Implementing Fault-Tolerance in Real-Time Programs by Automatic Program Transformations

TOLGA AYAV
INRIA and Izmir Institute of Technology, Turkey
and
PASCAL FRADET and ALAIN GIRAULT
INRIA and University of Grenoble, France

We present a formal approach to implement fault-tolerance in real-time embedded systems. The initial fault-intolerant system consists of a set of independent periodic tasks scheduled onto a set of fail-silent processors connected by a reliable communication network. We transform the tasks such that, assuming the availability of an additional spare processor, the system tolerates one failure at a time (transient or permanent). Failure detection is implemented using heartbeating, and failure masking using checkpointing and rollback. These techniques are described and implemented by automatic program transformations on the tasks' programs. The proposed formal approach to fault-tolerance by program transformations highlights the benefits of separation of concerns. It allows us to establish correctness properties and to compute optimal values of parameters to minimize fault-tolerance overhead. We also present an implementation of our method, to demonstrate its feasibility and its efficiency.

Categories and Subject Descriptors: C.3 [**Special Purpose and Application-Based Systems**]: Real-Time and Embedded Systems; C.4 [**Performance of Systems**]: Fault Tolerance; D.2.4 [**Software/Program Verification**]: Formal Methods; D.4.5 [**Reliability**]: Checkpoint/restart

General Terms: Algorithms, Design, Languages, Reliability, Theory

Additional Key Words and Phrases: Fault-tolerance, heartbeating, checkpointing, program transformations, correctness proofs

This work has been funded in part by the ARTIST2 Network of Excellence (`http://www.artist-embedded.org`).
A shorter version of this article has been published in the conference EMSOFT 2006 under the title "Implementing Fault-Tolerance in Real-Time Systems by Automatic Program Transformations."
Author's address: T. Ayav, Izmir Institute of Technology, Department of Computer Engineering, 35430 Urla Izmir, Turkey; emails: TolgaAyav@iyte.edu.tr; P. Fradet and A. Girault, Inria Rhône-Alpes & LIG (PopArt team), Inovallée, 655 Avenue de l'Europe, 38334 Saint-Ismier cedex, France; emails: {Pascal.Fradet, Alain.Girault}@inria.fr.

ACM Reference Format:
Ayav, T., Fradet, P., and Girault, A. 2008. Implementing fault-tolerance in real-time programs by automatic program transformations. ACM Trans. Embedd. Comput. Syst. 7, 4, Article 45 (July 2008), 43 pages. DOI = 10.1145/1376804.1376813 http://doi.acm.org/10.1145/1376804.1376813

1. INTRODUCTION

In most distributed embedded systems, such as automotive and avionics, fault-tolerance is a crucial issue [Cristian 1991; Nelson 1990; Jalote 1994]. It is defined as the ability of the system to comply with its specification despite the presence of faults in any of its components [Avizienis et al. 2004]. To achieve this goal, we rely on two means: *failure detection* and *failure masking*. Among the two classes of faults, hardware and software, we only address the former. Tolerating hardware faults requires redundant hardware, be it explicitly added by the system's designer for this purpose, or intrinsically provided by the existing parallelism of the system. We assume that the system is equipped with one spare processor, which runs a special monitor module, in charge of detecting the failures in the other processors of the system, and then masking one failure.

We achieve failure detection thanks to *timeouts*; two popular approaches exist: the so-called "pull" and "push" methods [Aggarwal and Gupta 2002]. In the pull method, the monitor sends liveness requests (i.e., "are you alive?" messages) to the monitored components, and considers a component as faulty if it does not receive a reply from that component within a fixed time delay. In the push method, each component of the system periodically sends heartbeat information (i.e., "I am alive" messages) to the monitor, which considers a component as faulty if two successive heartbeats are not received by the monitor within a predefined time interval [Aguilera et al. 1997]. We employ the *push method*, which involves only one-way messages.

We implement failure masking with *checkpointing* and *rollback* mechanisms, which have been addressed in many works. It involves storing the global state of the system in a stable memory, and restoring the last state upon the detection of a failure to resume execution. There exist many implementation strategies of checkpointing and rollback, such as user-directed, compiler-assisted, system-level, and library-supported [Ziv and Bruck 1997; Kalaiselvi and Rajaraman 2000; Beck et al. 1994]. The pros and cons of these strategies are discussed in Silva and Silva [1998]. Checkpointing can be synchronous or asynchronous. In our setting where we consider only independent tasks, the simplest approach is asynchronous checkpointing: tasks take *local checkpoints periodically* without any coordination with each other. This approach allows maximum component autonomy for taking checkpoints and does not incur any message overhead.

We propose a framework based on *automatic program transformations* to implement fault-tolerance in distributed embedded systems. Our starting point is a fault-intolerant system, consisting of a set of independent periodic hard real-time tasks scheduled onto a set of fail-silent processors. The goal of the transformations is to automatically obtain a system tolerant to one hardware failure. One *spare processor* is initially free of tasks: it will run a special monitor task, in charge of detecting and masking the system's failures. Each transformation will implement a portion of either the detection or the masking of failures. For instance, one transformation will add the checkpointing code into the real-time tasks, while another one will add the rollback code into the monitor task. The transformations will be guided by the fault-tolerance properties required by the user. Our assumption that all tasks are independent (i.e., they do not communicate with each other) simplifies the problem of consistent global checkpointing, since all local checkpoints belong to the set of global consistent checkpoints.

One important point of our framework is the ability to *formally prove* that the transformed system satisfies the real-time constraints even in the presence of one failure. The techniques that we present (checkpointing, rollback, heartbeating, etc.) are pretty standard in the OS context. Our contribution is to study them in the context of hard real-time tasks, to express them formally as automatic program transformations, and to prove formal properties of the resulting system after the transformations. Another benefit is to allow the computation of optimal checkpointing and heartbeating periods to minimize the recovery time when a failure occurs.

Section 2 gives an overview of our approach. In Section 3, we give a formal definition for the real-time tasks and we introduce a simple programming language. Section 4 presents program transformations implementing checkpointing and heartbeating. We present the monitor task in Section 5 and extend our approach to transient and multiple failures in Section 6. In Section 7, we illustrate the implementation of our approach on the embedded control program of an autonomous vehicle. Finally, we review related work in Section 8 and conclude in Section 9.

2. OVERVIEW OF THE PROPOSED SYSTEM

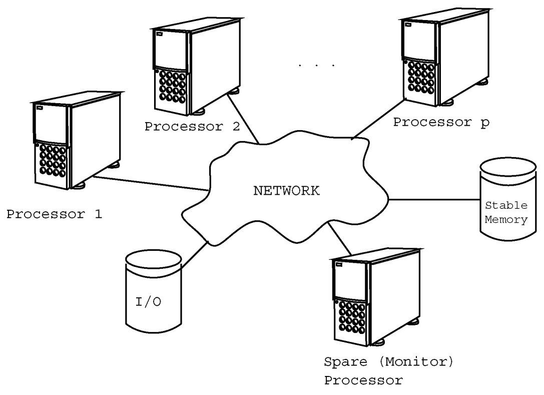
We consider a distributed embedded system consisting of p processors plus a spare processor, a stable memory, and I/O devices. All are connected via a communication network (see Figure 1(a)). We make two assumptions regarding the communication and failure behavior of the processors.

Assumption 1. The communication network is reliable and the transmission time is deterministic.

Moreover, for the sake of clarity, we assume that the message transmission time between processors is zero, but our approach holds for nonzero transmission times as well.

Assumption 2. All processors are fail-silent [Jalote 1994]. This means that the processors may transiently or permanently stop responding, but do not pollute the healthy remaining ones.

The system also has n real-time tasks, each fitting the simple-task model of TTP [Kopetz 1997]: all tasks are periodic and independent (i.e., without precedence constraints). More precisely, the program of each task has the form



(a)

```
Initialize
for each period $T$ do
    Read Inputs
    Compute
    Update Outputs
end for each
```

(b)

Fig. 1. (a) System architecture. (b) Program model of periodic real-time tasks.

described in Figure 1(b). Even though we present our method by assuming this simple-task model, it can perfectly be applied to *dependent* tasks (i.e., with precedence constraints). Indeed, in Section 7, we present such an application with static schedules composed of dependent tasks and deterministic and nonzero communication times, which we solve with our method.

We do not address the issue of distribution and scheduling of the tasks onto the processors. Hence, for the sake of clarity, we assume that each processor runs one single task (i.e., $n = p$). Executing more than one task on each processor (e.g., with a multirate cyclic execution approach) is still possible however.

Our approach deals with the programs of the tasks and defines program transformations on them to achieve fault-tolerance. We consider programs in compiled form at the assembly or binary code level, which allows us to evaluate *exact execution times* (EXET) of the basic instructions and, hence, the *worst-case execution times* (WCET) and *best-case execution times* (BCET) of complex programs having conditional statements. We represent these three-address programs using a small imperative language. Since the system contains only one redundant processor, we provide a masking of only one processor failure at a time. Masking of more than one transient processor failure at a time could be achieved with additional spare processors (see Section 6).

Assumption 3. There exists a stable memory to keep the global state for error recovery purposes.

The stable memory is used to store the global state. The global state provides masking of processor failures by rolling back to this safe state as soon as a failure is detected. The stable memory also stores one shared variable per processor, used for failure detection: the program of each task, after transformation, will periodically write a `1` into this shared variable, while the monitor will periodically (and with the same period) check that its value is indeed `1` and will reset it to `0`. When a failure occurs, the shared variable corresponding to the faulty processor will remain equal to `0`, therefore allowing the monitor to detect the failure. The spare processor provides the necessary hardware redundancy and executes the monitor program for failure detection and masking purposes.

Assumption 4. The communications between the processors and the stable memory are only validated when they have completed successfully (*i.e.,* they are considered as atomic transactions).

The above assumption guarantees that if a processor fails while writing some data into the stable memory (e.g., when performing a checkpoint), then this transaction is not validated.

When the monitor detects a processor failure, it rolls back to the latest local state of the faulty processor stored in the stable memory. It then resumes the execution of the task that was running on the faulty processor from this local state. Remember that, since the tasks are independent, the other tasks do not need to roll back to their own previous local state. This failure-masking process is implemented by an asynchronous checkpointing, i.e., processors take local checkpoints periodically without any coordination with each other.

The two program transformations used for adding periodic heartbeating/failure detection and periodic checkpointing/rollback amount to inserting code at specific points. This process may seem easy, but the conditional statements of the program to be transformed, i.e., `if` branchings, create many different execution paths, making it actually quite difficult. We, therefore, propose a preliminary program transformation, which equalizes the execution times between all the possible execution paths. This is done by padding dummy code in `if` branchings. After this transformation, the resulting programs have a constant execution time. Then, checkpointing and heartbeating commands are inserted into the code at constant time intervals. The periods between checkpoints and heartbeats are chosen in order to minimize their cost while satisfying the real-time constraints. A special monitoring program is also generated from the parameters of these transformations. The monitor consists of a number of tasks that must be scheduled by an algorithm providing deadline guarantees.

The algorithmic complexity of our program transformations is linear in the size of the program. The overhead in the transformed program is the result of fault-tolerance techniques we use (heartbeating, checkpointing and rollback). This overhead is unavoidable and compares favorably to the overhead induced by other fault-tolerance techniques, e.g., hardware and software redundancy.

The memory overhead is also linear in the memory size of the program. Indeed, this overhead results from the need to store the global state of each task when performing the checkpointing. In addition, an array of integers of size n, where n is the total number of tasks, is used for the heartbeating and fault detection.

3. TASKS

A real-time periodic task $\tau = (S, T)$ is specified by a program S and a period T. The program S is repeatedly executed each T units of time. A program usually reads its input (which is stored in a local variable), executes some statements, and writes its output (see Figure 1(b)). Each task also has a deadline $d \leq T$ that it must satisfy when writing its output. To simplify the presentation, we take the deadline equal to the period, but our approach does not depend on this assumption. Hence, the real-time constraint associated to the task (S, T) is that its program S must terminate before the end of its period T.

Programs are written in the following imperative programming language:

S ::=	x:=A	*assignment*
|	`skip`	*no operation*
|	`read`(i)	*input read*
|	`write`(o)	*output write*
|	S_1;S_2	*sequencing*
|	`if` B `then` S_1 `else` S_2	*conditional*
|	`for` $i = n_1$ `to` n_2 `do` S	*iteration*

where A and B denote, respectively, integer expressions (arithmetic expressions on integer variables) and boolean expressions (comparisons, `and`, `not`, etc), and n_1 and n_2 denote integer constants. Here, we assume that the only variables used to store the input and the output are i and o, respectively. These instructions could be generalized to multiple reads and writes or to I/O operations parameterized with a port. This language is well-known, simple, and sufficiently expressive. The reader may refer to Nielson and Nielson [1992] for a complete description.

The following example program *Fac* reads an unsigned integer variable and places it in i. It bounds the variable i by 10 and computes the factorial of i, which it finally writes as its output. Here, *Fac* should be seen as a generic computation simple enough to concisely present our techniques. Of course, as long as they are expressed in the previous syntax, much more complex and realistic computations can be treated as well.

```
Fac = read(i) ;
      if i > 10 then i := 10; o := 1; else o := 1;
      for l = 1 to 10 do
           if l <= i then o := o * l; else skip;
      write(o);
```

The simplest statement of the language is `skip` (the *nop* instruction), which exists on all processors. We take the EXET of the `skip` command to be the unit of time and we assume that the execution times of all other statements are multiple of EXET (`skip`). A more fundamental assumption is that the execution times (be it EXET, WCET, or BCET) of any statement or expression can be evaluated. A more fundamental assumption is that the exact execution times (EXET) of any basic instruction is known. This is required to precisely insert periodic heartbeats and checkpoints. Other techniques (e.g., inserting heartbeats and checkpoints *at least* every x time units) would only require knowing the WCET of each basic instruction. Either way, it is possible to evaluate the WCET and BCET of all statements and programs. Languages with more complex control

Table I. Exact, Worst-Case, and Best-Case Execution Times of Our Programming Language's Statements

$\text{EXET}(\texttt{skip}) = \text{BCET}(\texttt{skip}) = \text{WCET}(\texttt{skip})$	$= 1$
$\text{EXET}(\texttt{read}) = \text{BCET}(\texttt{read}) = \text{WCET}(\texttt{read})$	$= 3$
$\text{EXET}(\texttt{write}) = \text{BCET}(\texttt{write}) = \text{WCET}(\texttt{write})$	$= 3$
$\text{EXET}(x := e) = \text{BCET}(x := e) = \text{WCET}(x := e)$	$= 3$
$\text{EXET}(S_1;S_2)$	$= \text{EXET}(S_1) + \text{EXET}(S_2)$
$\text{BCET}(S_1;S_2)$	$= \text{BCET}(S_1) + \text{BCET}(S_2)$
$\text{WCET}(S_1;S_2)$	$= \text{WCET}(S_1) + \text{WCET}(S_2)$
$\text{WCET}(\texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2)$	$= 1 + \max(\text{WCET}(S_1), \text{WCET}(S_2))$
$\text{BCET}(\texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2)$	$= 1 + \min(\text{BCET}(S_1), \text{BCET}(S_2))$
$\text{EXET}(\texttt{for } i = n_1 \texttt{ to } n_2 \texttt{ do } S)$	$= (n_2 - n_1 + 1) \times (3 + \text{EXET}(S))$
$\text{BCET}(\texttt{for } i = n_1 \texttt{ to } n_2 \texttt{ do } S)$	$= (n_2 - n_1 + 1) \times (3 + \text{BCET}(S))$
$\text{WCET}(\texttt{for } i = n_1 \texttt{ to } n_2 \texttt{ do } S)$	$= (n_2 - n_1 + 1) \times (3 + \text{WCET}(S))$

structures could be considered as well. Of course, this may lead to (potentially very) conservative WCET. In any cases, the WCET of programs *must* be computable to formally prove that deadlines are met.

