Wrapping Real-time Systems from Temporal Logic Specifications

Manuel Rodríguez, Jean-Charles Fabre and Jean Arlat

LAAS-CNRS, 7 Avenue du Colonel Roche, 31077 Toulouse Cedex 4 — France
E-mail: {rodriguez, fabre, arlat}@laas.fr

Abstract. This paper defines a methodology for developing wrappers for real-time systems starting from temporal logic specifications. Error confinement wrappers are automatically generated from the specifications of the target real-time system. The resulting wrappers are the executable version of the specifications, and account for both timing and functional constraints. They are executed on-line by a runtime checker, a sort of virtual machine that interprets temporal logic. A reflective approach is used to implement an observation layer placed between the runtime checker and the target system. It allows the wrappers to obtain the necessary event and data items from the target system so as to perform at runtime the checks defined by the temporal logic specifications. The proposed method has been applied to the use of real-time microkernels in dependable systems. Fault injection is used to assess the detection coverage of the wrappers and analyze trade-offs between performance and coverage.

1. Introduction

A wrapper can be defined in general terms as a software component that sits around a target component or system. Traditionally, wrappers have been used in the security domain (e.g., [1]) to enforce security policies through firewalls.

The notion of wrapper was initially defined by the DARPA Information Science and Technology working group, as a software entity composed of two parts: an *adapter*, providing additional services to applications, and an *encapsulation mechanism*, responsible for linking components. This definition is mostly related to interfacing heterogeneous systems.

As far as dependability is concerned, the definition of error confinement wrappers is a crucial issue. The notion of *error confinement wrapper* was defined by Voas [2, 3] in relation with the use of COTS (Commercial Off-The-Shelf) components in the design and implementation of dependable systems. The author distinguishes between input wrappers, which filter syntactically incorrect inputs, and output wrappers, which submit outputs to an acceptance test. An example of such a type of input wrappers for Windows-NT applications is provided in [4].

Error confinement wrappers are built from *executable assertions* [5-7]. Executable assertions can be used during software development, to aid developers in finding faults in the system [5], but also when the system is in operation, as part of fault-tolerance mechanisms [6]. As an example of the latter, the work reported in [8] defines an efficient platform for running wrappers based on executable assertions of COTS microkernels.

When executable assertions are derived from formal specifications of the target system, we talk of *runtime verification* [9-13]. In these works, a monitor checks system constraints at runtime against an executable formal description of the system. The wrapping approach proposed in this paper is related to runtime verification for error confinement.

Although the notion of wrapping is well established today, the means available to support the implementation of wrappers remain very limited. In particular, executable assertions do provide a nice paradigm to implement error confinement. However, no wrapping framework has been defined to specify and integrate portable wrappers to various real-time executive software components.

The aim of this work is to provide a methodology, an implementation framework and supporting tools, to improve the dependability of real-time executive systems by means of error confinement wrappers. The methodology aims at translating formal specifications of system requirements into error confinement wrapping code, and supports the verification of properties at runtime for a given target executive software. The benefits of the wrapping are then evaluated by fault injection techniques.

In our framework, the expected behavior of real-time systems is expressed using temporal logic specifications. Error confinement wrappers are automatically generated by a compilation process from early defined temporal logic specifications of the target real-time system services. The wrappers are the executable version of the specifications, and account for both timing and functional constraints. They are executed on-line by a runtime checker, a sort of virtual machine that interprets temporal logic. A reflective approach is then used to implement an observation layer placed between the runtime checker and the target system. Such a layer allows the wrappers to obtain the necessary event and data items from the target system so as to perform at runtime the checks defined by the temporal logic specifications. Fault injection is then used to evaluate the efficiency of the selected wrappers.

The proposed method has been applied to the use of real-time microkernels in dependable systems. We use software implemented fault injection (SWIFI) to characterize the error detection coverage and the tradeoff between performance and coverage of a set of wrappers generated from the microkernel specifications provided in [14]. The target real-time system used is composed of the Chorus microkernel [15] and the mine drainage control system application [16].

Significant research has been done in the field of runtime verification [9-13, 17]. The work in [9] proposes the concept of *observer* for designing self-checking distributed systems. The observer is an on-line monitor that checks the system behavior against an executable model of the system. In the paper, the observer concept is developed for formal models based on Petri nets and LAN based distributed systems built on a broadcast service. The approach is applied to a virtual ring MAC protocol, the Link and Transport layers in an industrial LAN, and the OSI layering in an open system architecture. Many works have used Real Time Logic (RTL) to monitor timing constraints of real-time tasks at runtime. For instance, in the work reported in [11], timing properties of tasks are modeled in RTL and an efficient runtime monitor is derived from the defined set of constraints. The objective is to detect timing violations as early as possible. The system is viewed as a sequence of event occurrences triggered by tasks and sent to the monitor. The latter detects timing violations by resolving constraints with the actual timestamps of events. The work reported in [12] defines an out-of-time supervisor for programs whose requirements specifications consist of nondeterministic SDL models. The system is viewed as a set of input and output signals that are processed by the supervisor an arbitrary amount of time after their occurrence (i.e., out-of-time). The approach is exemplified with a simple telecom application.

A recent work [18] proposes an interesting approach to translate a past time linear temporal logic formula into an executable algorithm based on dynamic programming.

By analyzing the execution trace of a running program, this algorithm can determine whether the program behavior conforms to its specification. The objective was to monitor the execution of Java programs as part of the NASA PathExplorer project.

The paper is structured as follows. Section 2 provides an overview of the proposed wrapping framework. Section 3 presents briefly some notions about temporal logic and describes the runtime checker as well as the process of translating temporal logic specifications into error confinement wrappers. In Section 4, a reflective approach is defined to provide the wrappers with the information they need so as to check the behavior of a real-time microkernel-based system. In Section 5, fault injection is used to characterize a real-time system encapsulated with a set of microkernel wrappers. Section 6 sketches the conclusion to the paper.

2. Wrapping Framework

The proposed framework is composed of four elements: i) the *reference*, which is the formal description of the system requirements, ii) the *wrapping*, which comprises the wrappers and the runtime checker, iii) the *observation*, which characterizes how the behavior of the system is perceived by the wrappers, and iv) the *Target Software Component* (TSC), which consists of the target component of the system that is to be wrapped. Fig. 1 provides an overall description of this framework. It is worth noting that while this framework is generic and can be used with any TSC, our work focuses on its application to real-time microkernel-based systems.

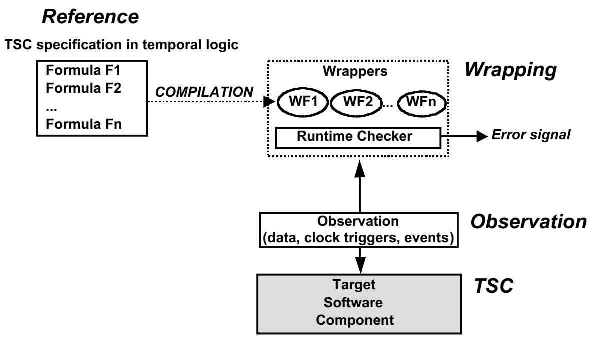


Fig. 1. Overall framework

The reference consists of the *specification* of a set of temporal and functional properties of the TSC that are to be verified at runtime (NB. the issue of proving the specification correct is out of the scope of the paper). This specification is given as a set of formulas (*F1*, *F2*, …, *Fn* in the figure) expressed in future time linear temporal logic [19], which has proved to be a suitable logic for specifying properties of reactive and concurrent systems. The formulas of the specification are written in the form of assertions (i.e., *antecedent* ⇒ *consequent*, which means that the consequent is not checked until the antecedent is true). We have extended some temporal operators of standard temporal logic to manage clock ticks and asynchronous events explicitly. The so-extended temporal logic has been called CETL (see Section 3.1).

