the assumptions that are made for the development of the upper layer software (middleware and applications), in particular regarding the design of fault tolerance features.

7.2.7 Integrator vs. Supplier Viewpoints

From a system integrator viewpoint, the type of experiment carried out with MAFALDA makes some weaknesses clearly visible (cf. Section 7.1). This is an important input for the selection of a microkernel candidate. Still, the evidence of weak error detection mechanisms in one of the microkernel component or at its interface can lead to reject the candidate as is, while it might be acceptable with some simple error confinement wrappers. Of course, the final decision must take into account the necessary trade-offs between error confinement efficiency and performance. Basically, the development of wrappers depends on the analysis of the failure modes, the possibility of deriving executable assertions or checks, but also on the availability of the reflection module (see Fig. 3 and Section 6.3). The specification of this module derives from the assertions for which some internal microkernel information must be made visible. When this module has been specified, its implementation depends on the availability of the source code and/or an agreement between the system integrator and the microkernel supplier. This aspect relates, in fact, to the integrator strategy regarding COTS microkernels.

However, from our experience, the implementation of the metainterface by the microkernel supplier is something acceptable. From a supplier viewpoint too, MAFALDA can also be seen as a special debugger that showed evidence to be very useful for identifying weaknesses not raised by standard benchmarking and conformance testing activities.

7.2.8 Porting MAFALDA

MAFALDA was initially used to evaluate the Chorus microkernel. It was then ported to evaluate the LynxOS microkernel. In both cases, the kernels were running on the same Pentium platform.

As shown in Fig. 3, MAFALDA is made up of various components, distributed on two different machines, namely, the target machine and the host machine. Clearly, only the modules of MAFALDA running on the target machine, namely, the workload processes, the injectors (interception and injection modules), and the wrappers (wrapping and reflection modules), need to be ported. The following activities are mandatory to port MAFALDA to a different microkernel:

1. *Analyze the new microkernel*. This implies generating (i.e., compiling) an instance of the kernel, identifying the addresses and boundaries of each kernel component on memory and classifying the system calls provided by the kernel interface.
2. *Adapt the synthetic workload*. This means modifying its source code to match the new kernel interface (if necessary) and compiling it on the new microkernel.
3. *Compile the injectors*. When the hardware platform is different, it is also necessary to *adapt* the part of the injectors written in assembly language to match the new hardware architecture.
4. *Adapt the wrappers*. This implies implementing the corresponding observation interface (the *reflection*

TABLE 4
Specific Porting Activities (from Chorus to LynxOS)

Activities	Comments	Duration*
Analyze the new target microkernel	LynxOS offers a graphical interface to select kernel components (also called KPIs) and generate a kernel instance. The addresses and boundaries of each KPI in memory were easily found. The proprietary kernel interface of LynxOS is different to that of Chorus. The LynxOS interface is closer to POSIX.	3
Adapt the synthetic workload processes	The workload had to be adapted to match the new semantics of the memory and the communication components of LynxOS. However, the part of the workload related to synchronization and scheduling remained unchanged.	3
Compile the injectors	The injectors used on Chorus consisted of supervisor threads. The injectors had to be written in the form of drivers for LynxOS.	2
Adapt the wrappers	Not carried out, because the source code of LynxOS was not available.	
Adapt the host-target communication	Chorus provides a proprietary remote shell to remotely execute processes. LynxOS uses a standard Unix remote shell. Chorus provides a proprietary console utility, based on UDP, which was used to retrieve the evaluation results from the target machines. In LynxOS, it was performed using standard Unix NFS, so the evaluation results were written directly on a file placed on the host machine.	2

* person-weeks

module of Fig. 3) on the new kernel and compiling the wrappers.

5. *Adapt communication between the target machine and the host machine*. This means that:
   a. The Controller program must be able to launch a process on the target machine (i.e., to request to run the synthetic workload, the injectors, and the wrappers);
   b. The Controller program must be able to retrieve the results issued from the fault injection campaigns.

Table 4 summarizes the specific activities carried out during the porting from Chorus to LynxOS, together with their duration in person-weeks (excluding porting activity 4).