The WCET analysis is the topic of much work (see Puschner and Burns [2000] and Lisper [2006] for surveys); we shall not dwell upon this issue any further. This is not a critical assumption, since WCET analysis has been applied with success to real-life processors with branch prediction [Colin and Puaut 2000] or with caches and pipelines [Theiling et al. 2000].

For the remainder of the article, we fix the execution times of statements to be (in time units) those of Table I.

Of course, when the EXET of a statement is known, it is also equal to its WCET and its BCET. The above figures are valid for any "simple" expressions e or b. Using temporary variables, it is always possible to split complex arithmetic and boolean expressions so that they remain simple enough (as in three-address code). The WCET (resp. BCET) of the `for` statement is computed in the same way, by replacing EXETby WCET in the right-hand part (resp. BCET); the same thing for the ";".

With these figures, we get $\text{WCET}(Fac) = 84$. In the rest of the article, we consider the task $(Fac, 200)$, that is to say *Fac* with a deadline/period of 200 time units.

The real-time property for a system of n tasks $\{(S_1, T_1), \ldots, (S_n, T_n)\}$ is that each task must meet its deadline. Since each processor runs a single task, it amounts to:

$$\forall i \in \{1, 2, \ldots, n\}, \text{WCET}(S_i) \leq T_i \tag{1}$$

The semantics of a statement S is given by the function $[\![S]\!] : \textbf{State} \rightarrow \textbf{State}$. A state $s \in \textbf{State}$ maps program variables $\mathcal{V}$ to their values. The semantic function takes a statement S, an initial state s_0 and yields the resulting state s_f obtained after the execution of the statement: $[\![S]\!]s_0 = s_f$. Several equivalent formal definitions of $[\![.]\!]$ (operational, denotational, and axiomatic) can be found in Nielson and Nielson [1992].

The IO semantics of a task (S, T) is given by a pair of streams

$$(i_1, \ldots, i_n, \ldots), (o_1, \ldots, o_n, \ldots)$$

where i_k is the input provided by the environment during the kth period and o_k is the last output written during the kth period. So, if several *write*(o) are performed during a period, the semantics and the environment will consider only the last one. We also assume that the environment proposes the same input during a period: several *read*(i) during the same period will result in the same readings.

For example, if the environment proposes 2 as input then the program

$$\texttt{read}(i);\, o := i;\, \texttt{write}(o);\, \texttt{read}(i);\, o := o * i;\, \texttt{write}(o)$$

produces 4 as output during that same period, and not $(2, 4)$. Assuming the sequence of integers as inputs, the IO semantics of *Fac* is:

$$(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, \ldots),$$
$$(0, 1!, 2!, 3!, 4!, 5!, 6!, 7!, 8!, 9!, 10!, 10!, 10!, \ldots)$$

4. AUTOMATIC PROGRAM TRANSFORMATIONS

Failure detection and failure masking rely on inserting heartbeating and checkpointing instructions in programs. These instructions must be inserted such that they are executed *periodically*. We therefore transform a task program such that a heartbeat and a checkpoint are executed every T_{HB} and T_{CP} period of time respectively. Conditional statements complicate this insertion. They lead to many paths with different execution times. It is therefore impossible to insert instructions at constant time intervals without duplicating the code. To avoid this problem, we first transform the program in order to fix the execution time of all conditionals to their worst-case execution time. Intuitively, it amounts to adding dummy code to conditional statements. After this time, equalization, checkpoints, and heartbeats can be introduced simply using the same transformation.

A transformation may increase the WCET of programs. Thus, after each transformation $\mathcal{T}$, the real-time constraint $\text{WCET}(\mathcal{T}(S)) \leq T$ must be checked; thanks to our assumptions on WCET, this can be done automatically.

4.1 Equalizing Execution Time

Equalizing the execution time of a program consists in padding dummy code in least expensive branches. The dummy code added for padding is sequences of skip statements. We write $\texttt{skip}^n$ to represent a sequence of n skip statements: $\text{EXET}(\texttt{skip}^n) = n$. This technique is similar to the one used in "single path programming" [Puschner 2002].

The global equalization process is defined recursively by the following transformation rules, noted $\mathcal{F}$. The rules below must be understood like a case expression in the programming language ML [Milner et al. 1990]: cases are evaluated from top to bottom, and the transformation rule corresponding to the first pattern that matches the input program is performed.

Transformation Rule 1

$$\begin{array}{lll}
1.\ \mathcal{F}[\texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2] & = & \texttt{if}\ B\ \texttt{then}\ \mathcal{F}[S_1];\ \texttt{skip}^{\max(0,\delta_2-\delta_1)}; \\
 & & \quad\ \ \texttt{else}\ \mathcal{F}[S_2];\ \texttt{skip}^{\max(0,\delta_1-\delta_2)}; \\
 & & \textit{with}\ \ \delta_i = \textsc{wcet}(\mathcal{F}[S_i])\ \ \text{for}\ i = 1,2 \\
2.\ \mathcal{F}[\texttt{for}\ i = n_1\ \texttt{to}\ n_2\ \texttt{do}\ S] & = & \texttt{for}\ i = n_1\ \texttt{to}\ n_2\ \texttt{do}\ \mathcal{F}[S] \\
3.\ \mathcal{F}[S_1;S_2] & = & \mathcal{F}[S_1];\ \mathcal{F}[S_2] \\
4.\ \mathcal{F}[S] & = & S \qquad \textit{otherwise}
\end{array}$$

Conditionals are the only statements subject to code modification (Rule 1). The transformation adds as many `skip` as needed to match the execution time of the other branch: hence, the $\max(0, \delta_2 - \delta_1)$ in the `then` branch. The "most expensive" branch remains unchanged, while the "least expensive branch" ends up taking the same time as the most expensive one. The transformation is applied inductively to the statement of each branch prior to this equalization.

We now prove that, for any program S, the best- and worst-case execution times of $\mathcal{F}[S]$ are identical:

PROPERTY 1. $\forall S,\ \textsc{bcet}(\mathcal{F}[S]) = \textsc{wcet}(\mathcal{F}[S]) = \textsc{exet}(\mathcal{F}[S])$.

PROOF. The proof is by induction on the structure of the program S.

—Let $S = \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2$. The induction hypothesis is that $\textsc{bcet}(\mathcal{F}[S_1]) = \textsc{wcet}(\mathcal{F}[S_1]) = \textsc{exet}(\mathcal{F}[S_1]) = \delta_1$ and $\textsc{exet}(\mathcal{F}[S_2]) = \textsc{bcet}(\mathcal{F}[S_2]) = \textsc{wcet}(\mathcal{F}[S_2]) = \textsc{exet}(\mathcal{F}[S_2]) = \delta_2$. According to Rule 1 and Table I, we thus have $\textsc{wcet}(\mathcal{F}[S]) = 1+\max(\delta_1+\max(0, \delta_2-\delta_1), \delta_2+\max(0, \delta_1-\delta_2))$.

Without loss of generality, assume that $\delta_1 \geq \delta_2$ (the symmetrical case yields similar computations). Then $\delta_1 + \max(0, \delta_2 - \delta_1) = \delta_1 + 0 = \delta_1$, and $\delta_2 + \max(0, \delta_1 - \delta_2) = \delta_2 + \delta_1 - \delta_2 = \delta_1$. Hence $\textsc{wcet}(\mathcal{F}[S]) = 1 + \max(\delta_1, \delta_1) = 1 + \delta_1$.

Conversely, we also have $\textsc{bcet}(\mathcal{F}[S]) = 1 + \min(\delta_1 + \max(0, \delta_2 - \delta_1), \delta_2 + \max(0, \delta_1-\delta_2))$. Then, still by assuming that $\delta_1 \geq \delta_2$, we also find $\textsc{bcet}(\mathcal{F}[S]) = 1 + \min(\delta_1, \delta_1) = 1 + \delta_1$.

In conclusion, $\textsc{bcet}(\mathcal{F}[S]) = \textsc{wcet}(\mathcal{F}[S])$ and therefore it is also equal to $\textsc{exet}(\mathcal{F}[S])$.

—Let $S = \texttt{for}\ i = n_1\ \texttt{to}\ n_2\ \texttt{do}\ S_1$. The induction hypothesis is that $\textsc{bcet}(\mathcal{F}[S_1]) = \textsc{wcet}(\mathcal{F}[S_1]) = \textsc{exet}(\mathcal{F}[S_1]) = \delta_1$. According to Rule 2 and Table I, we thus have $\textsc{exet}(\mathcal{F}[S]) = (n_2 - n_1 + 1) \times (3 + \delta_1)$. Since n_1 and n_2 are constant and by induction hypothesis, this is also equal to $\textsc{bcet}(\mathcal{F}[S])$ and $\textsc{wcet}(\mathcal{F}[S])$.

—Let $S = S1;S2$. The induction hypothesis is that $\textsc{bcet}(\mathcal{F}[S_1]) = \textsc{wcet}(\mathcal{F}[S_1]) = \textsc{exet}(\mathcal{F}[S_1]) = \delta_1$ and $\textsc{bcet}(\mathcal{F}[S_2]) = \textsc{wcet}(\mathcal{F}[S_2]) = \textsc{exet}(\mathcal{F}[S_2]) = \delta_2$. According to Rule 3 and Table I, we thus have $\textsc{exet}(\mathcal{F}[S]) = \delta_1 + \delta_2$. By induction hypothesis, this is also equal to $\textsc{bcet}(\mathcal{F}[S])$ and $\textsc{wcet}(\mathcal{F}[S])$.

Thus, we conclude that for any S, $\textsc{bcet}(\mathcal{F}[S]) = \textsc{wcet}(\mathcal{F}[S]) = \textsc{exet}(\mathcal{F}[S])$. □

We also prove that the transformation $\mathcal{F}$ does not change the WCET of programs:

PROPERTY 2. $\forall S,\ \text{WCET}(S) = \text{WCET}(\mathcal{F}[S])$.

PROOF. The proof is by induction on the structure of the program S.

—Let $S = \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2$. The induction hypothesis is that $\text{WCET}(S_1) = \text{WCET}(\mathcal{F}[S_1]) = \delta_1$ and $\text{WCET}(S_2) = \text{WCET}(\mathcal{F}[S_2]) = \delta_2$. According to Rule 1 and Table I, we thus have:

$$\begin{aligned}\text{WCET}(\mathcal{F}[S]) &= 1 + \max(\delta_1 + \max(0, \delta_2 - \delta_1), \delta_2 + \max(0, \delta_1 - \delta_2))\\ &= 1 + \max(\max(\delta_1, \delta_1 + \delta_2 - \delta_1), \max(\delta_2, \delta_2 + \delta_1 - \delta_2))\\ &= 1 + \max(\max(\delta_1, \delta_2), \max(\delta_2, \delta_1))\\ &= 1 + \max(\delta_1, \delta_2)\end{aligned}$$

According to Table I, we also have $\text{WCET}(S) = 1 + \max(\text{WCET}(S_1), \text{WCET}(S_2))$. By induction hypothesis, this is equal to $1 + \max(\delta_1, \delta_2)$, that is, $\text{WCET}(\mathcal{F}[S])$.

—Let $S = \texttt{for}\ i = n_1\ \texttt{to}\ n_2\ \texttt{do}\ S_1$. The induction hypothesis is that $\text{WCET}(S_1) = \text{WCET}(\mathcal{F}[S_1])$. According to Rule 2 and Table I, we thus have $\text{WCET}(\mathcal{F}[S]) = (n_2 - n_1 + 1) \times (3 + \text{WCET}(\mathcal{F}[S_1]))$. Moreover, according to Table I, we also have $\text{WCET}(S) = (n_2 - n_1 + 1) \times (3 + \text{WCET}(S_1))$. By induction hypothesis, this is equal to $\text{WCET}(\mathcal{F}[S])$.

—Let $S = S1; S2$. The induction hypothesis is that $\text{WCET}(S_1) = \text{WCET}(\mathcal{F}[S_1])$ and $\text{WCET}(S_2) = \text{WCET}(\mathcal{F}[S_2])$. According to Rule 3 and Table I, we thus have $\text{WCET}(\mathcal{F}[S]) = \text{WCET}(\mathcal{F}[S_1]) + \text{WCET}(\mathcal{F}[S_2])$. Moreover, according to Table I, we also have $\text{WCET}(S) = \text{WCET}(S_1) + \text{WCET}(S_2)$. By induction hypothesis, this is equal to $\text{WCET}(\mathcal{F}[S])$.

Thus, we conclude that for any S, $\text{WCET}(S) = \text{WCET}(\mathcal{F}[S])$. □

The transformation $\mathcal{F}$ applied on example *Fac* produces the new program Fac_1:

```
Fac_1 = F[Fac] = read(i);
                 if i > 10 then i := 10; o := 1; else o := 1; skip^3;
                 for l = 1 to 10 do
                      if l <= i then o := o * l; else skip^3;
                 write(o);
```

4.2 Checkpointing and Heartbeating

Checkpointing and heartbeating both involve the insertion of special commands at appropriate program points. The special commands we insert are:

—`hbeat` sends a heartbeat telling the monitor that the processor is alive. This command is implemented by setting a special variable in the stable memory. The vector HBT$[1 \ldots n]$ gathers the heartbeat variables of the n tasks. The command `hbeat` in task i is thus implemented as HBT$[i] := 1$. The failure detection will be presented in detail in Section 5. Informally, the monitor periodically decrements and checks each HBT[i] variables at the same period at which they are set to 1. A failure is detected as soon as HBT[i] reaches −2 (and not 0, to account for the eventual clock drifts).

—`checkpt` saves the current state in the stable memory. It is sufficient to save only the live variables and only those that have been modified since the last checkpoint. This information can be inferred by static analysis techniques. Here, we simply assume that `checkpt` saves enough variables to revert to a valid state when needed.

Heartbeating is usually done periodically, whereas the policies for checkpointing differ. Here, we chose periodic heartbeats and checkpoints. In our context, the key property is to meet the real-time constraints. We will see in Section 5 how to compute the *optimal* periods for those two commands, optimality being defined *w.r.t.* those real-time constraints.

In this section, we define a transformation $\mathcal{I}_c^T(S,t)$ that inserts the command c every T units of time in the program S. It will be used both for checkpointing and heartbeating. The parameter T denotes the period whereas the time counter t counts the time residual before the next insertion. Because the WCET of the "most expensive" atomic statement of our language is 3 and not 1 (e.g., WCET(`read`) = 3), it is not, in general, possible to insert the command c exactly every T time units. However, we will establish a *bound* on the maximal delay between any two successive commands c inserted in S.

The transformation $\mathcal{I}$ relies on the property that all paths of the program have the same execution time (see Property 1 in Section 4.1). In order to insert heartbeats afterward, this property should remain valid after the insertion of checkpoints. We may either assume that `checkpt` takes the same time when inserted in different paths (e.g., the two branches of a conditional), or reapply the transformation $\mathcal{F}$ after checkpointing. Again, the rules below must be understood like a case expression in ML.