A salient feature of our approach is that the wrapping software is divided into two layers, consisting of the *runtime checker* and the *wrappers* (Section 3). The runtime checker is a sort of virtual machine in charge of executing the wrappers (*WF1*, *WF2*, …, *WFn* in Fig. 1). Essentially, the runtime checker is an interpreter of temporal logic that raises an error signal whenever the consequent of an assertion is evaluated to false. Conversely, the wrappers are the executable version of the specification. Their role is to detect timing and value failures of the TSC operation at runtime. We have developed a compiler that automatically translates each assertion into a single wrapper in C language (e.g., *F1* into *WF1*, *F2* into *WF2*, etc.). The compilation process mechanizes the task of writing wrappers from specifications, so it helps make the wrapper code more robust.

The *observation* layer is in charge of providing the necessary TSC information to the runtime checker and the wrappers. Such an information may consist of messages [9], event occurrences [11], signals [12], or states [13]. Indeed, it depends very much on the formalism used to describe the TSC requirements. In our case, temporal logic is built from predicates that describe the internal state of the TSC at different instants of time signaled by clock triggers and event occurrences. Accordingly, the type of information we need to observe correspond to internal TSC data, clock triggers and asynchronous events, as indicated in Fig. 1. Note also that the observation layer makes the runtime checker and the wrappers independent from the particular implementation of the underlying TSC. In other words, when different implementations of the same TSC are to be tested (e.g., different implementations of the same POSIX interface), only the observation layer must be modified. We have used a *reflective* approach [20] to develop such an observation layer, which is described in Section 4.2 in the framework of real-time microkernels.

Finally, note that the wrapping code (wrappers and runtime checker) can run either in a separate machine or in the same target machine. In the first case, the wrapping code does not introduce any temporal overhead, and can also run asynchronously with the target component. In this paper, we deal with the second case, which is, in our opinion the most complex situation, given that the temporal overhead introduced by the wrapping code has to be considered. During the testing phase (see Section 5.1), we eliminate such an overhead by means of the evaluation tool used (MAFALDA-RT). In this way, the original execution times of the system are preserved, and it is possible to obtain precise evaluation measurements. During the operational phase, however, the wrapping overhead must be taken into account so as to check whether task deadlines are met (see Section 5.2).

3. From Temporal Logic Specifications to Error Confinement Wrappers

This section describes the way temporal logic specifications are translated into wrappers. First, we introduce the extensions done to standard future time linear temporal logic. Wrappers are executed on-line by the runtime checker. Accordingly, we then describe this important component of our approach. Finally, we illustrate with a simple example the process of translating an assertion into an error confinement wrapper.

3.1. The Temporal Logic CETL

Linear temporal logic is built on the notion of sequence of states, and does not take into account the type of event that originates a state change. As we are interested in differentiating several types of events triggering a state change, we have extended some of the standard operators of linear temporal logic (i.e., operators *next* ($\bigcirc$) and *sometime* ($\Diamond$)) in order to deal with two types of events: *clock triggers* and *asynchronous events*. Clock triggers (or *ticks*) correspond to the interrupts triggered by the clock of a real-time system (normally, every 10 ms.). Asynchronous events correspond to specific actions leading to state changes in an asynchronous way (for instance, system calls issued by real-time tasks or interrupts triggered by external events, signals or messages). The extended operators that we are considering are the following: $\bigcirc$ (next state triggered by a clock trigger), $\odot$ (next state triggered by an asynchronous event), $\Diamond[e]$ (some future state triggered by the asynchronous event called *e*), $\Diamond[e]^{<k}$ (some future state triggered by the asynchronous event called *e* before the occurrence of *k* clock triggers).

The linear temporal logic extended with these temporal operators is referred to in the paper as CETL, for *Clock and Event driven Temporal Logic*. Note that this type of extension is a common practice in the domain of runtime verification. For instance, in [18], past time linear temporal logic is extended with four new temporal operators (called *monitoring operators*), which are more intuitive and compact than the standard temporal operators. The authors call this extended logic *ptLTL*. In the domain of static verification, the work reported in [21] extends operator *until* of future time linear temporal logic to operator *during*, which is more appropriate to check temporal properties of fault tolerant circuits. The so-extended temporal logic is called *TL*.

Our main objective in this paper is to show how executable algorithms (i.e., the wrappers), that are derived from specifications can be efficiently run by a runtime checker, in order to check system properties on-line. Thus, we mainly focus on the practical issues of our approach. Due to space limitation, the complete definitions and semantics of CETL are not included in this paper; the interested reader can refer to [22]. Nevertheless, it is worth noting that an important property of CETL is that the extended operators are equivalent to the standard ones, as long as information concerning the type and the number of triggered states is available.

As any temporal logic, CETL is built from *temporal operators* (e.g., see the operators previously identified) and a *first order logic*. The first order logic is built from Boolean predicates combined with logical operators ($\wedge$,$\neg$, etc.), predicates which in turn are built from variables combined with relational ($<$, $\leq$, etc.) and arithmetic (+, –, etc.) operators. Moreover, in temporal logic a difference is usually made between *state variables* and *constant variables*. State variables refer to the current state of the target system, and hence their value can vary between states. However, the value of a constant variable is fixed all time, and there exists a (implicit) universal quantification over all the constant variables defined in a formula.

3.2. Runtime Checker

Essentially, the runtime checker supports the execution of the wrappers by interpreting the temporal logic CETL. Accordingly, the runtime checker provides an interface to the wrappers with services for managing the temporal operators (Table 1a), the predicates of the antecedent and the consequent of an assertion (Table 1b), and the constant variables (Table 1c). Note that, an error is signaled by the runtime checker, when a predicate of the consequent of an assertion is evaluated to false (service *ASSERT*).

The wrappers are executed concurrently by the runtime checker. Concurrency is made possible thanks to the functional decomposition of a wrapper into several tasks that are to be run at different instants. Internally, the runtime checker maintains a sort of *process context block* for each wrapper, which characterizes the state of the execution of the wrapper at different instants. Such an information is referred to as *wrapper context*, and corresponds to the values of the constant variables of a formula for such a particular execution of the wrapper. The related services are listed in Table 1c.