8 Conclusion

In its current prototype form, MAFALDA provides a comprehensive and unique framework that supports both the dependability assessment of COTS microkernels and their integration into computer systems. Although MAFALDA can contribute to sorting out various microkernel options available to a system developer, the main practical interest of the tool concerns the design aid capabilities provided to support the integration of the candidate mikrokernel.

The interest of MAFALDA encompasses application domains with stringent dependability requirements (e.g., avionics), but also systems that are critical from an economical viewpoint (e.g., mobile telecommunication) or both (e.g., automobile). The experiments conducted and the results presented illustrate the added-value provided by MAFALDA compared to standard benchmarking and conformance testing procedures. The industrial potential of MAFALDA is also asserted by the interest expressed by the industrial partners both within the Laboratory for Dependability Engineering ($\mathcal{LIS}$)[14] and beyond.

The campaigns carried out with MAFALDA on two target microkernels instances (Chorus and LynxOS) have provided many insights. The experiments carried out revealed that the contribution of MAFALDA to the design and validation of systems based on COTS microkernels has two main facets:

1. It may help microkernel *integrators* in the selection of a candidate microkernel according to the observed behavior in the presence of faults and it also provides the means to improve its behavior thanks to the notion of wrappers that can be tailored according to dependability requirements.
2. It may also benefit microkernel *suppliers* by contributing objective data to better design and implement their product. This is, of course, of high interest when the delivered microkernel itself has been customized to fit the customer's requirements.

It is worth noting that, besides pseudorandom selection of locations to be corrupted, a deterministic selection would make it possible to reduce the duration of the campaigns. Still, in that case, advances have to be made, in particular building on formal testing approaches (e.g., see [46], [47]) in order to efficiently guide the testing process.

Another important issue is the link of the results obtained with the workload processes being considered. In particular, it was interesting to see that workload processes originally developed for the Chorus microkernel could be reused (with minor adaptation) in the case of LynxOS. Nevertheless, the definition of a precise workload profile for a given application target is of major importance to getting increased confidence in the results obtained.

14. Hosted by LAAS, $\mathcal{LIS}$ was a Cooperative Laboratory gathering Airbus France, Astrium, Électricité de France, Technicatome, THALES, and LAAS-CNRS.

Portability of the assessment and design-aid methodology supported by MAFALDA to different platforms is another important issue. The experiments on Chorus and LynxOS microkernels were carried out on a Pentium-based platform. As far as we have identified from our experience in porting MAFALDA from Chorus to LynxOS, this porting could be achieved in about 10 person-weeks.

Further developments of MAFALDA are still underway. A new version of the tool targeting real-time applications is currently being developed. The assessment of failure modes is extended by considering timing failures, such as deadline-miss. This new version, called MAFALDA-RT, is currently being tested on the Chorus microkernel featuring a priority ceiling protocol. A specific workload application requiring real-time constraints has been developed too. Other related work in progress concerns the development of wrappers from the formal specifications of the properties required by a real-time kernel [42].

ACKNOWLEDGMENTS

This work was partially financed by the French Ministry of Research and Education, ESPRIT Project 20072, Design for Validation (DeVa) and IST Project 11585, Dependable Systems of Systems (DSoS). Part of this work was carried out in the framework of $\mathcal{LIS}$, a cooperative laboratory between industry and academia. In particular, Manuel Rodríguez was supported by THALES E&C. The authors would like to thank Jean-Claude Laprie (LAAS-CNRS) for his insightful comments made on this work. The help and support from Jean-Michel Sizun (now with EADS Astrium) and François Scheerens (THALES Avionics) are also acknowledged. The authors would like to thank the anonymous reviewers and guest editors for this special issue whose detailed and constructive comments made on an earlier version of the manuscript significantly contributed to improving this paper. The alphabetical ordering of the authors' names does not imply any seniority in authorship.

REFERENCES

[1] J.-C. Fabre, F. Salles, M. Rodríguez Moreno, and J. Arlat, "Assessment of COTS Microkernels by Fault Injection," *Dependable Computing for Critical Applications (Proc. Seventh IFIP Working Conf. Dependable Computing for Critical Applications (DCCA-7))*, C.B. Weinstock and J. Rushby, eds., pp. 25-44, Jan. 1999.