Transformation Rule 2

1. $\mathcal{I}_c^T(S,t) = c\,;\mathcal{I}_c^T(S, T - \text{EXET}(c) + t)$ *if* $t \leq 0$
2. $\mathcal{I}_c^T(a,t) = a$ *if* a *is atomic*
3. $\mathcal{I}_c^T(S_1;S_2,t) = \mathcal{I}_c^T(S_1,t)\,;\mathcal{I}_c^T(S_2,t_1)$
   with $t_1 = t - \text{EXET}(S_1)$ *if* $\text{EXET}(S_1) < t$
   with $t_1 = T - \text{EXET}(c) - r$ *if* $\text{EXET}(S_1) = t + q(T - \text{EXET}(c)) + r$
   with $q \geq 0 \quad 0 \leq r < T - \text{EXET}(c))$
4. $\mathcal{I}_c^T(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, t) = \texttt{if } b \texttt{ then } \mathcal{I}_c^T(S_1, t-1) \texttt{ else } \mathcal{I}_c^T(S_2, t-1)$
5. $\mathcal{I}_c^T(\texttt{for } l = n_1 \texttt{ to } n_2 \texttt{ do } S, t) = \mathit{Fold}\,(\mathcal{I}_c^T(\mathit{Unfold}\,(\texttt{for } l = n_1 \texttt{ to } n_2 \texttt{ do } S), t))$

Rule 1 inserts the command c when the time counter t is negative or null. This means that c is inserted either at the "right place" or "slightly after," but never "before the right place." The transformation proceeds with the resulting program and the time target for the next insertion is reset to $T - \text{EXET}(c) + t$, that is, it is computed *w.r.t.* the ideal previous insertion point to avoid any drift.

Rule 2 returns atomic commands unchanged. Indeed, an atomic command cannot be split (hence rule 3 does not apply) and since $t > 0$, no insertion must be performed (if $t \leq 0$, then rule 1 applies).

Rule 3 states that the insertion in a sequence $S_1; S_2$ is first done in S_1. The residual time t_1 used for the insertion in S_2 is either $(t - \text{EXET}(S_1))$ if no insertion has been performed inside S_1 or $(T - \text{EXET}(c) - r)$ if r is the time residual remaining after the $q + 1$ insertions inside S_1 (i.e., if $\text{EXET}(S_1) = t + q(T - \text{EXET}(c)) + r$).

Rule 4 states that, for conditional statements, the insertion is performed in both branches. The time of the test and branching is taken into account by decrementing the time residual $(t - 1)$.

Rule 5 applies to loop statements. It unrolls the loop completely (thanks to the *Unfold* operator), performs the insertion in the unrolled resulting program, and then factorizes code by folding code in `for` loops as much as possible (thanks to the *Fold* operator). The *Unfold* operator is defined by the following transformation rule:

Transformation Rule 3

1. $\textit{Unfold}\,(\texttt{for}\ l = n_1\ \texttt{to}\ n_2\ \texttt{do}\ S) = l := n_1; S; l := n_1 + 1; \ldots l := n_2; S$

While the *Fold* operator is based on the following transformation rules:

Transformation Rule 4

1. $l := n; S; l := n + 1; S \quad \leftrightharpoons \quad \texttt{for}\ l = n\ \texttt{to}\ n + 1\ \texttt{do}\ S$
2. $(\texttt{for}\ l = n_1\ \texttt{to}\ n_2\ \texttt{do}\ S); l := n_2 + 1; S \quad \leftrightharpoons \quad \texttt{for}\ l = n_1\ \texttt{to}\ n_2 + 1\ \texttt{do}\ S$
3. $l := n_1; S; (\texttt{for}\ l = n_1 + 1\ \texttt{to}\ n_2\ \texttt{do}\ S) \quad \leftrightharpoons \quad \texttt{for}\ l = n_1\ \texttt{to}\ n_2\ \texttt{do}\ S$

In fact, it would be possible to express the transformation $\mathcal{I}$ such that it minimally unrolls loops and does not need folding. However, the transformation rules would be much more complex, and we instead chose a simpler presentation involving the *Fold* operator.

Transformation rules 2 assume that the period T is greater than the execution time of the command c, i.e., $T > \text{EXET}(c)$. Otherwise, the insertion may loop by inserting c within c and so on.

We now give a bound on the time interval between any two successive commands c in the transformed program $\mathcal{I}_c^T(\mathcal{F}(S), T)$:

PROPERTY 3. *In a transformed program $\mathcal{I}_c^T(\mathcal{F}(S), T)$, the actual time interval Δ between the beginning of two successive commands c is such that:*

$$T - \varepsilon \leq \Delta < T + \varepsilon$$

with ε being the EXET *of the most expensive atomic instruction (assignment or test) in the program. Please also note that for the first c inserted in the program, Δ is defined as just the beginning time of c.*

We formalize and prove Property 3 in the appendix.

In order to check the real-time constraints, we must precisely compute the overhead resulting from the transformation $S' = \mathcal{I}_c^T(S,t)$:

$$\text{WCET}(\mathcal{I}_c^T(S,t)) = \text{WCET}(S) + \left\lceil \frac{\text{WCET}(S) - t}{T - \text{WCET}(c)} \right\rceil \times \text{WCET}(c) \tag{2}$$

Indeed, the first c is inserted after t units of time, hence the numerator $\text{WCET}(S){-}t$. Also, each time a c is inserted, the time counter is reset to $T - \text{WCET}(c)$; hence the denominator $T - \text{WCET}(c)$. Finally, if $\text{WCET}(S) - t = n \times (T - \text{WCET}(c)) + r$ with $0 < r < T - \text{WCET}(c)$, then the total number of inserted c is $n+1$, while if $r = 0$, then the total number of inserted c is n since Transformation rules 2 do not insert a last c when the time counter is 0 and at the same time S is terminated; hence the $\lceil . \rceil$ function. Equation (2) is valid only when the denominator is strictly positive, that is, when $T - \text{WCET}(c) > 0$. This is reasonable because, if $T \leq \text{WCET}(c)$, it means that the program S' resulting from the transformation $\mathcal{I}$ performs only c commands and has absolutely no time to perform the computations of the initial program S.

CONDITION 1. *For the transformation $S' = \mathcal{I}_c^T(S,T)$ to be valid, the condition $T - \text{WCET}(c) > 0$ must hold.*

Both checkpointing and heartbeating are performed using the transformation $\mathcal{I}$. First, checkpoints are inserted, and then heartbeats. The period between two checkpoints must take into account the overhead that will be added by heartbeats afterward. By applying equation (2) to the period T'_{CP} for S, we obtain (where we take $\overline{h} = \text{WCET}(\texttt{hbeat})$ for conciseness):

$$T_{CP} = T'_{CP} + \left\lceil \frac{T'_{CP}}{T_{HB} - \overline{h}} \right\rceil \times \overline{h}$$

We are not interested in the exact computation within *one* given period T'_{CP}, but rather in the *average* computation over all the periods T'_{CP}. Therefore, we suppress the $\lceil . \rceil$ function to obtain:

$$\begin{aligned} T_{CP} &= T'_{CP} + \frac{T'_{CP}}{T_{HB} - \overline{h}} \times \overline{h} = T'_{CP} \times \left(1 + \frac{\overline{h}}{T_{HB} - \overline{h}}\right) \\ \iff \quad T'_{CP} &= \frac{T_{CP}}{1 + \frac{\overline{h}}{T_{HB} - \overline{h}}} = \frac{T_{CP} \times (T_{HB} - \overline{h})}{T_{HB}} \end{aligned} \tag{3}$$

According to Condition 1, we must have $T'_{CP} > \overline{c}$, that is:

$$T_{CP}\,(T_{HB} - \overline{h}) > T_{HB}\,\overline{c} \iff T_{CP}\,T_{HB} - T_{CP}\,\overline{h} - T_{HB}\,\overline{c} > 0 \tag{4}$$

Figure 2 illustrates Equation (4). The portion of the plane that satisfies this condition is located *strictly above* the curve. The portion of the plane located above the inner blue rectangle (the dashed area) satisfies the more restrictive, but easier to check, condition $T_{CP} > 2\,\overline{c} \wedge T_{HB} > 2\,\overline{h}$. Intuitively, $T_{CP} = 2\,\overline{c}$ means

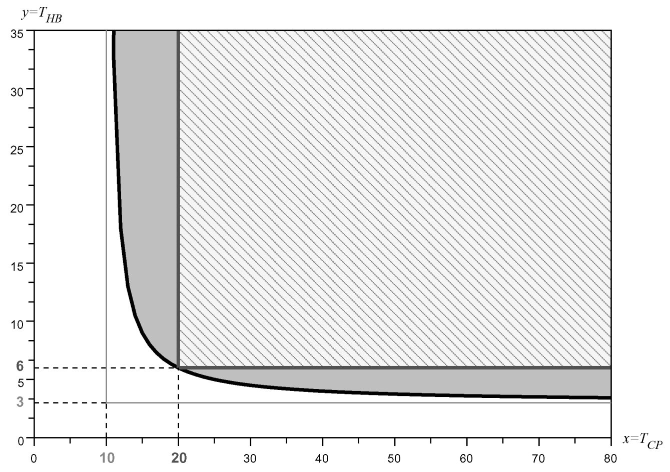


Fig. 2. Curve $T_{CP}\, T_{HB} - T_{CP}\, \overline{h} - T_{HB}\, \overline{c} = 0$ for $\overline{h} = 3$ and $\overline{c} = 10$.

that the transformed program spends 50% of its time performing checkpoints, while $T_{HB} = 2\,\overline{h}$ means that it spend 50% of its time performing heartbeats.

With these notations, the insertion of checkpoints and heartbeats is described by the following ML code:

$$\begin{array}{l} \text{let } (S', -) = \mathcal{I}^{T'_{CP}}_{\texttt{checkpt}}(S, T'_{CP}) \text{ in} \\ \text{let } (S'', -) = \mathcal{I}^{T_{HB}}_{\texttt{hbeat}}(S', 0) \quad \text{ in} \\ \quad S''; \texttt{hbeat}(k) \end{array}$$

The command $\texttt{hbeat}(k)$ is a special heartbeat that sets the variable to k instead of 1, that is, $\text{HBT}[i] := k$. Following this last heartbeat, the monitor will therefore decrease the shared variable and will resume failure detection when the variable becomes 0 again. This mechanism accounts for the idle interval of time between the termination of S'' and the beginning of the next period. Hence, k has to be computed as:

$$k = \left\lceil \frac{T - \text{WCET}(S''; \texttt{hbeat})}{T_{HB}} \right\rceil \tag{5}$$

Figure 3 illustrates the form of a general program (i.e., not Fac_3) after all the transformations.

By applying Equation (2) to the first transformation $S' = \mathcal{I}^{T'_{CP}}_{\texttt{checkpt}}(S, T'_{CP})$ and to the second transformation $S'' = \mathcal{I}^{T_{HB}}_{\texttt{hbeat}}(S', 0)$, we get:

$$\text{WCET}(S') = \text{WCET}(S) + \left\lceil \frac{\text{WCET}(S) - T'_{CP}}{T'_{CP} - \text{WCET}(\texttt{checkpt})} \right\rceil \times \text{WCET}(\texttt{checkpt}) \tag{6}$$

$$\text{WCET}(S'') = \text{WCET}(S') + \left\lceil \frac{\text{WCET}(S')}{T_{HB} - \text{WCET}(\texttt{hbeat})} \right\rceil \times \text{WCET}(\texttt{hbeat}) \tag{7}$$

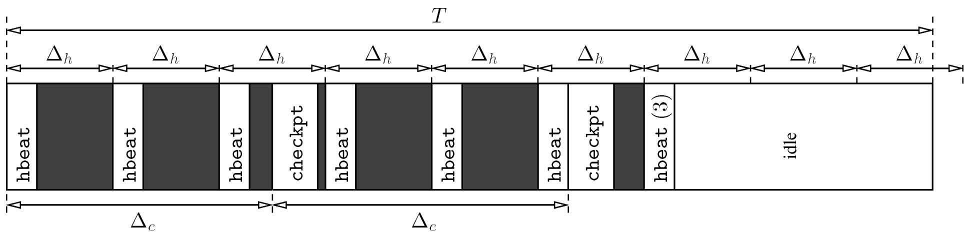


Fig. 3. Program with checkpointing and heartbeating.

After the insertion of heartbeats, the period between checkpoints will be equal to $T'_{CP}\left(1+\frac{\overline{h}}{T_{HB}-\overline{h}}\right)$, i.e., T_{CP}. More precisely, it follows from Property 3 that:

PROPERTY 4. *The actual time intervals Δ_{CP} and Δ_{HB} between two successive checkpoints and heartbeats are such that:*

$$T_{CP}-\varepsilon \le \Delta_{CP} < T_{CP}+\varepsilon+\overline{h} \quad \textit{and} \quad T_{HB}-\varepsilon \le \Delta_{HB} < T_{HB}+\varepsilon$$

PROOF. The proof is based on Property 3. After transformation $\mathcal{I}^{T'_{CP}}_{\texttt{checkpt}}(S, T'_{CP})$, Property 3 gives:

$$T'_{CP}-\varepsilon \le \Delta'_{CP} < T'_{CP}+\varepsilon \tag{8}$$

Assuming the WCET of the most expensive atomic command of `checkpt` is less than or equal to ε, after the second transformation, $\mathcal{I}^{T_{HB}}_{\texttt{hbeat}}(S',0)$, Property 3 satisfies the condition $T_{HB}-\varepsilon \le \Delta_{HB} < T_{HB}+\varepsilon$. The second transformation, however, changes Δ'_{CP} given in Equation (8) to Δ_{CP} such that each portion with the time interval T'_{CP} in the final program will be augmented with $\frac{T'_{CP}}{T_{HB}-\overline{h}}$ `hbeat` commands. Therefore, by following Equation (3), $T'_{CP}+\frac{T'_{CP}}{T_{HB}-\overline{h}}.\overline{h}$ leads to T_{CP}, i.e., the desired value of checkpointing interval. Although we take into account heartbeating in the first transformation, the heartbeating command `hbeat` is invisible to the first transformation. The worst case occurs in the boundary condition of Equation (3) when a heartbeat is inserted just before a checkpoint command. In this case, T_{CP} is shifted upward by $\overline{h}$. In the best case, this shift is zero. Therefore, by shifting up the lower and upper bounds of Δ_{CP} with $[0,\overline{h}]$, we finally derive $T_{CP}-\varepsilon \le \Delta_{CP} < T_{CP}+\varepsilon+\overline{h}$. □

As pointed out above, the transformation $\mathcal{I}$ requires the period to be bigger than the cost of the command. For checkpointing and heartbeating we must ensure that:

$$T_{HB} > \text{WCET}(\texttt{hbeat}) \qquad \text{and} \qquad T'_{CP} > \text{WCET}(\texttt{checkpt})$$

To illustrate these transformations on our previous example, we take:

$$\text{EXET}(\texttt{hbeat})=3 \qquad \text{EXET}(\texttt{checkpt})=10 \qquad T_{CP}=80 \qquad T_{HB}=10$$

Thus, we get $T'_{CP} = 80 - \frac{3 * T'_{CP}}{10-3}$ i.e., $T'_{CP} = 56$ and $\mathcal{I}^{56}_{\texttt{checkpt}}(\mathit{Fac}_1, 56)$ produces:

```
Fac_2 = read(i);
        if $i > 10$ then $i := 10$; $o := 1$; else $o := 1$; skip$^3$;
        for $l = 1$ to 6 do
            if $l <= i$ then $o := o * l$; else skip$^3$;
        $l := 7$; if $l <= i$ then checkpt; $o := o * l$; else checkpt; skip$^3$;
        for $l = 8$ to 10 do
            if $l <= i$ then $o := o * l$; else skip$^3$;
        write($o$);
```

A single checkpt is inserted after 56 time units, which occurs inside the conditional of the 7th iteration of the for loop. The checkpoint is inserted exactly at the desired point in both branches of the conditional. The transformation proceeds by unrolling the loop and inserting checkpt at the right places. Portions of the code are then folded to make two for loops.