Services (C language)	***Meaning***
a) Management of temporal operators (F is a CETL formula, and e is an event identifier)	
NEXT (k, F, context);	$\bigcirc^k(F)$, F is true at the kth state
NEXT_CLOCK (k, F, context);	$\boldsymbol{\bigcirc}^k(F)$, F is true at the kth state triggered by a tick
NEXT_EVENT (k, F, context);	$\odot^k(F)$, F is true at the kth state triggered by an event
SOMETIME (e, F, context);	◈$[e](F)$, F is true in a future state triggered by event e
K_SOMETIME (e,k,F,context);	◈$[e]^{<k}(F)$, F is true in a future state triggered by event e, before the occurrence of k ticks
b) Management of predicates	
CONDITION (predicates);	*Evaluates predicates of the antecedent*
ASSERT (predicates);	*Evaluates predicates of the consequent and signals an error when false*
c) Management of the wrapper context	
NEW_CONTEXT ();	*Creates a context from a static memory pool*
CONTEXT_SET(value,context,index);	*Assigns parameter value to context[index]*
CONTEXT_GET (context, index)	*Returns the contents of context[index]*
DELETE_CONTEXT (context);	*Deletes a wrapper context*

Table 1. Services provided by the runtime checker

3.3. Error Confinement Wrappers

The translation of a CETL assertion into an algorithm is based on a simple *rewriting* process. Indeed, the original CETL assertion is just rewritten into an algorithm that can be effectively executed by the runtime checker. This rewriting process is carried out by a compiler, which automatically translates a CETL assertion into its corresponding wrapper. The compiler has been developed using PCCTS, a C version of ANTRL [23]. Instead of giving the long and tedious list of the rewriting rules used by the compiler, we illustrate the rewriting process by means of a simple example. Our objective here is that the reader has a general but precise idea of how the global rewriting process works.

Table 2 shows the rewriting of a CETL assertion into an error confinement wrapper. Let us consider assertion AS defined in Table 2a. The antecedent of assertion AS is represented by the term ◈$[e](a{=}t \wedge a{=}u)$, while its consequent corresponds to $\odot(a<u \wedge \boldsymbol{\bigcirc}^2(a>u))$. Variable a is a constant integer number. Note that this way of specifying constant variables is a standard notation in temporal logic.

Assertion AS verifies that, it is always true ($\Box$) that, whenever event e occurs and system variable t is equal to system variable u, at the next occurrence of an asynchronous event, the old value of t (represented by variable a) is lower than u, and two clock triggers later, it is higher than u. The pseudo algorithm in Table 2b implements assertion AS by using runtime checker services. Note that symbol $\Rightarrow$ does not correspond to the standard logical implication, but it means that the consequent of an assertion is not checked until the antecedent is true. For didactic reasons, we provide an equivalent pseudo algorithm in Table 2c that illustrates the behavior of the runtime checker services when executing algorithm in Table 2b. The rewriting process works globally as follows. In the pseudo algorithm shown in Table 2b, each temporal operator of assertion AS is substituted by a call to a temporal operator service of the runtime checker (*SOMETIME*, *NEXT_EVENT*, *NEXT_CLOCK*); the actual values of the variables are obtained by executing a *get_* instruction (*get_t*, *get_u*); predicates of the antecedent are assessed by service *CONDITION*, while those of the consequent are evaluated by service *ASSERT*. Note the particular case of variable a, which is assigned the value of variable t and is never modified in the sequel.

a) Assertion AS

$\forall a \in Z, \Box(\Diamond[e](a = t \wedge a = u) \Rightarrow \odot(a < u \wedge \bigcirc^{2}(a > u)))$

b) Pseudo algorithm with runtime checker services

```
Function AS
int a, u;
loop
   /* Antecedent */
       SOMETIME (e);
       a = get_t ();
       t = get_t ();
       CONDITION (a == t);
       u = get_u ();
       CONDITION (a == u);
   /* Consequent */
       NEXT_EVENT (1);
       u = get_u ();
       ASSERT (a < u);
       NEXT_CLOCK (2);
       u = get_u ();
       ASSERT (a > u);
end loop
```

c) Plain pseudo algorithm equivalent to assertion AS of a)

```
Function AS
int a, u;
loop
      /* Antecedent */
      wait_event (e);
      a = get_t ();
      t = get_t ();
      if (a == t)
         u = get_u ();
         if (a == u)
               /* Consequent */
               wait_event (any);
               u = get_u ();
               if not (a < u)
                    signal_error ()
               else
                    wait_clock_triggers (2);
                    u = get_u ();
                    if not (a > u)
                          signal_error ()
                    end if
               end if
         end if
      end if
end loop
```

d) Error confinement wrapper (C language)

```
int start () {return SOMETIME (e, ANT, null) ;}
  /* Antecedent */
    int ANT (Context* context) {
    int a = get_t ();
    int t = get_t ();
    CONDITION (a == t);
    int u = get_u ();
    CONDITION (a == u);
    context = NEW_CONTEXT ();
    CONTEXT_SET (a, context, 1);
    return NEXT_EVENT (1, CON_1, context);
  }
  /* Consequent */
int CON_1 (Context* context) {
    int a = CONTEXT_GET (context, 1);       /* Retrieve data from the context */
    int u = get_u ();                                   /* Obtain TSC data */
    ASSERT (a < u);                   /* Request the evaluation of predicates */
    return NEXT_CLOCK (2, CON_2, context);     /*Set the next temporal operator*/
  }
int CON_2 (Context* context) {
    int a = CONTEXT_GET (context, 1);
    int u = get_u ();
    ASSERT (a > u);
    DELETE_CONTEXT ();
}
```

Table 2. From a CETL assertion to a wrapper

To handle concurrent evaluations of wrappers at runtime (in a similar way to the concurrent execution of real-time tasks), algorithm in Table 2b must be divided into several routines, one for each temporal operator defined in assertion AS. The resulting algorithm is provided in Table 2d, and corresponds to the error confinement wrapper for assertion AS. Routine *ANT* represents the antecedent, routines *CON_1* and *CON_2* represent the consequent, plus the initialization function *start*). In each routine, a group of predicates is to be checked together against the state of the system at a given instant. Note that the wrapper in Table 2d introduces the notion of wrapper context. The wrapper context is composed of the set of constant variables defined by a formula. Thus, the wrapper context for assertion AS is composed of constant variable *a*. The wrapper context is retrieved at the beginning of each routine (e.g., see routines

CON_1 and *CON_2*), and extended (when applicable) at the end of the routine. The exception to this is the first routine of the antecedent (routine *ANT*), where the wrapper context is not retrieved but created for the first time. The last routine of the consequent (routine *CON_2*) deletes the context. Note that the context can be deleted before by the runtime checker, if a predicate is evaluated to false.