[2] F. Salles, M. Rodríguez, J.-C. Fabre, and J. Arlat, "Metakernels and Fault Containment Wrappers," *Proc. 29th IEEE Int'l Symp. Fault-Tolerant Computing (FTCS-29)*, pp. 22-29, 1999.

[3] M. Rodríguez, F. Salles, J.-C. Fabre, and J. Arlat, "MAFALDA: Microkernel Assessment by Fault Injection and Design Aid," *Proc. Third European Dependable Computing Conf. (EDCC-3)*, pp. 143-160, 1999.

[4] J.-C. Fabre, M. Rodríguez, J. Arlat, F. Salles, and J.-M. Sizun, "Building Dependable COTS Microkernel-Based Systems Using MAFALDA," *Proc. 2000 Pacific Rim Int'l Symp. Dependable Computing (PRDC-2000)*, pp. 85-92, 2000.

[5] *OSE Real Time Kernel*, OSE Systems Inc. (ENEA group), Täby, Sweden, 1997 (see also: http://www.ose.com).

[6] *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems, Part 3: Software Requirements*, Int'l Electrotechnical Commission (IEC), Standard Document no. 61508-3, first ed., 1998.

[7] H. Kantz and C. Koza, "The ELEKTRA Railway Signalling-System: Field Experience with an Actively Replicated System with Diversity," *Proc. 25th Int'l Symp. Fault-Tolerant Computing (FTCS-25)*, pp. 453-458, 1995.

[8] "Chorus/ClassiX r3—Technical Overview,"Chorus Systems, Technical Report no. CS/TR-96-119.8, 1996.

[9] *LynxOS Real-Time Operating System*, LynuxWorks (formally Lynx RTS), 2000.

[10] *VxWorks Realtime Kernel*, WindRiver Systems, 1998.

[11] J.-C. Laprie, "Dependable Computing: Concepts, Limits, Challenges," *Proc. 25th Int'l Symp. Fault-Tolerant Computing (FTCS-25), Special Issue*, pp. 42-54, 1995.

[12] D. Powell, G. Bonn, D. Seaton, P. Veríssimo, and F. Waeselynck, "The Delta-4 Approach to Dependability in Open Distributed Computing Systems," *Proc. 18th Int'l Symp. Fault-Tolerant Computing Systems (FTCS-18)*, pp. 246-251, 1988.

[13] D. Powell, J. Arlat, L. Beus-Dukic, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabéjac, and A. Wellings, "GUARDS: A Generic Upgradable Architecture for Real-Time Dependable Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 10, no. 6, pp. 580-599, June 1999.

[14] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault Injection for Dependability Validation—A Methodology and Some Applications," *IEEE Trans. Software Eng.*, vol. 16, no. 2, pp. 166-182, Feb. 1990.

[15] M.-C. Hsueh, T.K. Tsai, and R.K. Iyer, "Fault Injection Techniques and Tools," *Computer*, vol. 30, no. 4, pp. 75-82, Apr. 1997.

[16] J.V. Carreira, D. Costa, and J.G. Silva, "Fault Injection Spot-Checks Computer System Dependability," *IEEE Spectrum*, vol. 36, pp. 50-55, Aug. 1999.

[17] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System," *IEEE Trans. Computers*, vol. 44, no. 2, pp. 248-260, Feb. 1995.

[18] M. Rimén, J. Ohlsson, and J. Torin, "On Microprocessor Error Behavior Modeling," *Proc. 24th Int'l Symp. Fault-Tolerant Computing (FTCS-24)*, pp. 76-85, 1994.

[19] D.R. Avresky, S.J. Geoghegan, and P.K. Tapadiya, "A Software-Based Fault Injection Tool," *Int'l J. Computer Systems Science and Eng.*, vol. 13, no. 6, pp. 125-135, Nov. 1998.

[20] E. Fuchs, "Validating the Fail-Silence of the MARS Architecture," *Dependable Computing for Critical Applications (Proc. Sixth IFIP Int'l Working Conf. Dependable Computing for Critical Applications (DCCA-6))*, M. Dal Cin, C. Meadows, and W.H. Sanders, eds., pp. 225-247, 1998.

[21] Z. Kalbarczyk, G. Ries, M.S. Lee, Y. Xiao, J. Patel, and R.K. Iyer, "Hierarchical Approach to Accurate Fault Modeling for System Evaluation," *Proc. Int'l Computer Performance and Dependability Symp. (IPDS '98)*, pp. 249-258, 1998.