For the next step, we suppose, for the sake of the example, that checkpt, which takes 10 units of time, can be split in two parts checkpt = $\texttt{checkpt}_1$;$\texttt{checkpt}_2$ where $\texttt{checkpt}_1$ and $\texttt{checkpt}_2$ take, respectively, 7 and 3 time units exactly. Recall that checkpt is made of the basic instructions of Table I. In other words, the largest WCETof an atomic instruction remains 3 (it would be 10 if checkpt was atomic). We add a heartbeat as a first instruction and, in order to finish with a heartbeat, we must add 5 skip at the end. The transformation $\mathcal{I}^{10}_{\texttt{hbeat}}(\mathit{Fac}_2, 0)$ inserts a heartbeat every 10 time units and yields:

```
Fac_3 = hbeat; read(i);
        if $i > 10$ then $i := 10$; hbeat; $o := 1$; else $o := 1$; hbeat; skip$^3$;
        for $l = 1$ to 6 do
            if $l <= i$ then hbeat; $o := o * l$; else hbeat; skip$^3$;
        $l := 7$;if $l <= i$ then hbeat; checkpt$_1$; hbeat; checkpt$_2$; $o := o * l$;
                      else hbeat; checkpt$_1$; hbeat; checkpt$_2$; skip$^3$;
        for $l = 8$ to 10 do
            hbeat; if $l <= i$ then $o := o * l$; else skip$^3$;
        write($o$); hbeat; skip$^5$; hbeat;
```

Notice that the exact interval between any two successive hbeatis always equal to 10 time units, except at two points:

—Between the hbeat located between $\texttt{checkpt}_1$ and $\texttt{checkpt}_2$, and the hbeat located inside the second for loop, the interval is 12 time units. This is because of the fact that when the transformation $\mathcal{I}$reaches the second for loop, the residual time t is equal to 9; hence the hbeat cannot be inserted right away, the $\mathcal{I}$ transformations enters the for body and the time residual becomes 12. So the hbeat is inserted at the beginning of the for body. To avoid a clock drift, the residual time t at this point is reset to 8 since the hbeat should have been inserted 2 time units earlier. Unfortunately, the next hbeat cannot be inserted after 8 time units, the reason being similar; instead it is inserted after 10 time units.

—Between the last but one `hbeat` and the last `hbeat`, the interval is 8 time units. Indeed, the residual time after inserting the last but one `hbeat` is 8 time units. Since we are at the end of the program and we want to terminate with a `hbeat`, we insert a `skip`[5] to match the desired residual time, which is equal to $8 - \text{EXET}(\texttt{hbeat}) = 8 - 3 = 5$ at the end of the `hbeat`.

In Fac_3, the checkpoint is performed after 83 units of time in both branches, which is inside the $[80, 86)$ interval of Property 4. Finally, since $\text{WCET}(Fac_3) = 143$ and the period is 200, Equation (5) gives $\lceil \frac{200-143}{10} \rceil = 6$, so the last `hbeat` must be changed into `hbeat`(6).

5. IMPLEMENTING THE MONITOR

A special program called *monitor* is executed on the spare processor. As already explained, the monitor performs failure detection by checking the heartbeats sent by each other task. The other responsibility of the monitor is to perform a rollback recovery in case of a failure. In our case, rollback recovery involves restarting the failed task on the spare processor from its latest state stored in the stable memory. In the following subsections, we comprehensively explain heartbeat detection and rollback recovery actions, together with the implementation details and conditions for real-time guarantee.

5.1 Failure Detection

The monitor periodically checks the heartbeat variables HBT[i] to be sure of the liveness of the processor running the tasks τ_i. For a correct operation and fast detection, it must check each HBT[i] at least at the period T_{HB_i}. Since each processor (or each task) has a potentially different heartbeat period, the monitor should concurrently check all the variables at their own speed. A common solution to this problem is to schedule one periodic task for each of the n other processors, whose period is equal to the corresponding heartbeating interval. Therefore, the monitor runs n real-time periodic tasks $\Gamma_i = (Det_i, T_{HB_i})$, with $1 \leq i \leq n$, plus one aperiodic recovery task that will be explained later. The deadline of each task Γ_i is equal to its period T_{HB_i}. The program Det_i is:

```
$Det_i$ = HBT[$i$] := HBT[$i$] − 1;
          if HBT[$i$] = −2 then run $Rec(i)$;
```

When positive, HBT[i] contains the number of T_{HB_i} periods before the next heartbeat of τ_i, hence the next update of HBT[i]. When it is equal to -2, the monitors decides that the processor i is faulty, so it must launch the failure-recovery program *Rec*. When HBT[i] is equal to -1, the processor i is suspected, but not yet declared faulty. Indeed, it might just be late, or HBT[i] might not have been updated yet because of the clock drift between the two processors.

In order to guarantee the real-time constraints, we must compute the worst-case failure detection time α_i for each task τ_i. Since the detector is not synchronized with the tasks, the heartbeat send times $(\sigma_k)_{k\geq 0}$ of τ_i and the heartbeat check times $(\sigma'_k)_{k\geq 0}$ of Det_i may differ in such a way that $\forall k \geq 0, |\sigma_k - \sigma'_k| < T_{HB_i}$. The worst case is when $\sigma_k - \sigma'_k \simeq T_{HB_i}$ and τ_i fails right after sending a heartbeat:

in such a case, the detector receives this heartbeat one period later and starts suspecting the processor i. Hence, it detects its failure at the end of this period. As a result, at worst, the detector program detects a failure after $3 \times T_{HB}$. Remember that the program transformation always guarantees the interval between two consecutive heartbeats to be within $[T_{HB_i}, T_{HB_i} + \varepsilon)$.

Let L_r and L_w denote respectively the times necessary for reading and writing a heartbeat variable, let ξ_i be the maximum time drift between Det_i and τ_i within one heartbeat interval ($\xi_i \ll T_{HB_i}$), the worst-case detection time α_i of the failure of task τ_i then satisfies:

$$\alpha_i < 3(T_{HB_i} + \varepsilon + \xi_i) + L_r + L_w \tag{9}$$

Finally, the problem of the clock drift between the task τ_i that writes HBT[i], and the task Det_i that reads HBT[i], must be addressed. Those two tasks have the same period T_{HB_i}, but since the clocks of the two processors are not synchronized, there are drifts. We assume that these clocks are *quasi-synchronous* [Caspi et al. 1999], meaning that any of the two clocks cannot take the value `true` more than twice between two successive `true` values of the other one. This is the case in many embedded architectures, e.g., TTA and FlexRay for automotive [Rushby 2001]. With this hypothesis, τ_i can write HBT[i] twice in a row, which is not a problem. Similarly, Det_i can read and decrement HBT[i] twice in a row. Again, which is not a problem since Det_i decides that τ_i is faulty only after three successive decrements (i.e., from 1 to -2).

5.2 Rollback Recovery

As soon as the monitor detects a processor failure, it restarts the failed task from the latest checkpoint. This means that the monitor does not exist anymore since the spare processor stops the monitor task and starts executing the failed task instead. The following program represents the recovery operation:

```
Rec (x) = FAILED := x;
          restart (τ_x, CONTEXT_x);
```

where `restart` $(\tau_x, \text{CONTEXT}_x)$ is a macro that stops the monitor application and instead restarts τ_x from its latest checkpoint specified by CONTEXT_x. The shared variable FAILED holds the identification number of the failed task. FAILED $= 0$ indicates that there is no failed processor. FAILED $= x \in \{1, 2, \ldots, n\}$ indicates that τ_x has failed and has been restarted on the spare processor. The recovery time (denoted with β) after a failure occurrence can be defined as the sum of the failure detection time plus the time to reexecute the part of the code after the last checkpoint. If we denote the time for context recovering by L_C, then the worst-case recovery time β is:

$$\beta = 3\left(T_{HB} + \varepsilon + \max_{1 \le i \le n} \xi_i\right) + T_{CP} + L_r + L_w + L_C + \text{WCET}(Det) + \text{WCET}(Rec) \tag{10}$$

5.3 Satisfying the Real-Time Constraints

After the program transformations, the WCET of the fault-tolerant program of the task (S'', T), taking into account the recovery time, is given by Equation (11) below:

$$\text{WCET}(S'') = \text{WCET}(S) + \left\lceil \frac{\text{WCET}(S'')}{T_{HB}} \right\rceil \times \text{WCET}(\texttt{hbeat}) + \left(\left\lceil \frac{\text{WCET}(S'')}{T_{CP}} \right\rceil - 1 \right) \times \text{WCET}(\texttt{checkpt}) \tag{11}$$

where the "-1" accounts for the fact that there is no checkpt either at the very beginning or at the very end of the program, that is, when there is nothing to backup.

Note that this WCET does not include the error detection time and recovery time; for this, we must add the β term computed by Equation (10). We now wish to prove that Equation (11) is consistent with Equations (6) and (7), and with the value of T'_{CP}. For the sake of conciseness, we write $\overline{h}$ for WCET(hbeat), $\overline{c}$ for WCET(checkpt), and $\overline{S}$ for WCET(S). Equation (11), therefore, becomes (note that neglecting the $\lceil . \rceil$ function is for the purpose of a computation averaged over all the periods):

$$\overline{S''} = \overline{S} + \frac{\overline{S''} \cdot \overline{h}}{T_{HB}} + \frac{\overline{S''} \cdot \overline{c}}{T_{CP}} - \overline{c} \iff \overline{S''} \cdot \left(1 - \frac{\overline{h}}{T_{HB}} - \frac{\overline{c}}{T_{CP}} \right) = \overline{S} - \overline{c}$$

$$\iff \overline{S''} = \frac{\overline{S} - \overline{c}}{1 - \frac{\overline{h}}{T_{HB} - \frac{\overline{c}}{T_{CP}}}} = \frac{(\overline{S} - \overline{c})\, T_{CP}\, T_{HB}}{T_{CP}\, T_{HB} - T_{CP}\, \overline{h} - T_{HB}\, \overline{c}} \tag{12}$$

It is interesting to check that, by combining Equations (6) and (7), we will obtain the same expression. Indeed, we first get:

$$\begin{aligned} \overline{S''} &= \overline{S} + \overline{c} \cdot \frac{\overline{S} - T'_{CP}}{T'_{CP} - \overline{c}} + \frac{\overline{h}}{T_{HB} - \overline{h}} \left(\overline{S} + \overline{c} \cdot \frac{\overline{S} - T'_{CP}}{T'_{CP} - \overline{c}} \right) \\ &= \left(\overline{S} + \overline{c} \cdot \frac{\overline{S} - T'_{CP}}{T'_{CP} - \overline{c}} \right) \cdot \frac{T_{HB}}{T_{HB} - \overline{h}} \\ &= \frac{\overline{S}\,(T'_{CP} - \overline{c} + \overline{c}) - \overline{c}\, T'_{CP}}{T'_{CP} - \overline{c}} \cdot \frac{T_{HB}}{T_{HB} - \overline{h}} = \frac{(\overline{S} - \overline{c})\, T'_{CP}}{T'_{CP} - \overline{c}} \cdot \frac{T_{HB}}{T_{HB} - \overline{h}} \end{aligned}$$

By combining with Equation (3), which gives the value of T'_{CP}, we obtain:

$$\overline{S''} = \frac{(\overline{S} - \overline{c})\, T_{CP}}{\frac{T_{CP}\,(T_{HB} - \overline{h})}{T_{HB}} - \overline{c}} = \frac{(\overline{S} - \overline{c})\, T_{CP}\, T_{HB}}{T_{CP}\, T_{HB} - T_{CP}\, \overline{h} - T_{HB}\, \overline{c}} \tag{13}$$

As expected, we can see that Equations (12) and (13) are identical.

Now, one may also be interested in the *optimum* values, $T^{\star}_{CP}$ and $T^{\star}_{HB}$, i.e., the values that offer the best trade-off between fast failure detection, fast failure recovery, and least overhead as a result of the code insertion. If we add the term β of Equation (10) to Equation (13), we obtain a two-value function f of the form:

$$f(x, y) = \frac{(\overline{S} - \overline{c})\, x\, y}{x\, y - x\, \overline{h} - y\, \overline{c}} + x + 3\, y + K \tag{14}$$

where x stands for T_{CP}, y stands for T_{HB}, and K is a constant. This function is defined only when the denominator is strictly positive, that is, when $x\, y - x\, \overline{h} - y\, \overline{c} > 0$. Note that this is exactly Equation (4) resulting from Condition 1.

Since the least overhead resulting from the code insertion means the smallest WCET for S'', we have to minimize f. Now, the computation of its two partial derivatives yields:

$$\begin{aligned}
\frac{\partial f}{\partial x} &= \frac{(\overline{S} - \overline{c})\, y\, (x\, y - x\, \overline{h} - y\, \overline{c}) - (\overline{S} - \overline{c})\, x\, y\, (y - \overline{h})}{(x\, y - x\, \overline{h} - y\, \overline{c})^2} + 1 \\
&= \frac{(\overline{S} - \overline{c})(x y^2 - x y \overline{h} - y^2 \overline{c} - x y^2 + x y \overline{h})}{(x\, y - x\, \overline{h} - y\, \overline{c})^2} + 1 = 1 - \frac{(\overline{S} - \overline{c})\, y^2 \overline{c}}{(x\, y - x\, \overline{h} - y\, \overline{c})^2} \\
\frac{\partial f}{\partial y} &= \frac{(\overline{S} - \overline{c})\, x\, (x\, y - x\, \overline{h} - y\, \overline{c}) - (\overline{S} - \overline{c})\, x\, y\, (x - \overline{h})}{(x\, y - x\, \overline{h} - y\, \overline{c})^2} + 3 \\
&= \frac{(\overline{S} - \overline{c})(x^2 y - x^2 \overline{h} - x y \overline{c} - x^2 y + x y \overline{c})}{(x\, y - x\, \overline{h} - y\, \overline{c})^2} + 3 = 3 - \frac{(\overline{S} - \overline{c})\, x^2 \overline{h}}{(x\, y - x\, \overline{h} - y\, \overline{c})^2}
\end{aligned}$$

From this, we compute the Hessian matrix of f (we skip the details):

$$\nabla^2 f(x, y) = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x y} \\ \frac{\partial^2 f}{\partial y x} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix} = \begin{pmatrix} \frac{2\,(\overline{S} - \overline{c})\, y^2\, \overline{c}\, (y - \overline{h})}{(x\, y - x\, \overline{h} - y\, \overline{c})^3} & \frac{2\,(\overline{S} - \overline{c})\, x\, y\, \overline{c}\, \overline{h}}{(x\, y - x\, \overline{h} - y\, \overline{c})^3} \\ \frac{2\,(\overline{S} - \overline{c})\, x\, y\, \overline{c}\, \overline{h}}{(x\, y - x\, \overline{h} - y\, \overline{c})^3} & \frac{2\,(\overline{S} - \overline{c})\, x^2 \overline{h}\, (x - \overline{c})}{(x\, y - x\, \overline{h} - y\, \overline{c})^3} \end{pmatrix} \tag{15}$$