The runtime checker executes a wrapper according to the algorithm shown in Fig. 2. For example, a run of routine *CON_1* in Table 2d is as follows:

– Upon receipt of any event, routine *CON_1* executes and retrieves the wrapper context, composed of variable *a* (*int a = CONTEXT_GET (context, 1)*).
– The value for variable *u* is obtained from the TSC (*int u = get_u ()*).
– Predicate $a < u$ is evaluated (*ASSERT (a < u)*).
– Finally, the subsequent temporal operator is set (*NEXT_CLOCK (2, CON_2, context)*).

```
1. Upon receipt of either a clock trigger or an event occurrence, check whether a wrapper is waiting for it.
2. If not, return control to the TSC.
3. If yes, for each wrapper:
   3.1. Let the wrapper retrieve the context (if applicable).
   3.2. Let the wrapper obtain the data needed for the assessment of the predicates from the TSC.
   3.3. Evaluate the predicates.
        3.3.1. If a predicate part of the antecedent is false, finish the wrapper instance (no error signal is raised).
        3.3.2. If a predicate part of the consequent is false, raise an error signal and finish the wrapper instance.
   3.4. Let the wrapper extend the context (or create or destroy it when applicable).
   3.5. Set up the subsequent temporal operator (if applicable).
4. Return control to the TSC.
```

Fig. 2. Execution steps carried out by the runtime checker

4. Wrapping Real-Time Microkernel-Based Systems

A microkernel is an essential component of a system responsible for providing basic services to upper layers, such as scheduling, synchronization, process management or time management. This section describes how microkernel services specified in CETL can be verified *in practice* by means of error confinement wrappers. A simple example illustrates all the steps described in Fig. 1 when the TSC is a real-time microkernel. We first introduce a simple kernel specification and its corresponding error confinement wrapper. Then, we describe the approach used to observe the internal state of the microkernel. Finally, we exemplify how wrappers execute with the help of the runtime checker.

4.1. Compiling kernel Specifications into Wrappers: Example

Let consider a typical kernel service, namely *Create* (Fig. 3). Fig. 3a gives the CETL specification for the creation of higher priority tasks by means of service *Create*. A comprehensive temporal logic specification of real-time microkernels can be found in [14]. Fig. 3b provides the wrapper generated by the CETL compiler for assertion *Create*. The interpretation of assertion *Create* is as follows. When the running task, represented by *tha*, requests the creation of a higher priority task *thb*, the kernel routine corresponding to service *Create* is then executed (indicated by event $\uparrow$*Create*). Some time later, the kernel inserts the newly created task *thb* into the ready queue (event $\uparrow$*signal*). As the child task has a higher priority than its parent, the latter is preempted after a context switch operation (event $\downarrow$*context_switch*). As a result, child task *thb* is elected to run (predicate *running = thb*), while parent task *tha* is inserted back into the ready queue (predicate $tha \in ready(prio(tha))$).

$$\begin{aligned} \forall tha, thb \in Z,\ \Box(&\Diamond[\uparrow Create]\ (created\_th = thb \wedge running = tha \wedge prio(thb) > prio(tha)) \\ &\wedge \Diamond[\uparrow signal]\ (signaled\_th = thb \wedge running = tha)) \\ &\Rightarrow \odot(event = \downarrow context\_switch \wedge running = thb \wedge tha \in ready\ (prio(tha)))) \end{aligned}$$

a) Assertion Create

```
int start () {
       return SOMETIME (ev_begin_Create, ANT_1, null);
}
int ANT_1 (Context* context) {
       int created_th = get_created_th ();
       int thb = get_created_th ();
       CONDITION (created_th == thb);
       int running = get_running ();
       int tha = get_running ();
       CONDITION (running == tha);
       CONDITION (prio(thb) > prio(tha));
       context = NEW_CONTEXT ();
       CONTEXT_SET (thb, context, 1);
       CONTEXT_SET (tha, context, 2);
       return SOMETIME (ev_begin_signal, ANT_2, context);
}
int ANT_2 (Context* context) {
       int thb = CONTEXT_GET (context, 1);
       int tha = CONTEXT_GET (context, 2);
       int signaled_th = get_signaled_th ();
       CONDITION (signaled_th == thb);
       int running = get_running ();
       CONDITION (running == tha);
       return NEXT_EVENT (1, CON, context);
}
int CON (Context* context) {
       int thb = CONTEXT_GET (context, 1);
       int tha = CONTEXT_GET (context, 2);
       int event = get_event ();
       ASSERT (event == ev_end_context_switch);
       int running = get_running ();
       ASSERT (running == thb);
       ASSERT (isInQueue (tha, ready (prio(tha))));
}
```

b) Wrapper Create

Fig. 3. Assertion *Create* and its associated wrapper

4.2. Using Reflection to observe the Target System

We describe now how the internal state of a microkernel can be observed using *reflection* [20]. In a reflective approach, the target system delivers events to the wrappers (*reification*) and the wrappers get the necessary information from the target system (*introspection*). In addition, reflection also allows the behavior of the target system to be controlled using mechanisms based on the concept of *intercession*. These notions are refined in the next paragraphs.

In a reflective system [24, 25], a clear distinction is made between the so-called *base-level*, running the target system, and the *metalevel*, responsible for controlling and updating the behavior of the target system. Information is provided from the base-level to the metalevel, that becomes *metalevel data* or *metainformation*. Any change in the metainformation is reflected to the base-level. The distinction made between the base-level and the metalevel provides a clear separation of concerns between the functional aspects handled at the base-level and the non-functional aspects (here, error detection and error confinement) handled at the metalevel.

Fig. 4 illustrates the various layers, components and mechanisms that make up the reflective framework. This framework complies with and extends the principles introduced in [8]. Here, the base-level is the real-time microkernel, while the metalevel (the *metakernel)* is composed of both the wrappers and the runtime checker. The association of both layers leads to the notion of *reflective real-time microkernel*.

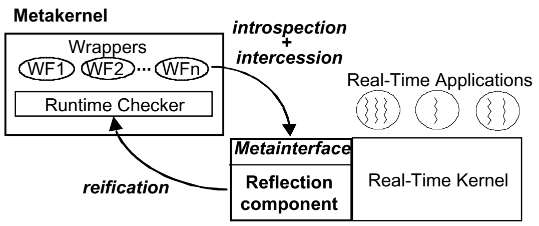


Fig. 4. Reflective framework

The kernel provides the necessary observation through the so-called *reflection component*, which is a special component added to the target microkernel. The reflection component is responsible for the management of the intercepted events (i.e., reification), the observation of internal items (i.e., introspection), and the required actions down into the real-time kernel (i.e., intercession). The reified events are delivered as *upcalls* to the metakernel, whereas introspection and intercession are provided by the reflection component through the so-called *metainterface*. The metainterface is defined as a set of services providing access to the necessary information from and actions into the real-time kernel. It is worth noting, however, that the metainterface we define in this section only considers introspection services.

Reification is carried out using *upcalls*, a jump instruction inserted into the kernel that diverts the execution flow from the kernel to the metakernel, thus not triggering any context switch. For example, assertion *Create* defines event ↑*Create*, which corresponds to the start of the kernel service that carries out the execution of system call *Create*. Accordingly, an upcall is inserted at the beginning of the *Create* routine of the kernel, which takes as an input parameter the identifier of event ↑*Create*. When the kernel enters routine `service_create`, the upcall is executed and diverts execution to the runtime checker. Events ↑*signal* and↓*context_switch* of assertion *Create* are reified in a similar way — see hereafter (left).

```
service_create (...) {              clock_handler (...) {
     upcall (ev_begin_create);           upcall (clock_trigger);
...                                 ...
}                                   }
```

Clock triggers can also be reified by inserting an upcall at the beginning of the `clock handler` routine of the kernel as shown above (right).