[22] R. Johansson, "On Single Event Upset Error Manifestation," *Proc. First European Dependable Computing Conf. (EDCC-1)*, pp. 217-231, 1994.

[23] H. Madeira, D. Costa, and M. Vieira, "On the Emulation of Software Faults by Software Fault Injection," *Proc. Int'l Conf. Dependable Systems and Networks (DSN-2000)*, pp. 417-426, 2000.

[24] J. Carreira, H. Madeira, and J.G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers," *IEEE Trans. Software Eng.*, vol. 24, no. 2, pp. 125-136, Feb. 1998.

[25] P. Koopman and J. DeVale, "Comparing the Robustness of POSIX Operating Systems," *Proc. 29th Int'l Symp. Fault-Tolerant Computing (FTCS-29)*, pp. 30-37, 1999.

[26] W.-L. Kao, R.K. Iyer, and D. Tang, "FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults," *IEEE Trans. Software Eng.*, vol. 19, no. 11, pp. 1105-1118, Nov. 1993.

[27] W.-L. Kao and R.K. Iyer, "DEFINE: A Distributed Fault Injection and Monitoring Environment," *Fault-Tolerant Parallel and Distributed Systems*, D. Pradhan and D.R. Avresky, eds., pp. 252-259, Los Alamitos, Calif.: IEEE CS Press, 1995.

[28] W. Cheswick and S. Bellovin, *Firewalls and Internet Security: Repelling the Willy Hacker*. Addison Wesley, 1994.

[29] J.M. Voas, "Certifying Off-the-Shelf Software Components," *Computer*, vol. 31, no. 6, pp. 53-59, June 1998.

[30] J.-M. Ayache, P. Azéma, and M. Diaz, "Observer: A Concept for Detection of Control Errors in Concurrent Systems," *Proc. Ninth Int'l Symp. Fault-Tolerant Computing (FTCS-9)*, pp. 79-85, 1979.

[31] A. Mahmood, D.M. Andrews, and E.J. McCluskey, "Executable Assertions and Flight Software," *Proc. Sixth Digital Avionics Systems Conf.,* pp. 346-351, 1984.
[32] C. Rabéjac, J.-P. Blanquart, and J.-P. Queille, "Executable Assertions and Timed Traces for On-Line Software Error Detection," *Proc. 26th Int'l Symp. Fault-Tolerant Computing (FTCS-26),* pp. 138-147, 1996.
[33] M. Hiller, "Executable Assertions for Detecting Data Errors in Embedded Control Systems," *Proc. Int'l Conf. Dependable Systems and Networks (DSN-2000),* pp. 24-33, 2000.
[34] P. Maes, "Concepts and Experiments in Computational Reflection," *Proc. Conf. Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '87),* pp. 147-155, 1987.
[35] F. Salles, "Dependability of Microkernel-Based Operating Systems: Failure Mode Analysis and Error Confinement," doctorate dissertation, Paul Sabatier Univ., Toulouse, France, 1999.
[36] M. Rozier et al., "Overview of the CHORUS Distributed Operating Systems," Technical Report no. CS/TR-90-25.1, Chorus Systems, 1991.
[37] A.S. Tanenbaum, *Operating Systems: Design and Implementation.* Englewood Cliffs, N.J.: Prentice Hall, 1987.
[38] N. Audsley and A. Wellings, "Analysing APEX Applications," *Proc. Int'l Real-Time Systems Symp. (RTSS '96),* pp. 39-44, 1996.
[39] A. Burns, R.I. Davis, and S. Punnekkat, "Feasibility Analysis of Fault-Tolerant Real-Time Task Sets," *Proc. Euromicro Real-Time Systems Workshop,* pp. 29-33, 1996.
[40] J. Lehoczky, "Real-Time Queueing Network Theory," *Proc. Int'l Real-Time Systems Symp. (RTSS '97),* pp. 220-229, 1997.
[41] B. Dutertre, "Formal Analysis of the Priority Ceiling Protocol," *Proc. 21st Real-Time Systems Symp. (RTSS 2000),* pp. 151-160, 2000.
[42] M. Rodríguez, J.-C. Fabre, and J. Arlat, "Formal Specification for Building Robust Real-time Microkernels," *Proc. 21st Real-Time Systems Symp. (RTSS 2000),* pp. 119-128, 2000.
[43] M. Daran and P. Thévenod-Fosse, "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations," *Proc. Int'l Symp. Software Testing and Analysis (ISSTA '96),* pp. 158-171, 1996.
[44] J. Arlat, J. Boué, and Y. Crouzet, "Validation-Based Development of Dependable Systems," *IEEE Micro,* vol. 19, no. 4, pp. 66-79, July-Aug. 1999.
[45] M. Rodríguez, J.-C. Fabre, and J. Arlat, "Dependability Assessment of Real-Time Systems," Research Report no. 01-189, LAAS-CNRS, May 2001.
[46] D. Avresky, J. Arlat, J.-C. Laprie, and Y. Crouzet, "Fault Injection for the Formal Testing of Fault Tolerance," *IEEE Trans. Reliability,* vol. 45, no. 3, pp. 443-455, 1996.
[47] A. Arazo and Y. Crouzet, "Formal Guides for Experimentally Verifying Complex Software-Implements Fault Tolerance Mechanisms," *Proc. Seventh Int'l Conf. Eng. of Complex Computer Systems (ICECCS 2001),* pp. 69-79, 2001.