This matrix is positive definite since its two eigen values are strictly positive whenever the condition $x\, y - x\, \overline{h} - y\, \overline{c} > 0$ holds (again, we skip the details). Hence, the function f if *convex* and it admits a unique minimum $(x^\star, y^\star)$ in the portion of the plane, where $x\, y - x\, \overline{h} - y\, \overline{c} > 0$. The optimal values $x^\star$ and $y^\star$ are those that nullify the two first order partial derivatives. Hence, they are the solutions of Equations (16) and (17) below:

$$\frac{(\overline{S} - \overline{c})\, y^2\, \overline{c}}{(x\, y - x\, \overline{h} - y\, \overline{c})^2} = 1 \iff (\overline{S} - \overline{c})\, y^2\, \overline{c} = (x\, y - x\, \overline{h} - y\, \overline{c})^2 \tag{16}$$

$$\frac{(\overline{S} - \overline{c})\, x^2\, \overline{h}}{(x\, y - x\, \overline{h} - y\, \overline{c})^2} = 3 \iff (\overline{S} - \overline{c})\, x^2\, \overline{h} = 3\, (x\, y - x\, \overline{h} - y\, \overline{c})^2 \tag{17}$$

On the one hand, by combining Equations (16) and (17), we get:

$$(\overline{S} - \overline{c})\, x^2\, \overline{h} - 3\, (\overline{S} - \overline{c})\, y^2\, \overline{c} = 0 \iff x^2 = \frac{3\, y^2\, \overline{c}}{\overline{h}} \iff x = y\, \sqrt{\frac{3\, \overline{c}}{\overline{h}}} \tag{18}$$

On the other hand, Equation (16) can be rewritten as:

$$(\overline{S} - \overline{c})\, y^2\, \overline{c} = (x\, y - x\, \overline{h} - y\, \overline{c})^2 \iff y\, \sqrt{\overline{c}\,(\overline{S} - \overline{c})} = x\, y - x\, \overline{h} - y\, \overline{c} \tag{19}$$

where only the positive square root is kept, since we are only interested in positive solutions. Now, in Equation (19), by replacing x, as given by its expression (18), we get:

$$\begin{aligned} & y\, \sqrt{\overline{c}\,(\overline{S} - \overline{c})} = y^2 \sqrt{\frac{3\,\overline{c}}{\overline{h}}} - y\, \overline{h} \sqrt{\frac{3\,\overline{c}}{\overline{h}}} - y\, \overline{c} \\ \iff\quad & \sqrt{\overline{c}\,(\overline{S} - \overline{c})} = y\, \sqrt{\frac{3\,\overline{c}}{\overline{h}}} - \overline{h} \sqrt{\frac{3\,\overline{c}}{\overline{h}}} - \overline{c} \quad \text{(since } y \neq 0) \\ \iff\quad & y = \overline{h} + \sqrt{\frac{\overline{c}\,\overline{h}}{3}} + \sqrt{\frac{\overline{h}\,(\overline{S} - \overline{c})}{3}} \end{aligned}$$

Finally, by replacing this value of y in Equation (18), we get:

$$x = \overline{c} + \sqrt{3\,\overline{c}\,\overline{h}} + \sqrt{\overline{c}\,(\overline{S} - \overline{c})}$$

In conclusion, the optimal values $T^\star_{CP}$ and $T^\star_{HB}$ that minimize the overhead resulting from the code insertion are:

$$T^\star_{CP} \;=\; \overline{c} + \sqrt{3\,\overline{c}\,\overline{h}} + \sqrt{\overline{c}\,(\overline{S} - \overline{c})} \tag{20}$$

$$T^\star_{HB} \;=\; \overline{h} + \sqrt{\frac{\overline{c}\,\overline{h}}{3}} + \sqrt{\frac{\overline{h}\,(\overline{S} - \overline{c})}{3}} \tag{21}$$

With our *Fac* example, we get $T^\star_{CP} = 46.69$ ms and $T^\star_{HB} = 14.76$ ms. This means that the values we have chosen, respectively, 80 ms and 10 ms, were not the optimal values. Figure 4 is a three-dimension plot of f with the numerical values of the *Fac* example.

Equations (20) and (21) give the optimal values for the heartbeat and checkpoint periods. In order to satisfy the real-time property of the whole system, the only criterion that should be checked is:

$$f(T_{CP_i}, T_{HB_i}) < T_i, \qquad \forall i \in \{1, 2, \ldots, n\} \tag{22}$$

Removing the assumption of zero communication time just involves adding a worst-case communication delay parameter in Equations (9) and (10), which does not have an effect on the optimum values, $T^\star_{CP}$ and $T^\star_{HB}$.

Finally, we give the following property in order for our framework to be complete and sound:

PROPERTY 5. *The real-time distributed system with the specifications drawn in this work can always tolerate one failure and still respect its real-time constraints.*

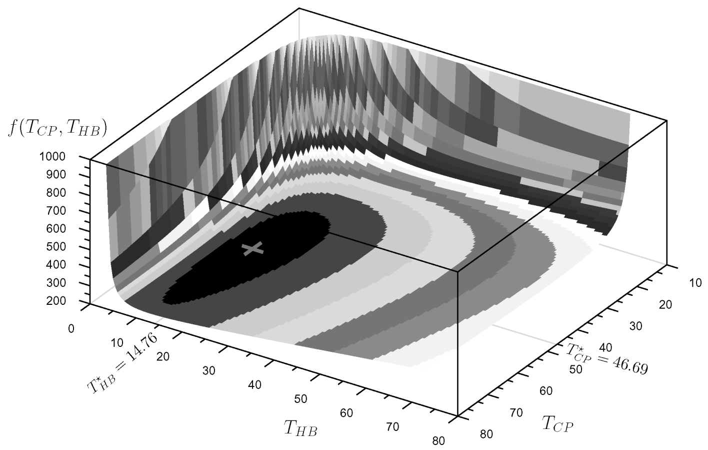


Fig. 4. Three dimensional plot of $f(T_{CP}, T_{HB})$.

PROOF. The recovery time β given in Equation (10) relies on fixed heartbeating and checkpointing intervals (given in Property 4). Therefore, according to Condition (22), there exist T_{CP} and T_{HB} such that the algorithm completes before its deadline against one failure. □

5.4 Scheduling All the Detection Tasks

The monitoring application consists of n detector tasks plus one recovery task. Detector tasks are periodic and independent, whereas the recovery task will be executed exactly once, at the end of the monitoring application (when a failure is detected). Therefore, it can be disregarded in the schedulability analysis. We thus have the task set $\Gamma = \{(Det_1, T_{HB_1}), (Det_2, T_{HB_2}), \ldots, (Det_n, T_{HB_n})\}$ that must satisfy:

$$\forall i \in \{1, 2, \ldots, n\}, \text{WCET}(Det_i) \leq T_{HB_i}. \tag{23}$$

Preemptive scheduling techniques such as rate-monotonic (RM) and earliest-deadline-first (EDF) settle the problem. Both RM and EDF are the major paradigms of preemptive scheduling, and basic schedulability conditions for them were derived by Liu and Layland for a set of n periodic tasks under the assumptions that all tasks start at time $t = 0$, relative deadlines are equal to their periods, and tasks are independent [Liu and Layland 1973]. RM is a fixed-priority based preemptive scheduling, where tasks are assigned priorities inversely proportional to their periods. In EDF, however, priorities are dynamically assigned inversely proportional to each task's distance from its deadline (in other words, as a task gets nearer to its deadline, its priority increases). For many reasons, as remarked in Buttazzo [2005], RM is the most common scheduler implemented in commercial RTOS kernels. In our context, it guarantees that Γ is schedulable if:

$$\sum_{i=1}^{n} \frac{\text{WCET}(Det_i)}{T_{HB_i}} \leq 2(2^{1/n} - 1) \tag{24}$$

Under the same assumptions, EDF guarantees that Γ is schedulable if:

$$\sum_{i=1}^{n} \frac{\text{WCET}(Det_i)}{T_{HB_i}} \leq 1 \tag{25}$$

The above schedulability conditions highlight the fact that both RM and EDF are appropriate and sufficient for scheduling the monitoring tasks with deadline guarantee. EDF allows a better processor utilization, but at the cost of a lot of context switching when processor utilization is close to 1.

6. EXTENSIONS

We propose two extensions to our approach. The first one concerns *transient* failures. The second extension is to tolerate *several* failures at a time.

6.1 Tolerating Transient Failures

Our framework tolerates one *permanent* processor failure. Relaxing this assumption to make the system tolerate one *transient* processor failure (one at a time, of course) implies addressing the following issue. After restarting the failed task on the spare processor, if the failure of the processor is transient, it could likely happen that the failed task also restarts, although probably in an incorrect state. Hence, a problem occurs when the former task updates its outputs since we would have *two tasks* updating the same output in parallel. This problem can be overcome by enforcing a property such that all tasks must check the shared variables FAILED and SPARE so that they can learn the status of the system and take a precaution if they have already been replaced by the monitor. When a task realizes that it has been restarted by the monitor, it must terminate immediately. In this case, since there is no more monitor in the system, the task terminates itself and restarts the monitor application, thus returning the system to its normal state where it can again tolerate one transient processor failure. Note that this would require the architecture to be *homogeneous*: all the processors should be able to replace any other one with equivalent performances.

The following code implements the needed action:

```
$Rem_i$ = if $\text{FAILED} = i$ and $\text{SPARE} \neq \mathit{This\,Processor}$ then
          $\text{SPARE} := \mathit{This\,Processor}$; $\text{FAILED} := 0$; restart_monitor ;
```

where `restart_monitor` is a macro that terminates the task and restarts the monitoring application, and *This Processor* is the ID of the processor executing that code. The shared variable SPAREis initially set to the ID number of the spare processor.

For example, assume that the task i has failed and has been restarted on the spare processor. When the previous code is executed on the spare processor, it will see that even if FAILED is set to i, the task should not be stopped, since it runs on the spare processor. On the other hand, the same task resuming after a transient failure on the faulty processor will detect that it must stop and will restart the monitor task.

At the very least, an Rem_i must be inserted in the program of τ_i just before the output update: $\texttt{write}(o) \Longrightarrow Rem_i; \texttt{write}(o)$. Besides, each $Rem_i; \texttt{write}(o)$ sequence of code must be an atomic transaction. Thus, in order to detect any transient processor failure and to guarantee the real-time constraints, the minimal duration of the transient failure must be larger than the max of all the tasks' periods. If we want to tolerate transient failures with shorter durations, we must insert Rem_i statements at shorter intervals.

6.2 Tolerating Several Failures at a Time

We assumed that the system had one spare processor running a special monitoring program. In fact, additional spare processors could be added to tolerate more processor failures at a time. This does not incur any problem with our proposed approach. The only concern is the implementation of a coordination mechanism between the spare processors, in order to decide which one of them should resume the monitor application after the monitor processor has restarted a failed task τ_i.

7. APPLICATION: THE CYCAB VEHICLE

We illustrate the implementation of our program transformations on the embedded control program of the CyCab autonomous vehicle. This application *does not* exactly fit our theoretical model, the main difference being that it consists of communicating tasks rather than independent tasks. Our goal in this section is precisely to show that our technique can be adapted to such applications, and therefore that the independent tasks assumption can be relaxed. Note, however, that we have not actually made the CyCab fault-tolerant, meaning that we have not modified its hardware architecture. We have just used its control program as a case study.

First, in Section 7.1, we present the CyCab and show how a static schedule is created for its distributed architecture. The program transformations on the CyCab's application are given in Section 7.2. Finally, experimental results with fault injection are presented in Section 7.3.

7.1 Overview of the CYCAB and the AAA Methodology

The CyCab is a vehicle that was designed to transport up to two persons in downtown areas, pedestrian malls, large industrial or amusement parks, and airports, at a maximum speed of 30 km$\cdot$h^{-1} [Baille et al. 1999; Sekhavat and Hermosillo 2000]. It is shown in Figure 5. The mechanics of CyCab is borrowed from a small electrical golf car frame, already produced in small series. The steering is done through an electrical jack mechanically linked to the wheels. Each wheel motor block has its own power amplifier. There are two MPC555 microcontrollers, named F555 and R555, which drive, respectively, the power amplifiers of the two front wheels and the rear wheels. The communications between the nodes are made through a CAN serial bus. The CAN bus has been designed specially for automotive applications and allows safe communications in disturbed environment, with a rate of 1 Mbit$\cdot$s^{-1}. The architecture also includes a PC board that drives the screen and the hard disk.

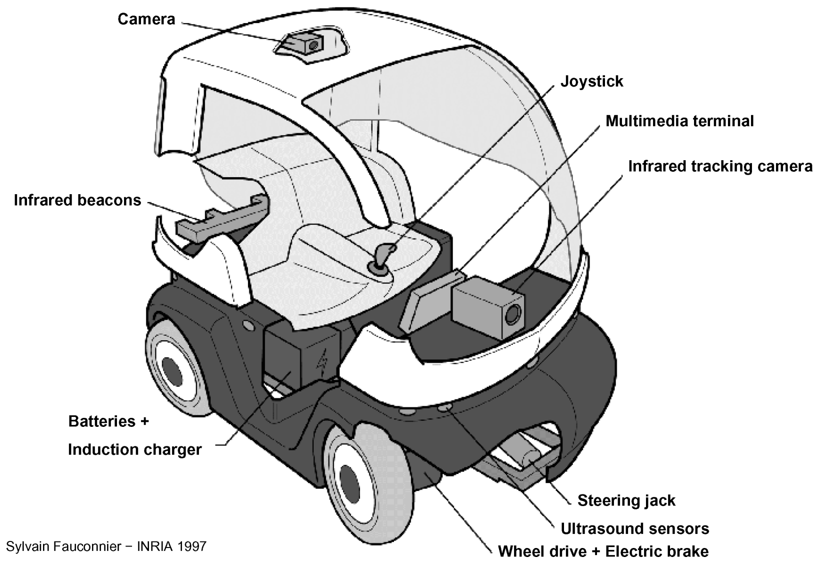


Fig. 5. The CyCab vehicle.

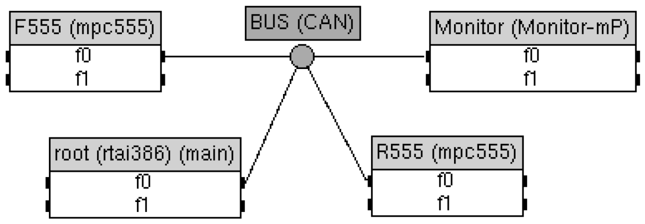


Fig. 6. Architecture graph of the CyCab application.

In the remainder of this article, we call these nodes F555, R555, and ROOT, respectively.

Concretely implementing our program transformations would require one additional node, named MONITOR, connected to the four motor blocks via dual commands in order to be able to control them after the failure of either one of the F555 or R555 processors. The architecture graph of the CyCab is therefore given in Figure 6. Also, neither the MPC555 microcontroller nor the PC board are fail-silent; so guaranteeing that they obey this assumption requires additional hardware: for each processor, this requires the addition of a dual board with self-checking hardware to switch off the output when a failure is detected.