On the other hand, introspection consists in obtaining the necessary information through the metainterface. The definition of the metainterface is directly derived from the kernel specification. Indeed, the specification points out the necessary events, data structures and functions of the kernel that must be observed and controlled. To illustrate this point, Table 3 lists the set of services of the metainterface corresponding to assertion *Create*.

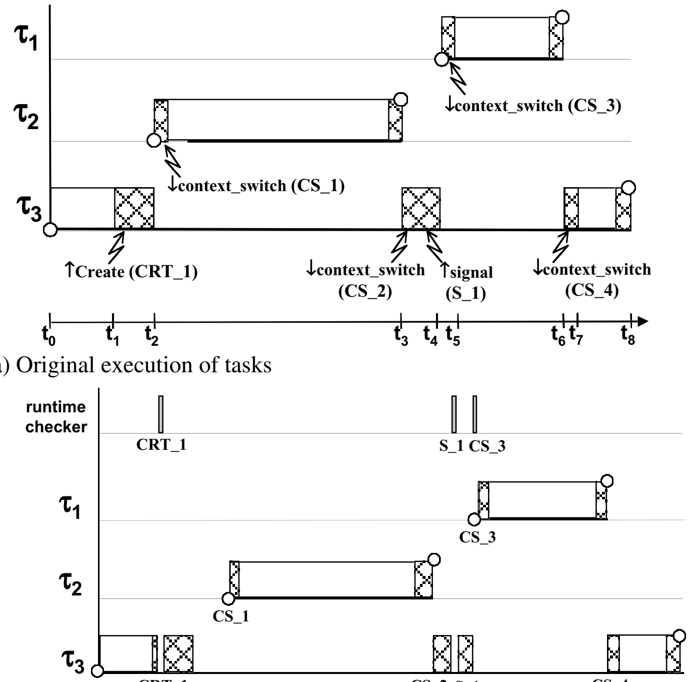
Temporal logic	*Metainterface*		
created_th	`int`	`get_created_th`	`()`
running	`int`	`get_running`	`()`
signaled_th	`int`	`get_signaled_th`	`()`
event	`int`	`get_event_id`	`()`
***prio*(*th*)**	`int`	`prio`	`(int th)`
***ready* (*level*)**	`int`	`ready`	`(int level)`
th* ∈ *queue	`int`	`isInQueue`	`(int th, int queue)`

Table 3. Metainterface necessary to wrapper *Create*

Porting the kernel wrappers to other systems depends on the ability of supplying the microkernel with the adequate reflection component. Indeed, the reflection component makes both the wrappers and the runtime checker independent from the underlying kernel. The specification of the reflection component, comprising both the metainterface and the identified upcalls, remains the same whenever the same set of assertions is used. For instance, the reflection component for assertion *Create* is fully defined by services in Table 3 plus the associated upcalls. Therefore, to port wrapper *Create* to another system it is only necessary to recompile the runtime checker on the new system, as long as the target kernel provides the corresponding reflection component.

4.3. Executing the Error Confinement Wrappers

This section illustrates the execution of error confinement wrappers and the checks they perform on the target system. The target system is represented in Fig. 5 by a set of real-time tasks executing concurrently and requesting kernel service *Create*. Fig. 5a represents the original execution of such tasks together with the events triggered into the microkernel, while Fig. 5b represents the same set of tasks extended with the runtime checker, which executes wrapper *Create*. The horizontal axis represents the pass of time. The vertical axis represents real-time tasks with respect to their priority (τ_1 has higher priority than τ_2, and so on). A white box represents the execution of a task in user mode, while a pattern box represents its execution in kernel mode. A task that enters ready state is represented by a circle on the bottom left corner of the box; a circle on the top right corner means that the task leaves the ready state (e.g., task τ_3 is ready to run during the whole interval represented in Fig. 5a).



a) Original execution of tasks

b) Verification by the runtime checker

Fig. 5. Execution of wrapper *Create* by the runtime checker

t_0	Task τ_3 is running in user mode.
t_1	τ_3 requests the creation of a higher priority task (τ_1) by means of system call *Create*. Event CRT_1 is triggered when task τ_3 enters service *Create* in kernel mode.
t_2	A higher priority task τ_2 preempts tasks τ_3. Task τ_2 obtains the processor after a context switch (event CS_1).
t_2-t_3	Task τ_2 executes.
t_3	Task τ_2 suspends and task τ_3 is given the processor after a context switch (event CS_2). Task τ_3 continues execution of service *Create* in kernel mode.
t_4	Creation of task τ_1 is completed (event S_1).
t_5	Because priority of τ_1 is higher than priority of τ_3, the latter is preempted by its child, which obtains the processor after a context switch (event CS_3).
t_5-t_6	Child task executes.
t_6	Child task ends execution. Its parent obtains the processor after a context switch (event CS_4).
t_7	Task τ_3 finishes executing service *Create* in kernel mode, and continues execution in user mode.
t_8	Task τ_3 ends execution.

Table 4. Original execution of tasks

The detailed behavior of the original set of tasks represented in Fig. 5a is described in Table 4. Wrapper *Create* is executed by the runtime checker during the intervals represented in Fig. 5b, labeled by the kernel event at the origin of the activation of the runtime checker. Remember that the runtime checker is a sort of virtual machine in charge of executing wrappers, which is activated after the occurrence of an event triggered within the target system. Note also that the runtime checker does not preempt, but simply interrupts, the task executing at the moment of its activation, so no context switch is triggered.

In other words, the runtime checker executes at the highest priority on behalf of the running task. In consequence, checks carried out by the wrappers by means of the runtime checker do not modify the original scheduling of tasks, as shown in Fig. 5b. However, the time needed for the wrappers to execute have to be taken into account during the operational phase of the system, in order to check that task deadlines are not violated because of the additional temporal overhead introduced by the wrappers (see Section 5.2).

Each activation of the runtime checker leads thus to the execution of one or several wrappers concurrently. The steps followed by the runtime checker to execute wrapper *Create*, as well as the checks performed by this wrapper, are detailed in Table 5.

Events		***Runtime checker actions***				
Activated	Expected	Wrapper	Routine	Cxt	Expressions checked by services CONDITION and ASSERT	Result
…	CRT					
CRT_1	CRT	Create	ANT_1		created_th == τ1 running == τ3 prio(τ1) > prio(τ3)	TRUE TRUE TRUE
CS_1	S, CRT					
CS_2	S, CRT					
S_1	S, CRT	Create	ANT_2	c1	signaled_th == τ1 running == τ3	TRUE TRUE
CS_3	Any	Create	CON	c1	event == CS running == τ1 isInQueue (τ3, ready (prio(τ3)))	TRUE TRUE TRUE
CS_4	CRT					
…	CRT					

Table 5. Event occurrences and actions carried out by the runtime checker to verify *Create*

Column *Activated event* contains the various events triggered during the execution of the system, while column *Expected event* corresponds to the events waited for by the runtime checker at a given moment. Columns *Wrapper*, *Routine* and *Ctx* refer respectively to the name of the wrapper activated, to the wrapper routine executed, and to the wrapper context used. Column *Expressions checked* reports the verifications performed by the wrappers by means of services *CONDITION* and *ASSERT* of the runtime checker. Note that constant variables *tha* and *thb* have been substituted by the task identifier they represent (τ_1, τ_2, etc.), depending on the information contained into the corresponding wrapper context.