Jean Arlat (M'80) received the Engineer degree from the National Institute of Applied Sciences of Toulouse in 1976 and the PhD and Docteur ès-Sciences degrees from the National Polytechnic Institute of Toulouse in 1979 and 1990, respectively. He is currently *Directeur de Recherche* of CNRS, the French National Organization of Scientific Research and a member of the research group on Dependable Computing and Fault Tolerance at LAAS-CNRS. His research interests focus on the dependability evaluation of fault-tolerant systems and the experimental dependability characterization of off-the-shelf software components. He has written more than 80 papers on these subjects and was involved in several national and European research projects (currently, DSOS and DBench) as well as international cooperations. From January 1997 until June 2000, he led LIS (*Laboratoire d'Ingénierie de la Sûreté de fonctionnement*: laboratory for dependability engineering) hosted by LAAS. Since January 1999, he has been chairman of IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance. He is a member of the ACM, the IEEE, the IEEE Computer Society, and the SEE Working Group on Dependable Computing.

Jean-Charles Fabre (M'97) received the PhD degree in computer science in 1982 and the HDR (*Habilitation à Diriger les Recherches*) in 1992 from the University of Toulouse. He was first involved in the Chorus project at INRIA in Paris and responsible for the design and implementation of fault tolerance strategies in the Chorus microkernel-based distributed architecture. Since 1984, he has been with the research group on Dependable Computing and Fault Tolerance at LAAS-CNRS. He is currently *Chargé de Recherche* of CNRS. His research interests concern distributed algorithms, object-oriented development of fault and intrusion tolerant systems, and validation of COTS executive components by fault-injection. He is the main designer of the FRIENDS system, a fault and intrusion-tolerant distributed architecture based on reflective technology and CORBA. Author or coauthor of more than 50 publications, he has also been involved in several European projects (currently DSOS and DBench) and international cooperations. He is a member of the IEEE Computer Society, the French Chapter of ACM-SIGOPS, and the SEE Working Group on Dependable Computing.

Manuel Rodríguez received the MS degree from the Polytechnic University of Valencia, Spain, in 1998. He is currently a PhD student in the Research Group on Dependable Computing and Fault Tolerance at LAAS-CNRS in Toulouse, France. His current research interests focus on dependability evaluation and improvement of real-time systems, involving fault injection techniques, error containment approaches, and formal methods. His research work contributed to the cooperative laboratory LIS and is currently part of the DSOS IST project. He is a member of the SEE Working Group on Dependable Computing.

Frédéric Salles joined the research group on Dependable Computing and Fault Tolerance at LAAS-CNRS in 1994 to prepare his Masters and Doctorate degrees in computer science. He received the PhD degree in 1999 from the University of Toulouse. His main research interests during this period were the dependability assessment of microkernel-based operating systems, based on failure mode analysis and error confinement. He is currrently with Sun Microsystems in Palo-Alto, California, where he is involved in the design and development of high availability software.