For the present case study, we consider the "manual-driving" application implemented on the CyCab. This application is distributed on the architecture using the SynDEx tool that supports the algorithm architecture adequation methodology (AAA). The goal of this methodology is to find out an optimized implementation of an application algorithm on an architecture, while satisfying distribution constraints. AAA is based on graphs models to exhibit both the potential parallelism of the algorithm and the available parallelism of the

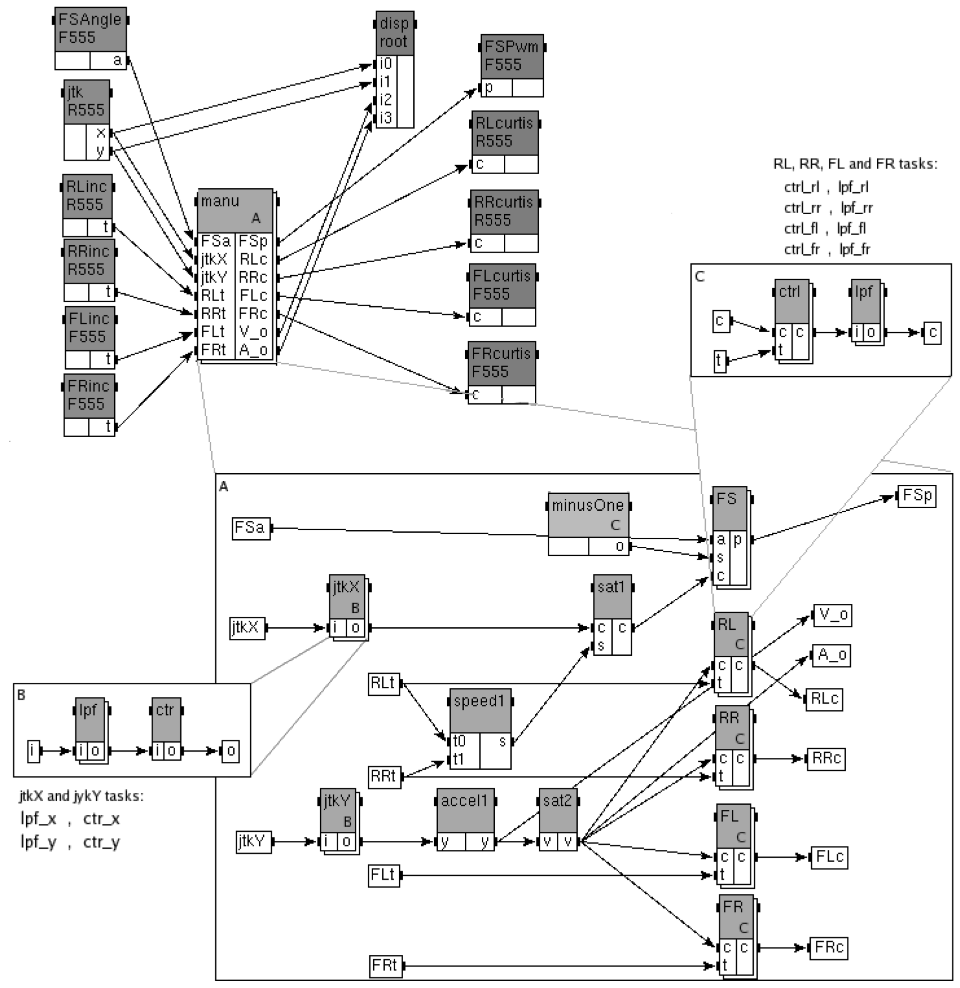


Fig. 7. Application graph of CyCab. A processor name written inside a task indicates a processor constraint, i.e., that task must be scheduled onto that processor.

multicomponent architecture. The implementation is formalized in terms of graphs transformations [Grandpierre et al. 1999; Grandpierre and Sorel 2003]. Concretely, starting from a graph specification of the application and a graph specification of the target architecture, SynDEx first produces a static multiprocessor schedule of the application on the architecture. It then generates the corresponding embeddable code.

Concerning the CyCab manual-driving application, its algorithm graph is given in Figure 7.

Task execution times and communication times are defined and given in Tables II and IV (see later), respectively ("n/a" means that this task cannot be executed onto this processor). We take into account the communication times between all the tasks (including to and from the tasks running on the MONITOR node).

Table II. Task Execution Times (ms) of the CyCab Application Algorithm

Task name	WCET on F555	on R555	on ROOT	on MONITOR
FSAngle, FSPwm	0.3	n/a	n/a	0.3
jtk	n/a	0.3	n/a	0.3
RLinc, RRinc	n/a	0.2	n/a	0.2
FLinc, FRinc	0.2	n/a	n/a	0.2
ctr_x, ctr_y, FS	0.6	0.6	0.6	0.6
lpf_x, lpf_y, speed1 sat2, ctrl_rl, lpf_rl ctrl_rr, lpf_rr, ctrl_fl lpf_fl, ctrl_fr, lpf_fr	0.2	0.2	0.2	0.2
accel1	0.3	0.3	0.3	0.3
sat1	0.3	0.3	0.3	0.3
RLcurtis, RRcurtis	n/a	0.5	n/a	0.5
FLcurtis, FRcurtis	0.5	n/a	n/a	0.5
disp	n/a	n/a	0.5	0.5

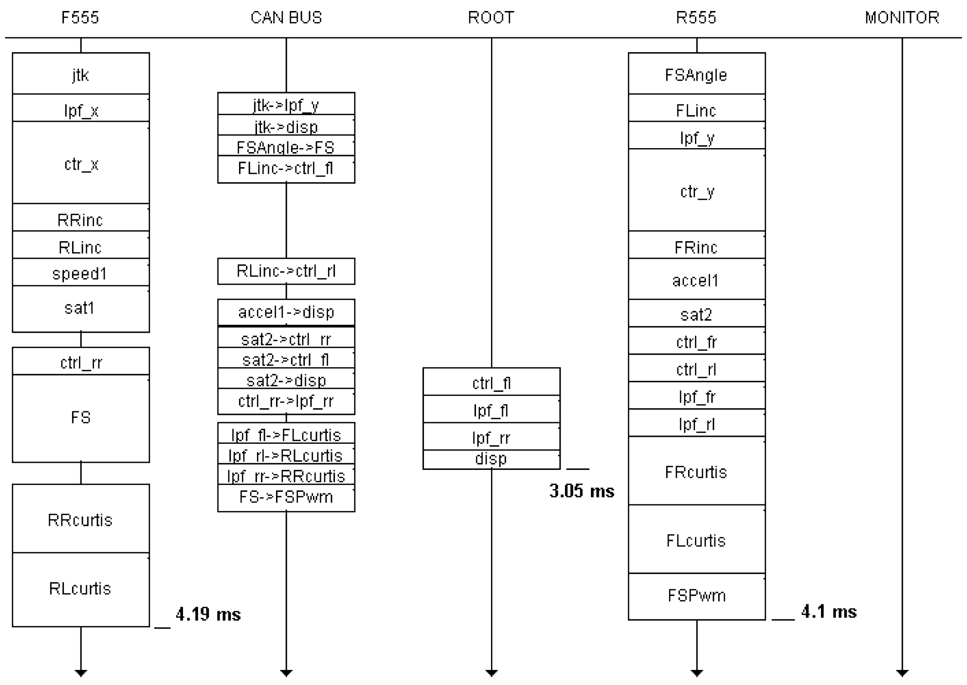


Fig. 8. Static schedule generated by the SynDEx tool (completion time = 4.19 ms).

The AAA algorithm of SynDEx produces the static schedule shown in Figure 8. The real-time constraint is the completion time of the whole algorithm. Let S_1, S_2, and S_3 be the programs of processors F555, ROOT, and R555 respectively. Let $\bar{S_1}$, $\bar{S_2}$, and $\bar{S_3}$ be equal to WCET(S_1), WCET(S_2), and WCET(S_3), respectively. The completion time of the whole algorithm is therefore given by Equation (26) below:

$$\overline{S} = \max(\overline{S_1}, \overline{S_2}, \overline{S_3}) \tag{26}$$

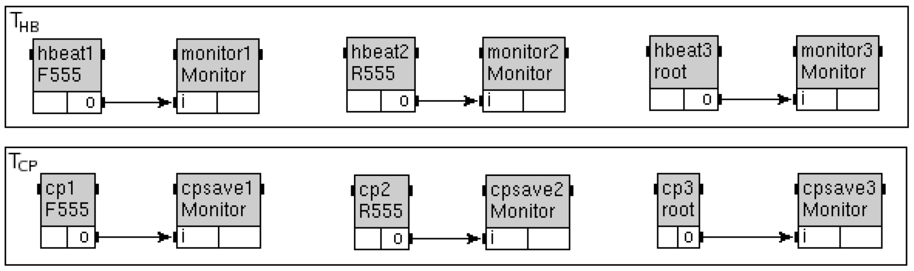


Fig. 9. Application graphs for heartbeating and checkpointing. The two algorithms are executed periodically with the periods T_{HB} and T_{CP} respectively.

According to Figure 8, $\overline{S_1} = 4.19$ ms, $\overline{S_2} = 3.05$ ms, $\overline{S_3} = 4.10$ ms, hence, $\overline{S} = 4.19$ ms. The period of the algorithm, i.e., the deadline, is set to 10 ms in this case study.

7.2 Applying Program Transformations

Our approach is to apply our program transformations on the *static schedules* generated by SynDEx rather than on the *embeddable code* generated by SynDEx.

The heartbeating and checkpointing program transformations periodically insert heartbeating and checkpointing codes at the appropriate places in the static schedule of Figure 8, while generating the monitor application for heartbeat checking and error recovery operations on the MONITOR processor. The graph representation of heartbeat and checkpoint operations is given with Figure 9. We assume that all the tasks are *atomic*, i.e., heartbeat and checkpoint codes cannot be inserted inside the tasks; rather, they are placed *between* the tasks. For example, according to Table II, the execution time of the longest task, ε, is equal to 0.6 ms. In fact, AAA suggests to divide tasks as much as possible to exhibit more potential parallelism (therefore achieving a better schedule length but at the cost of a more expensive heuristics). Hence, this approach simplifies the transformation while still satisfying the properties. Moreover, the checkpointed data to be stored will be much less since checkpoints are taken only between the tasks, i.e., internal variables of tasks are not included in the checkpoint data.

For proper operation, each processor failure should be detected. Therefore, heartbeating and checkpointing transformations are independently applied to each processor. In order to apply the transformations to a processor, we should fill the idle times between tasks with no-operations. For instance, the program S_2 of the ROOT processor is as follows:

$S_2 =$ *idle time*; ctrl_fl; lpf_fl; lpf_rr; disp;

Even though all idle times are filled with no-operations before insertion, task dependency may cause new idle times after placing a checkpoint or heartbeat, since an insertion slightly changes the static schedule. Hence, after each insertion, the resulting static schedule is checked once more and all idle times are filled again before continuing with the next insertion.

Before applying our transformations, we must also calculate the optimal heartbeating and checkpointing periods by modifying the computations presented in the previous sections. First, the worst case error detection time and the recovery time given with Equations (9) and (10) can be expressed by Equations (27) and (28) below:

$$\alpha_i < T_{HB_i} + \varepsilon + \xi_i + L_r + L_w \tag{27}$$

$$\beta = T_{HB} + \varepsilon + \max_{1\leq i\leq n} \xi_i + T_{CP} + L_r + L_w + L_C + \text{WCET}(Det) + \text{WCET}(Rec) \tag{28}$$

where n is the number of processors. The reason why the "3" factor in Equation (9) has been removed in Equation (27) is that the tasks $\texttt{monitor}_i$ are not scheduled anymore with a rate monotonic policy (implying a complete lack of synchronization with the tasks $\texttt{hbeat}_i$), but, instead, are scheduled statically by SynDEx thanks to the data-dependencies expressed in the application graphs of Figure 9 (implying a synchronization between each task $\texttt{hbeat}_i$ and its corresponding task $\texttt{monitor}_i$). The reasoning is the same between Equations (10) and (28).

Checkpoint and heartbeat transformations are applied to the processor's programs S_1, S_2, and S_3 independently. The following ML code illustrates how we apply the program transformations to the whole application.

$$\begin{array}{lll} \text{let } (S_i', -) = \mathcal{I}^{T'_{CP}}_{\texttt{checkpt}}(S_i, T'_{CP}) & \text{in} & \\ \text{let } (S_i'', -) = \mathcal{I}^{T_{HB}}_{\texttt{hbeat}}(S_i', 0) & \text{in} & \\ \quad S_i''; \texttt{hbeat}(k_i) & & \forall i \in \{1,2,3\} \end{array}$$

where

$$k_i = \left\lceil \frac{T - \text{WCET}(S_i'';\texttt{hbeat})}{T_{HB}} \right\rceil$$

Note that the timing analysis presented here does not use any knowledge of the initial static schedule and assumes the worst case, i.e., all processors and communication buses are fully-utilized. Fully-utilized here means that programs S_1, S_2, and S_3 do not have idle times between their tasks. The communication bus has also no idle time. Normally, as can be seen in Figure 8, idle times appear between tasks and between messages, because of the data dependency. These idle times might be filled with `hbeat` and `checkpt` tasks and their communications. If there is no idle time, then each insertion of `checkpt` (resp. `hbeat`) increases the completion time by $\overline{c} + \Delta_{\texttt{checkpt}}$ (resp. $\overline{h} + \Delta_{\texttt{hbeat}}$), at worst, because of the data dependency.

Therefore, the maximum value of the completion time of the algorithm in the presence of one failure can be computed as follows:

$$\overline{S'}_{max} = \overline{S} + \sum_{i=1}^{3} \frac{\overline{S}_i - T'_{CP}}{T'_{CP} - \overline{c}} \times (\overline{c} + \Delta_{\texttt{checkpt}}) \tag{29}$$

$$\overline{S''}_{max} = \overline{S'} + \sum_{i=1}^{3} \left[\overline{S}_i + \frac{\overline{S}_i - T'_{CP}}{T'_{CP} - \overline{c}} \times (\overline{c} + \Delta_{\texttt{checkpt}}) \right] \frac{\overline{h} + \Delta_{\texttt{hbeat}}}{T_{HB} - \overline{h}} \tag{30}$$

The worst-case recovery time β of Equation (10) can be rewritten for our application as follows:

$$\beta = T_{HB} + \varepsilon + \max_{1 \le i \le n} \xi_i + T_{CP} + L_r + L_w + L_C + \textsc{wcet}(Det) + \textsc{wcet}(Rec) \tag{31}$$

When we add the term β to Equation (30), we obtain the completion time of the application algorithm as a two-value function f of the form:

$$\begin{aligned} f(x, y) &= \overline{S''}_{max} + \beta \\ &= \overline{S} + (\overline{S}_1 + \overline{S}_2 + \overline{S}_3) \frac{\overline{h} + \Delta_{\texttt{hbeat}}}{y - \overline{h}} + x + y \\ &\quad + \frac{(\overline{S}_1 + \overline{S}_2 + \overline{S}_3) y - 3xy + 3x\overline{h}}{xy - x\overline{h} - y\overline{c}} \times \frac{(\overline{c} + \Delta_{\texttt{checkpt}})(y + \Delta_{\texttt{hbeat}})}{y - \overline{h}} + K' \end{aligned} \tag{32}$$

where x stands for T_{CP}, y stands for T_{HB} and K' is a constant, i.e., $\beta - x - y$. Taking into account the execution times of *Det* and *Rec*, we find that $K' \simeq 0.2$ ms.

Similarly, f is the WCET that may occur only if the initial schedule given in Figure 8 has fully utilized the processors and communication buses. The analysis considers the worst case and it holds for any given schedule. Generally, and as in our case seen in Figure 8, processors and communication buses will have idle times that might be filled by `hbeat` tasks, `checkpt` tasks, and their communications. Therefore, the actual completion time is expected to be less than the one given in Equation (32). In critical conditions, the analysis can be relaxed by taking into account the static schedule so that the completion time can be calculated precisely to check whether the deadline is met.