- Initially, given that *Create* is the only wrapper installed, the single event expected by the runtime checker is $\uparrow$*Create* (CRT).
- At the occurrence of event CRT_1, routine ANT_1 of wrapper *Create* is executed. As the child task has higher priority than its parent, event $\uparrow$*signal* (S) is programmed. Context c1 is then allocated with information *tha* = τ_3 and *thb* = τ_1. Next, the runtime checker suspends and waits for events $\uparrow$*Create* and $\uparrow$*signal.*
- Events CS_1 and CS_2 are ignored since they are not expected.
- Event S_1 triggers routine ANT_2 of wrapper *Create* under wrapper context c1. The antecedent of *Create* is then evaluated to true, since the task signaled during the execution of τ_3 is indeed τ_1. The runtime checker waits then for any event.
- At the occurrence of the next event (CS_3), the consequent of *Create* is evaluated under context c1. It is verified that: the event triggered is a context switch, the running task is τ_1, and task τ_3 has been preempted into the ready queue. Since this expression evaluates to true, assertion *Create* succeeds and no error is thus signaled.
- Finally, event CS_4 is ignored and the running checker waits for event $\uparrow$*Create*.

5. Assessment by Fault Injection

We characterize the failure coverage and the performance of wrappers in a real-time system consisting of the *Chorus* microkernel [15] and the *mine drainage control application* [16]. The Chorus kernel was protected by a set of error confinement wrappers derived from an extended kernel specification (see [14]). Note that we first translated these specifications into CETL before compiling them into wrappers. In total, 31 wrappers were used, corresponding to 18 scheduling assertions, 2 timer assertions and 11 synchronization assertions.

The mine drainage control application [16] was used by a number of authors (e.g., [26, 27]). Table 6 shows the main attributes of the tasks of this application. The objective is to pump to the surface mine water collected in a sump at the bottom of the shaft. The main safety requirement is that the pump should not be operated when the level of methane gas in the mine reaches a high value to avoid an explosion. The level of methane is monitored by task CH4 Sensor. Other environment parameters monitored are the level of carbon monoxide (task CO Sensor) and the flow of air in the mine (task Air-Flow Sensor). The flow of water in the pipes of the pump is checked by task Water-Flow Sensor, whereas the water levels in the sump are detected by task Hlw Handler.

Task	Type	Deadline (ms)	Period (ms)	Priority
CH4 Sensor	Periodic	30	80	10
CO Sensor	Periodic	60	100	8
Air-Flow Sensor	Periodic	100	100	7
Water-Flow Sensor	Periodic	40	1000	9
Hlw Handler	Sporadic	200	6000	6

Table 6. Attributes of tasks

MAFALDA-RT [28, 29] was used to assess the error detection coverage and the performance of the kernel wrappers. The tool has been developed to encompass the assessment by fault injection of both hard and soft real-time systems. It provides a facility to eliminate time intrusiveness by controlling the hardware clock of the target system. Such a facility was used to eliminate the temporal overhead introduced both by the tool itself and by the error confinement wrappers. Therefore, tasks were not aware neither of the execution of tool nor of the wrappers from a temporal viewpoint. Note that we are using the wrappers in a testbed system, not in the final system; we are thus interested in evaluating wrapper coverage and wrapper performance without increasing the original execution time of the tasks.

5.1. Error detection Coverage

Table 7 briefly describes the three different fault injection campaigns carried out with and without wrappers. MAFALDA-RT selects randomly the injection target (bits, parameters, etc.) and checks whether the corrupted element is accessed during the experiment, i.e., whether the fault is activated (only activated faults are considered).

↓ Target components Injected fauts →	**Bit-flip**	**Specific (Table 8)**
Priority Ceiling Protocol (PCP)	pPCP (parameters of PCP system calls)	—
Timers (TIM)	mTIM (code segment of TIM)	sTIM

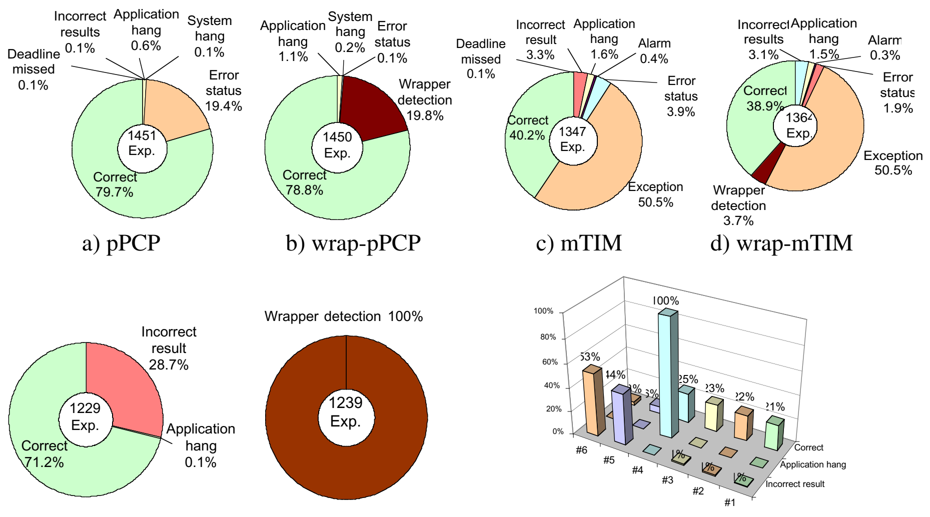
Table 7. Target components and types of injected faults for the three campaigns carried out

The targets of the injected faults were the Priority Ceiling Protocol component (PCP) and the timers component (TIM) of the microkernel. Faults based on bit-flips were uniformly injected over the memory image of the PCP parameters stack (campaign pPCP) and of the timers code segment (campaign mTIM). The specific faults considered in campaign sTIM are specified in Table 8.

#1	Random corruption by single bit-flip of the expiration time of a randomly selected sporadic timer.
#2	Avoiding once the insertion of a randomly selected sporadic timer into the timeout queue.
#3	Avoiding once the deletion of a randomly selected timer from the timeout queue.
#4	Random corruption by single bit-flip of the expiration time of a randomly selected periodic timer.
#5	Avoiding once the insertion of a randomly selected periodic timer into the timeout queue.
#6	Avoiding once the expiration of a randomly selected timer.

Table 8. Specific high-level faults injected in sTIM (only one fault injected by experiment)

Fig. 6 reports both the *first* fault manifestations observed for the standard kernel (campaigns pPCP, mTIM, sTIM), and the corresponding wrapper detection coverage observed for the wrapped version of the kernel (campaigns wrap-pPCP, wrap-mTIM, wrap-sTIM). Few errors impaired the system when the parameters of the synchronization system calls were corrupted (campaign pPCP), because of the high ratio of correct experiments observed (79.7%). This is mostly due to the corruption of unused bits within parameters (random selection by the fault injection tool). Conversely, the consistency checks implemented within the API (represented by class *error status*) detected most errors (19.4%). Few errors (0.9%) could thus propagate and lead to the failure of the application (classes *deadline missed*, *incorrect results*, *application hang* and *system hang*). In the wrapped version of the kernel (campaign wrap-pPCP), the wrappers detected the same class of errors previously detected by means of an error status with a shorter latency. All of them were related to the corruption of a parameter handling a critical section identifier. Obviously, wrappers cannot improve here error detection coverage, since it was already good in the standard kernel.



a) pPCP b) wrap-pPCP c) mTIM d) wrap-mTIM

e) sTIM f) wrap-sTIM g) Fault manifestations (specific faults in sTIM)

- The most critical situation occurs when an error propagates to the application, making it fail either in the time or in the value domain. Timing failures are represented by classes *Deadline missed*, *Application hang* and *System hang*, while value failures are represented by class *Incorrect result*.
- The error detection mechanisms of the microkernel are represented by classes *Alarm*, *Error status* and *Exception*.
- Class *Wrapper detection* represents timing and value errors detected by the wrappers.
- Class *Correct* represents the case when both the time production and the value of the application results are correct.

Fig. 6. *First* fault manifestations and wrapper detection coverage

However, if wrappers are to be used as a support to recovery actions, a wrapper profile as the one represented in Fig. 6b can result interesting.

When the code segment of the timers component was subjected to injection (campaign mTIM), more failures occurred (5.4%) than in campaign pPCP. Still, the majority of the errors were detected by means of an exception (50.5%). Indeed, the kind of errors induced by bit-flips affecting code segment cells corresponds very frequently to low-level errors readily detectable by exceptions (e.g., incorrect operation codes, segmentation faults, etc.). This is the main reason why the wrappers were little activated (3.7%) in campaign wrap-mTIM (also because only first detections are reported). Indeed, the used wrappers have been developed from a high level specification (see [14]), and the type of problems they detect are accordingly also complex. Since most bit-flips were intercepted by exceptions, they could not propagate and originate complex errors.

For that reason, we carried out a new fault injection campaign, sTIM, where the injected faults corresponded to the specific set of high level errors specified in Table 8. As shown by Fig. 6e, 28.7% of the injections led the application to issue incorrect results. Here, the injected faults are out of reach of the exception mechanism. Fig. 6g indicates that 97% of the failures were caused by fault types #5 and #6. Indeed, both of them prevented a periodic task, required for the correct computation of results, from being released. Interestingly, the wrappers avoided all the failures, but also caught all errors that did not previously lead to any observable abnormal situation (Fig. 6f).

In summary, we observed that the error detection mechanisms embedded in the standard kernel provided a high detection coverage with regard to errors caused by bit-flips in system call parameters and in code segment cells. It was thus expected that the improvement provided by error confinement wrappers be either redundant (as in campaign wrap-pPCP) or poor (as in campaign wrap-mTIM). However, using a different fault model, the standard kernel was unable to avoid the propagation of errors to upper layers that provoked an important rate of application failures. The coverage of the error confinement wrappers was then demonstrated, since they systematically prevented the application from misbehaving.

5.2. Worst Case Performance Measurements

Measuring the execution time of the wrappers is of primary importance to determine the feasibility of the wrappers with respect to the timing requirements of the real-time application. During the fault injection experiments carried out with MAFALDA-RT, the execution time of the wrappers could be eliminated (see Section 5.1 and [28]). However, when wrappers are to be integrated into the final system, their execution times must be explicitly taken into account.

Each release of the runtime checker leads to the concurrent execution of several instances of the wrappers. The maximum number of such wrapper instances running concurrently at any time was 9, even though a peak of 47 wrapper instances ready to run was observed. Table 9 shows the worst case overhead (OVH) and the worst case number of releases (REL) of the runtime checker observed in a task instance (i.e., interval between two consecutive releases of a task). The target system used was based on a Pentium running at 75Mhz.

Task	OVH (ms)	REL	OVH/REL
CH4 Sensor	18.119	32	0.566
CO Sensor	6.909	15	0.461
Air-Flow Sensor	6.884	15	0.459
Water-Flow Sensor	6.814	15	0.445
Hlw Handler	13.327	28	0.476

Table 9. Overhead (OVH), releases (REL) and ratio (OVH/REL)

The overhead depends on the number of releases of the runtime checker, i.e., the higher the number of runtime checker releases, the higher the overhead. Conversely, the higher the number of wrapped kernel services requested by a task, the higher the number of releases of the runtime checker. For instance, task CH4 Sensor presents the maximum overhead because it involves more wrapped operations than the other tasks. This means that the runtime checker overhead does not depend on the duration of a task, but rather on its behavior profile, i.e., the type and number of services the task requests to the kernel. For instance, the total overhead introduced by the runtime checker would be the same for two tasks with different computation times but with the same type and number of wrapped services requested. Hence, the overhead of the runtime checker is independent from the execution time of the tasks. This is supported by the low variation of ratio OVH/REL (Table 9). This indicates that a single release of the runtime checker always incurs a similar overhead, no matter the task on behalf of which it is executed. As a result, since the worst case execution times of the wrappers and the runtime checker can be known a priori (e.g., using static code analysis), the exact overhead induced by a given wrapped task can be determined beforehand by analyzing its behavior profile. The overhead can thus be tuned either by selecting the minimum set of wrappers that minimize the ratio between the overhead and the error detection coverage, or by deciding *on-the-fly*, whether enough spare time is available in the system to wrap a service requested by a task. The latter approach is similar to the way aperiodic servers accept or reject the execution of aperiodic tasks [30].

6. Conclusion

This paper proposed a methodology, a framework and supporting tools (e.g., fault injector, temporal logic compiler, etc.) for wrapping real-time systems from temporal logic specifications. System specifications expressed in linear temporal logic are automatically translated by a compiler into error confinement wrappers. Temporal logic provides a consistent way (few operators and state variables in the predicates) for describing the specifications of traditional executive functions. The case study illustrated this feature by considering complex functions, such as scheduling of real-time tasks.

A relevant attribute of our approach is that the wrappers are executed concurrently by a runtime checker, a sort of virtual machine that interprets temporal logic. The reflective approach provides the wrappers with the information they need to check the system behavior at runtime. The reflective software layer makes both the wrappers and the runtime checker independent from the underlying system: only this layer must be modified when different implementations of the same system are to be checked (e.g., different implementations of the same POSIX interface).

This methodology was applied to the wrapping of real-time microkernel-based systems. The behavior of the wrappers and the runtime checker was illustrated with a significant example, based on the verification of a well-known microkernel service (task creation and scheduling). It showed that the execution of the wrappers does not alter the original scheduling of the real-time tasks running on the target system.

The MAFALDA-RT tool was used to evaluate by fault injection (both bit flips and specific faults) a real-time system composed of the Chorus microkernel and a mine drainage control application. In that case, the error detection mechanisms embedded in the original kernel provided already good detection coverage. Accordingly, the wrappers used to protect the kernel could not significantly improve such coverage. However, when a different fault model based on high-level faults was used, most generated errors propagated and provoked the failure of the application. Such failures were systematically avoided by the wrappers. In addition, the performance measures reported showed that the overhead of the wrappers is bounded and can be tuned (e.g., by selecting the minimum set of wrappers minimizing the rate overhead–coverage, or by deciding *on-the-fly* whether enough time is available to execute a wrapper).