The computation of the two partial derivatives of $f(x, y)$ yields:

$$\begin{aligned} \frac{\partial f}{\partial x} &= \frac{1}{(\overline{h}x + (\overline{c} - x)y)^2} (\overline{h}^2 x^2 + 2\overline{h}(\overline{c} - x)xy + y(y(4\overline{c}^2 - 2\overline{c}x + x^2 + 3\overline{c}\Delta_{\texttt{checkpt}}) \\ &\quad + 3\overline{c}(\overline{c} + \Delta_{\texttt{checkpt}})\Delta_{\texttt{hbeat}}) - y(\overline{c} + \Delta_{\texttt{checkpt}})(y + \Delta_{\texttt{hbeat}})(\overline{S_1} + \overline{S_2} + \overline{S_3})) \\ \frac{\partial f}{\partial y} &= \frac{1}{(\overline{h} - y)^2(\overline{h}x + (\overline{c} - x)y)^2} ((\overline{h} - y)^2(\overline{c}^2(y^2 - 3x\Delta_{\texttt{hbeat}}) \\ &\quad + \overline{c}x(3\overline{h}x + 2\overline{h}y - 2y^2 + 3x\Delta_{\texttt{hbeat}} - 3\Delta_{\texttt{checkpt}}\Delta_{\texttt{hbeat}}) \\ &\quad + x^2(\overline{h}^2 - 2\overline{h}y + y^2 + 3\overline{h}\Delta_{\texttt{checkpt}} + 3\Delta_{\texttt{checkpt}}\Delta_{\texttt{hbeat}})) \\ &\quad - (\overline{h}x^2(\overline{h} - y)^2 - \overline{h}y(2\overline{h}x + (\overline{c} - 2x)y)\Delta_{\texttt{checkpt}} \\ &\quad - ((\overline{c} - x)x(\overline{h} - y)^2 + (\overline{h}^2 x + (\overline{c} - x)y^2)\Delta_{\texttt{checkpt}})\Delta_{\texttt{hbeat}})(S_1 + S_2 + S_3)) \end{aligned}$$

As in Section 5.3, we can compute the Hessian matrix of f and show that this matrix is positive definite since its two eigen values are strictly positive whenever the condition $xy - x\,\overline{h} - y\,\overline{c} > 0$ holds (again, we skip the details). Hence, the function f if *convex* and it admits a unique minimum $(x^\star, y^\star)$ in the portion of the plane where $x\,y - x\,\overline{h} - y\,\overline{c} > 0$. The optimal values $x^\star$ and $y^\star$ are those that nullify the two first order partial derivatives.

Table III. Task Execution Times (ms) of Heartbeating and Checkpointing

Task name	WCET on F555	on R555	on ROOT	on MONITOR
hbeat1, cp1	0.06	n/a	n/a	0.06
hbeat2, cp2	n/a	0.06	n/a	0.06
hbeat3, cp3	n/a	n/a	0.06	0.06
monitor1, monitor2, monitor3 cpsave1, cpsave2, cpsave3	n/a	n/a	n/a	0.06

Table IV. Communication Times

Communication	Duration (ms)
`hbeat` → `monitor`	$\Delta_{\texttt{hbeat}} = 0.12$
`checkpt` → `cpsave`	$\Delta_{\texttt{checkpt}} = 0.15$
Other messages	$\Delta = 0.15$

Taking into account the values given in Tables III and IV, we find the optimal values as follows:

$$T^{\star}_{CP} = 1.71348 \text{ ms}$$
$$T^{\star}_{HB} = 1.57404 \text{ ms}$$

Recalling Property 5 and Condition (22), it can be proved that the transformed program always meets its deadline, even in the presence of one failure:

$$\begin{aligned} & f(x^\star, y^\star) = \overline{S''}_{max} + y^\star + y^\star + K' && < T \\ \Longleftrightarrow\ & 6.48439 + 1.71348 + 1.57404 + 0.2 && < 10 \\ \Longleftrightarrow\ & 9.97191 \text{ ms} && < 10 \text{ ms} \end{aligned}$$

7.3 Results and Discussion

If we apply the transformations to insert `hbeat` and `checkpt` tasks with the periods of $T^{\star}_{HB}$ and $T^{\star}_{CP}$ respectively, we obtain the schedule given in Figure 10. For instance, the ROOT processor will have the following task sequence after our transformations:

`hbeat`; nop^{30}; `hbeat`; nop^{2}; `checkpt`; nop^{23} ; ctrl_fl; `hbeat`; lpf_fl; nop; `checkpt`; nop; lpf_rr; disp; `hbeat`;

In failure-free operation, the completion time of the new algorithm is 5.60 ms as shown in Figure 10. The overhead of the fault-tolerance properties is, therefore, $5.60 - 4.19 = 1.41$ ms.

Thanks to Equation (32), we prove that the deadline is always met in spite of one processor failure. Figure 11, on the other hand, illustrates how the failure detection and recovery operations are handled in one iteration of the algorithm.

For a correct failure recovery, two kinds of messages must be dealt with, the messages sent to the faulty processor and the messages sent by the faulty processor:

—Following a failure, the MONITOR rolls back to the last checkpoint and restarts the schedule of the faulty processor. To do this, it needs the data

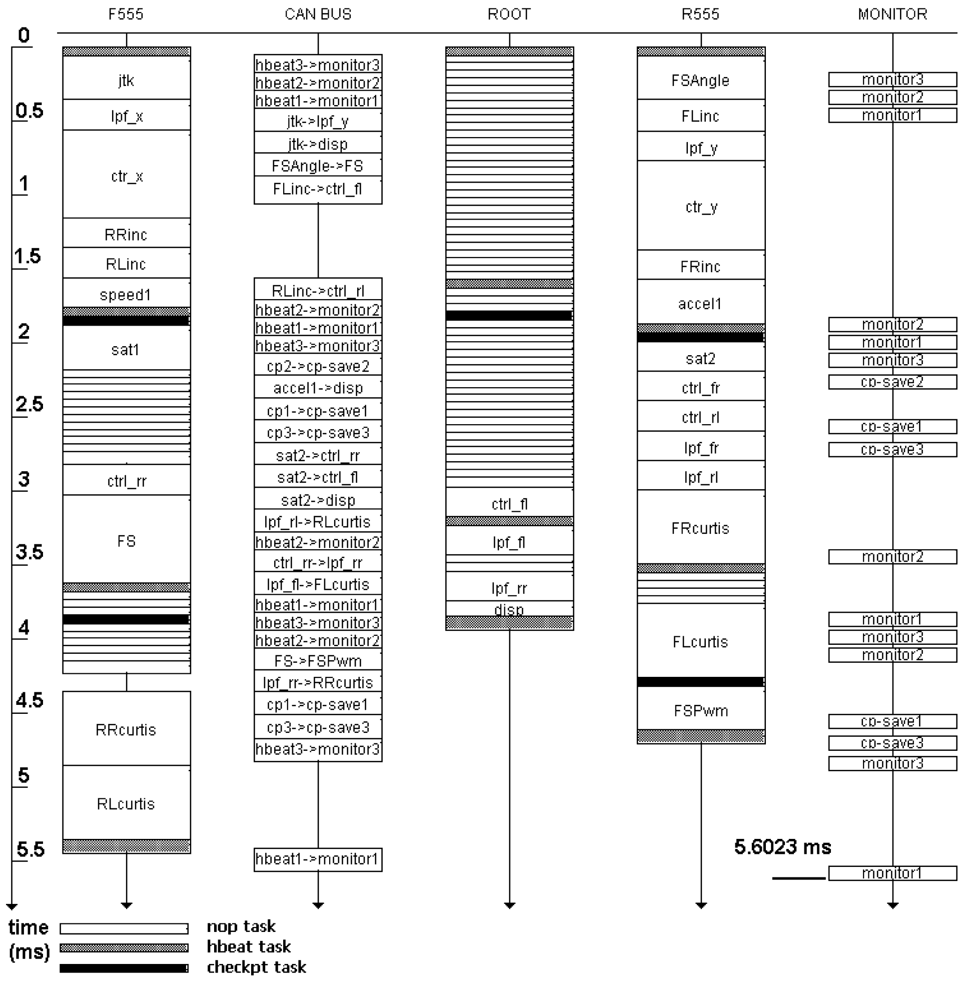


Fig. 10. Fault-tolerant static schedule with heartbeating and checkpointing (the completion time is 5.6023 ms).

sent by all the processors to the faulty one. Therefore, all the messages flying on the bus must be stored in the stable memory.

—A task blocked because it is waiting for some data that was supposed to be sent by the faulty processor, just needs to wait for the MONITOR to roll back, reexecute the schedule of the faulty processor, and send the awaited data (this is what happens between task lpf_rr on processor ROOT and task ctrl_rr on processor FMPC555 in Figure 11). To guarantee that the monitor task will not mistake the faulty processor and the blocked processor, each task waiting for some communication must periodically execute a `hbeat`.

Finally, we have performed some tests to show the completion time of the transformed program after a failure recovery. Figure 12 shows the completion

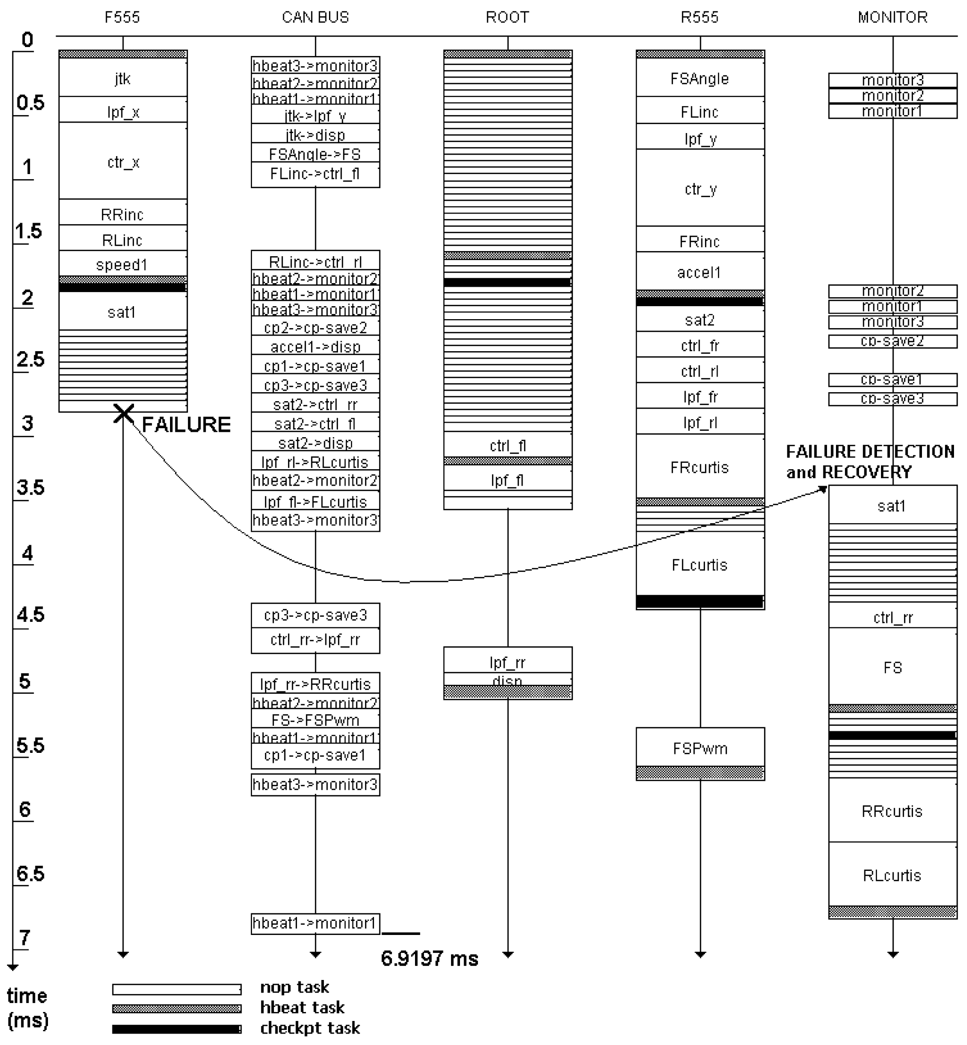


Fig. 11. Example of a recovery when processor F555 fails at time $t = 3$ ms (the completion time is 6.9197 ms).

times of the algorithm for 60 failure instants. For each failure instant, the figure illustrates three completion times for the failure of the three processors. Processor failures are injected by software at relative failure times that range from 0.1 to 6 ms with 0.1-ms intervals. For example, the completion time will be 8.78 ms if processor F555 fails at the failure instant $t = 3.8$ ms, 7.02 ms if processor ROOT fails at the same instant and 7.99 ms if processor R555 fails at the same instant.

According to this experiment, the maximum completion time is achieved for the failure of processor F555 when the failure instant is around 3.75 ms, that is, just after a heartbeat (it maximizes the detection delay) and just before a checkpoint (it maximizes the roll-back delay); see Figure 10. The monitor will

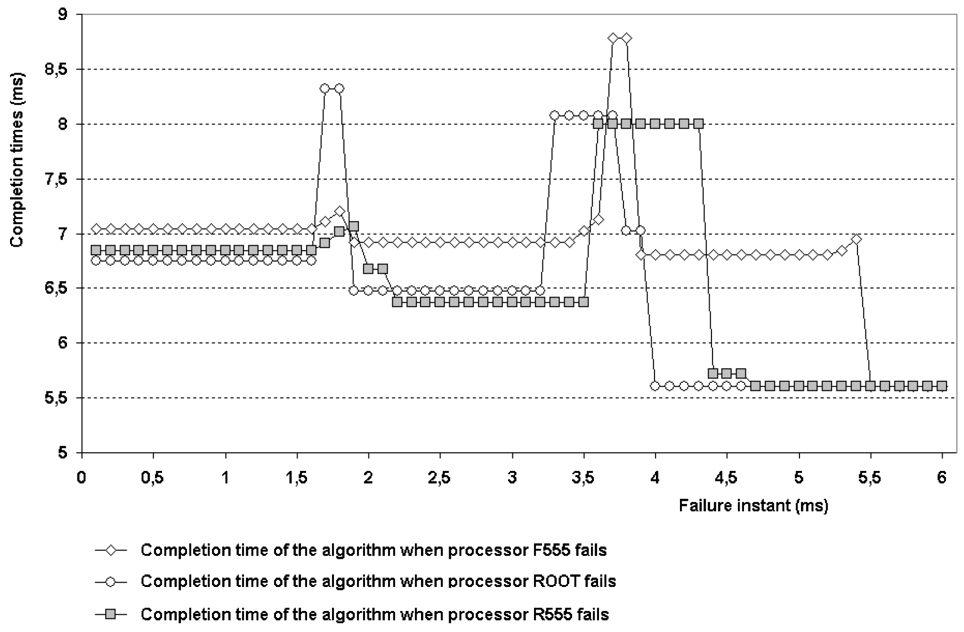


Fig. 12. Completion times when processors F555, ROOT, and R555 fails.

notice the failure approximately T_{HB} ms later and return to the last checkpoint that occurred approximately T_{CP} ms before the failure instant. Thus, the completion time will be approximately $T_{HB} + T_{CP}$ greater than the failure free completion time.