It is worth noting that the applicability of the proposed wrapping approach goes beyond real-time kernel functions, and can be of high benefit for various software components and applications. Indeed, it would also benefit embedded real-time systems that cannot accommodate massive redundancy due to weight and/or power constraints.

As a future work, we are currently extending the wrappers with error recovery mechanisms for real-time microkernel-based systems. The objective is not only to detect errors, but also to be able to recover from errors in a bounded time.

Acknowledgement. Manuel Rodríguez has been supported by THALES. The work reported in this paper was partially carried out in the framework of LIS[1] and is currently partially financed by the DSoS project (IST-1999-11585). Our thanks go to Prof. Scott Hazelhurst (Wits University, Joannesburgh, South Africa) for his useful insights on the temporal logic used in this work. The constructive comments made by the anonymous reviewers and by Prof. András Pataricza (Budapest University of Technology and Economics, Hungary) are gratefully acknowledged.

[1] Located at LAAS, the Laboratory for Dependability Engineering (LIS) was a Cooperative Laboratory between five industrial companies (Airbus France, Astrium, Électricité de France, Technicatome, THALES) and LAAS-CNRS.

References

[1] W. R. Cheswick and S. M. Bellovin, *Firewalls and Internet Security*, Addison-Wesley, 1994.

[2] J. Voas, K. Miller, *Interface Robustness for COTS-Based Systems,* Digest no. 97/013, Colloquium on COTS and Safety Critical Systems, IEE, Computing and Control Division, pp. 7/1-7/12, 1997.

[3] J. M. Voas, "Certifying Off-the-Shelf Software Components", *Computer*, pp. 53-59, 1998.

[4] A. K. Ghosh, M. Schmid, F. Hill, "Wrapping Windows NT Software for Robustness", in *Proc. 29th Int. Symp. on Fault-Tolerant Computing,* Madison, WI, USA, pp. 344-347, 1999.

[5] A. Mahmood, D. M. Andrews, E. J. McCluskey, "Executable Assertions and Flight Software", in *Proc. 6th Digital Avionics Systems Conf.,* Baltimore, MD, USA, pp. 346-351, 1984.

[6] C. Rabéjac, J.-P. Blanquart, J.-P. Queille, "Executable Assertions and Timed Traces for On-Line Software Error Detection", in *Proc. 26th Int. Symp. on Fault-Tolerant Computing,* Sendai, Japan, pp. 138-147, 1996.

[7] M. Hiller, "Executable Assertions for Detecting Data Errors in Embedded Control Systems", in *Proc. Int. Conf. on Dependable Systems and Networks,* New York, NY, USA, pp. 24-33, 2000.

[8] F. Salles, M. Rodríguez, J.-C. Fabre, J. Arlat, "Metakernels and Fault Containment Wrappers", in *Proc. 29th Int. Symp. on Fault-Tolerant Computing Systems,* Madison, WI, USA, pp. 22-29, 1999.

[9] M. Diaz, G. Juanole, J.-P. Courtiat, "Observer — A Concept for Formal On-Line Validation of Distributed Systems", *IEEE Trans. on Software Engineering*, vol. 20, no. 12, pp. 900-913, 1994.

[10] F. Jahanian, R. Rajkumar, S. Raju, "Runtime Monitoring of Timing Constraints in Distributed Real-Time Systems", *Real-Time Systems*, vol. 7, no. 3, pp. 247-274, 1994.

[11] A. K. Mok, G. Liu, "Efficient Run-Time Monitoring of Timing Constraints", in *Proc. 3rd Real-Time Technology and Applications Symp.,* Montreal, Canada, pp. 252-262, 1997.

[12] T. Savor, R. E. Seviora, "An Approach to Automatic Detection of Software Failures in Real-Time Systems", *Ibid.,* pp. 136-146, 1997.

[13] Enforceable Security Policies, TR98-1664, Dept. Comp. Science, Cornell Univ., Ithaca, NY (USA), 1998.

[14] M. Rodríguez, J.-C. Fabre, J. Arlat, "Formal Specification for Building Robust Real-time Microkernels", in *Proc. 21st Real-Time Systems Symp,* Orlando, FL, USA, pp. 119-128, 2000.

[15] Chorus Systems, "CHORUS/ClassiX release 3 - Technical Overview", TR CS/TR-96-119.12, Chorus Systems, 1997 (www.sun.com/chorusos).

[16] A. Burns, A. J. Wellings, *Real-time Systems and their Programming Languages*, Addison-Wesley, 1997.

[17] I. Majzik, J. Jávorszky, A. Pataricza E. Selényi, "Concurrent Error Detection of Program Execution Based on Statechart Specification", in *Proc. 10th European Workshop on Dependable Computing,* Vienna, Austria, pp. 181-185, 1999.

[18] K. Havelund, G. Rosu, "Synthesizing Monitors for Safety Properties", in *Proc. Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems,* Grenoble, France, pp. 342-256, 2002.

[19] B. C. Moszkowski, *Executing Temporal Logic Programs*, Cambrige University Press, 1987.

[20] P. Maes, "Concepts and Experiments in Computational Reflection", in *Proc. Conf. on Object-Oriented Programming, Systems and Aplications,* Orlando, FL (USA), pp. 147-155, 1987.

[21] S. Hazelhurst, J. Arlat, "Specifying and Verifying Fault Tolerant Hardware", in *Proc. Designing Correct Circuits,* Grenoble, France, 2002.

[22] M. Rodríguez Moreno, "Wrapping Technology for the Dependability of Real-Time Systems", Doctoral Dissertation, National Polytechnic Institute, Toulouse, France, July 2002. (in French).

[23] ANTLR, *ANTLR Complete Language Translation Solutions*, http://www.antlr.org.

[24] G. Kiczales, J. D. Rivières, D. G. Bobrow, *The Art of the Metaobject Protocol,* MIT Press, 1991.

[25] J.-C. Fabre, T. Pérennou, "A Metaobject Architecture for Fault Tolerant Distributed Systems: The FRIENDS Approach", *IEEE Trans. on Computers*, pp. 78-95, 1998.

[26] A. Burns, A. M. Lister, "A Framework for Building Dependable Systems", *The Computer Journal*, vol. 34, no. 2, pp. 173-181, 1991.

[27] M. Joseph, *Real-Time Systems: Specification, Verification and Analysis*, Prentice-Hall, 1996.

[28] M. Rodríguez, A. Albinet, J. Arlat, "MAFALDA-RT: A Tool for Dependability Assessment of Real-Time in *Proc. Int. Conf. on Dependable Systems and Networks,* Washington, DC, USA, pp. 267-272, 2002.

[29] M. Rodríguez, J.-C. Fabre and J. Arlat, "Assessment of Real-Time Systems by Fault-Injection", in *Proc. European Safety and Reliability Conference,* Lyon, France, pp. 101-108, 2002.

[30] J. A. Stankovic, M. Spuri, K. Ramamritham, G. C. Buttazzo, *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*, Kluwer Academic Publishers, 1998.