8. RELATED WORK

Related work on failure detectors is abundant. On the theoretical side, Fisher et al. [1985] have demonstrated that, in an asynchronous distributed system (i.e., no global clock, no knowledge of the relative speeds of the processes or the speed of the communications) with reliable communications (although messages may arrive in another order than they were sent), if one single process can fail permanently, then there is no algorithm that can guarantee consensus on a binary value in finite time. Indeed, it is impossible to tell if a process has died or if it is just very slow in sending its message. If this delayed process's input is necessary, say, to break an even vote, then the algorithm may be delayed indefinitely. Hence, no form of fault-tolerance can be implemented in totally asynchronous systems. Usually, one assumption is relaxed, for instance an upper bound on the communication time is known, and this is exactly what we do in this paper to design our failure detector. Then, Chandra and Toueg [1996] formalized unreliable failure detectors in terms of completeness and accuracy. In particular, they have shown what properties are required to reach consensus in the presence of crash failures. On the practical side, Aggarwal and Gupta [2002] present in a short survey on failure detectors. They explain the push and pull methods in detail and introduce QoS techniques to enhance the performance of failure detectors.

Our program transformations are related to software thread integration (STI). STI involves weaving a host secondary thread inside a real-time primary thread by filling the idle time of the primary thread with portions of the secondary thread [Dean and Shen 1998]. Compared to STI, our approach formalizes the program transformations and also guarantees that the real-time constraints of the secondary thread will be preserved by the obtained thread (and not only those of the primary thread).

Other work on program transformation for fault-tolerance has been conducted by Liu and Joseph [1992]. A program is modeled as a sequence of atomic actions, each action being a transition from the program's internal state to its next state. The union composition is used to model choice. The authors use a simple specification language with assignment, sequential composition, conditional composition (`if then else` with multiple clauses), and iterative composition (`do while`). Each action is assumed to terminate. They also define the semantics of this language. Failures are then formally specified as additional actions putting the program into an error state. The failures considered here are hardware fail-stop, and are assumed not to affect recovery actions. The error state is identified by a special variable f, which is true only in this state (it is assumed that the initial program never modifies f). This assumption eliminates the problem of failure detection. The authors then add a set of recovery actions to put back the error state of a faulty program in a good state. Backward and forward recovery actions are two special cases of such recovery actions. The authors then show how to insert checkpointing and recovery actions thanks to program refinement (a particular case of program transformation). Although they present a sound theoretical framework, they do not specifically deal with concrete fault-tolerance techniques to achieve fault-tolerance. Also, their assumption concerning failure detection eliminates the need for a specific program transformation to obtain this; in contrast, we treat this specifically with heartbeating. Finally, a crucial distinction with our own work is that they do not address real-time properties.

Other works on failure recovery include the efforts of reserving sufficient slack in dynamic schedule, i.e., gaps between tasks because of the precedence, resources, or timing constraints, so that the scheduler can reexecute faulty tasks without jeopardizing the deadline guarantees [Mossé et al. 2003]. Further studies proposed different heuristics for reexecution of faulty tasks in imprecise computation models such that faulty mandatory sub-tasks may supersede optional subtasks [Aydin et al. 2000]. In contrast, our work is entirely in the static scheduling context.

Other related work on automatic transformations for fault-tolerance include the work of Kulkarni and Arora [Kulkarni and Arora 2000]. It involves synthesizing a fault-tolerant program starting from a fault-intolerant program. A program is a set of states (valuations of the program's variables) and a set of transitions between states. A fault is a set of transitions. Two execution models are considered: high atomicity (the program can read/write any number of its variables in one atomic step, i.e., it can make a transition from any one state to any other state) and low atomicity (it can't). The initial fault-intolerant program ensures that its specification is satisfied in the absence of faults, although no guarantees are provided in the presence of faults. Three levels of fault-tolerance are studied: failsafe ft (in the presence of faults, the synthesized program guarantees safety), non-masking ft (in the presence of faults, the synthesized program recovers to states from where its safety and liveness are satisfied), and masking ft (in the presence of faults the synthesized program satisfies safety and recovers to states from where its safety and liveness are satisfied). Thus six algorithms are provided. In the high atomicity model (resp. low), the authors propose a sound algorithm that is polynomial (resp. exponential) in the state space of the initial fault-intolerant program. In the low atomicity model, the transformation problem is NP-complete. Each transformation involves recursively removing bad transitions. This principle of program transformation implies that the initial fault-intolerant program should be maximal (weakest invariant and maximal nondeterminism). In conclusion, Kulkarni et al. offer a comprehensive formal framework to study fault-tolerance. Our own work could be partially represented in terms of their model, since our programming language can be easily converted to the finite-state automaton consisting of a set of states and transitions. Moreover, our study complies well with their detector–corrector theory presented thoroughly in Arora and Kulkarni [1998]. However, we deal explicitly with the temporal relationships in the automatic addition of fault-tolerance by using heartbeating and checkpointing/rollback as a specific detector-corrector pair. Therefore, defining and implementing our system in terms of Kulkarni's model might require much effort and be of interest for future research.

Finally, discrete controller synthesis [Ramadge and Wonham 1987] has been successfully applied to derive automatic program transformation methods for fault-tolerance [Dumitrescu et al. 2004; Girault and Rutten 2004; Girault and Yu 2006; Dumitrescu et al. 2007]. The principle is similar to the work of Kulkarni and Arora, except that the set of events labeling the transitions is partitioned into the two subsets of controllable and uncontrollable events, faults being uncontrollable. Besides, a synthesis objective is given by the user, usually in terms of invariant or reachable states sets. Discrete controller synthesis then involves traversing exhaustively the state space of the system (with symbolic algorithms) to build a controller that will steer the system in such a way that it satisfies its synthesis objective whatever be the uncontrollable events. This approach is thus richer (thanks to the uncontrollability of events) and more flexible (thanks to the synchronous product used to specify the labeled transition system of the initial fault-intolerant system) than the work of Kulkarni and Arora.

9. CONCLUSION

In this article, we have presented a formal approach to fault-tolerance. Our fault-intolerant real-time application consists of periodic, independent tasks that are distributed onto processors showing omission/crash failure behavior, and of one spare processor for the hardware redundancy necessary to fault-tolerance. We derived program transformations that automatically convert the programs such that the resulting system is capable of tolerating one permanent or transient processor failure at a time. Fault-tolerance is achieved by heartbeating and checkpointing/rollback mechanisms. Heartbeats and checkpoints are thus inserted automatically, which yields the advantage of being transparent to the developer, and on a periodic basis, which yields the advantage of relatively simple verification of the real-time constraints. Moreover, we choose the heartbeating and checkpointing periods such that the overhead because of adding the fault-tolerance is minimized. We also proposed mechanisms to schedule all the detection tasks onto the spare processor, in such a way that the detection period is, at worst, three times the heartbeat period. To the best of our knowledge, the two main contributions presented in this article (i.e., the formalization of adding fault-tolerance with automatic program transformations, and the computation of the optimal checkpointing and heartbeating periods to minimize the fault-tolerance overhead) are novel.

This transparent periodic implementation, however, has no knowledge about the semantics of the application and may yield large overheads. In the future, we plan to overcome this drawback by shifting checkpoint locations within a predefined safe time interval such that the overhead will be minimum. This work can also be extended to the case where processors execute multiple tasks with an appropriate scheduling mechanism. On the other hand, these fundamental fault-tolerance mechanisms can also be followed by other program transformations in order to tolerate different types of errors such as communication, and data upsetting. These transformations are seemingly more user dependent, which may lead to the design of aspect-oriented based tools.

Another area of future research will be to use a proof assistant (e.g., PVS, ACL2, Coq, etc.) to prove that a task is fault-tolerant and meets its deadline even when a failure occurs. The results we have presented in this article are a first step towards this goal.

APPENDIX - FORMALIZATION AND PROOF OF PROPERTY 3

Property 3 ensures that the transformation $\mathcal{I}_c^T(S, T)$ inserts a command c after each T time units (modulo ε). This time interval is intuitively clear but not formalized. The standard approach to formalize and prove Property 3 would be to define a timed semantics of programs (i.e., a semantics where time evolution is explicit) and then to show that the execution of $\mathcal{I}_c^T(S, T)$ involves reducing c each T time units. In order to stick to our program transformation framework, we rather explicit all the execution traces of a program, and we prove by induction on all the possible traces that two successive commands c are always separated by T time units (modulo ε). For this, we define the function *Traces* which associates to each program the set of all its possible executions. An execution is represented as sequences of basic instructions $a_1; \ldots; a_n$. Basically, the *Traces* function unfold loops and replaces conditionals by the two possible executions depending on the test. Formally, it is defined as follows:

Transformation rule 5

$$\begin{array}{lll}
1.\ \mathit{Traces}\,(a) & = \{a\} & \text{if } a \text{ is atomic} \\
2.\ \mathit{Traces}\,(S_1;S_2) & = \{T_1;T_2 \mid T_1 \in \mathit{Traces}\,(S_1),\, T_2 \in \mathit{Traces}\,(S_2)\} & \\
3.\ \mathit{Traces}\,(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2) & = \{\texttt{skip};T \mid T \in \mathit{Traces}\,(S_1) \cup \mathit{Traces}\,(S_2)\} & \\
4.\ \mathit{Traces}\,(\texttt{for } l = n_1 \texttt{ to } n_2 \texttt{ do } S) & = \mathit{Traces}\,(\mathit{Unfold}\,(\texttt{for } l = n_1 \texttt{ to } n_2 \texttt{ do } S)) &
\end{array}$$

The instruction `skip` in rule 3 above represents the time taken by the test, i.e., one time unit. For any initial state, there is always a trace τ in *Traces* (S) representing exactly the execution of S. The important point is that such execution traces τ have a *constant* execution time (i.e., $\text{BCET}(\tau) = \text{WCET}(\tau) = \text{EXET}(\tau)$), and, moreover, we have for any τ:

$$\tau \in \mathit{Traces}\,(S) \Longrightarrow \begin{cases} \text{BCET}\,(S) \leq \text{EXET}\,(\tau) \leq \text{WCET}\,(S) \text{ and} \\ \text{BCET}\,(S) = \text{WCET}\,(S) \Rightarrow \text{EXET}\,(\tau) = \text{EXET}\,(S) \end{cases} \tag{A1}$$

We consider that *Traces* treats c (the command inserted by the transformation $\mathcal{I}$) as an atomic action.

We introduce the equivalence relation $\doteq$ to normalize and compare execution traces. The relation is a syntactic equivalence modulo the associativity of sequencing. It also allows the introduction of the dummy instruction `void`, similar to `skip`, except that $\text{EXET}(\texttt{void}) = 0$. The relation $\doteq$ is such that:

$$(\tau_1;\tau_2);\tau_3 \doteq \tau_1;(\tau_2;\tau_3) \qquad \tau \doteq (\texttt{void};\tau) \doteq (\tau;\texttt{void})$$

We generalize Property 3 to take into account any initial time residual before inserting the first command c:

PROPERTY 6. *Let S, c, t, and T be such that:*

$$\begin{array}{ll}
(0)\ \text{BCET}\,(S) = \text{WCET}\,(S) & (1)\ \text{EXET}\,(c) + \varepsilon < T \\
(2)\ t \leq \text{EXET}\,(S) & (3)\ -\varepsilon < t \leq T
\end{array}$$

Then $\forall \tau \in \mathit{Traces}\,(\mathcal{I}_c^T(S,t))$, $\tau \doteq S_1;c;S_2 \ldots c;S_n \ (1 \leq n)$ *and verifies:*

$$\begin{array}{lll}
t \leq \text{EXET}(S_1) < t + \varepsilon & & (\mathit{Init}) \\
T - \varepsilon < \text{EXET}\,(c;S_i) \leq T + \varepsilon & (1 < i < n) & (\mathit{Period}) \\
r - \varepsilon < \text{EXET}\,(S_n) \leq r & \mathit{If}\ \text{EXET}\,(S) = t + q(T - \text{EXET}\,(c)) + r & (\mathit{End}) \\
 & \mathit{with}\ \ 0 \leq q \text{ and } 0 \leq r < T - \text{EXET}(c) &
\end{array}$$

Property 6 states than any execution trace of the transformed program starts by an execution of t (modulo ε) time units before inserting the first command c. Then, the execution inserts a c every T time units (modulo ε). After the last c, the program takes less than $r < T - \text{EXET}(c)$ unit of times to complete, r being the remaining of the division of $\text{EXET}(S)$ by $(T - \text{EXET}(c))$. This last condition is based on a periodic decomposition of the execution of the source program S. It also ensures that there is no time drift. The property relies on the four following conditions:

0. The program S should have been time equalized beforehand.
1. The period T must be greater than the execution time of the command c plus the execution time of the most expensive atomic action. This condition ensures that it is possible to execute at least one atomic action between two c and therefore the program will make progress.
2. The global execution time must be greater than t (otherwise there is nothing to insert).
3. The time residual t might be negative but no less than ε. Otherwise, it would mean that the ideal point to insert c has been missed by more than ε time units.

PROOF THAT PROPERTY 6 HOLDS FOR POSITIVE TIME RESIDUALS. We prove that Property 6 holds for $0 < t \leq T$, by structural induction on S.

CASE $S = a$: By hypothesis, $0 < t \leq \text{EXET}(a)$, so $\mathcal{I}_c^T(a, t) = a;c$. The only execution trace is $a;c \doteq a;c;\texttt{void}$, which satisfies the property. Indeed:

—By definition of ε, $\text{EXET}(a) \leq \varepsilon$ and, by hypothesis, $0 < t$ and $t \leq \text{EXET}(a)$, therefore:

$$t \leq \text{EXET}(a) < t + \varepsilon \quad (\textit{Init})$$

—From $\text{EXET}(a) = t + r$ with $0 \leq r < \varepsilon$ and $\text{EXET}(\texttt{void}) = 0$, it follows that:

$$r - \varepsilon < \text{EXET}(\texttt{void}) \leq r \quad (\textit{End})$$

CASE $S = S_1;S_2$: There are two sub-cases depending on t.

1. $\text{EXET}(S_1) < t$: Therefore $\mathcal{I}_c^T(S_1;S_2, t) = S_1;\mathcal{I}_c^T(S_2, t - \text{EXET}(S_1))$ because of rules 1 and 4.
   Condition (2) enforces that $t < \text{EXET}(S_1;S_2) = \text{EXET}(S_1) + \text{EXET}(S_2)$, therefore $t - \text{EXET}(S_1) < \text{EXET}(S_2)$. Condition (3) enforces that $t \leq T$ and, by hypothesis, $\text{EXET}(S_1) < t$ therefore $0 < t - \text{EXET}(S_1) \leq T - \text{EXET}(S_1) \leq T$. Hence, S_2 satisfies the induction hypothesis, and $\forall \tau_2 \in \textit{Traces}(\mathcal{I}_c^T(S_2, t - \text{EXET}(S_1)))$, $\tau_2 \doteq S_{2,1};c;S_{2,2} \ldots c;S_{2,n}$ $(1 \leq n)$ and verifies:
   $$\begin{array}{lr} t - \text{EXET}(S_1) \leq \text{EXET}(S_{2,1}) < t - \text{EXET}(S_1) + \varepsilon & (\textit{Init}) \\ T - \varepsilon < \text{EXET}(c;S_{2,i}) \leq T + \varepsilon \ \ (1 < i < n) & (\textit{Period}) \\ r - \varepsilon < \text{EXET}(S_{2,n}) \leq r \quad \text{If } \text{EXET}(S_2) = t - \text{EXET}(S_1) + q(T - \text{EXET}(c)) + r & \\ \qquad\qquad\qquad\qquad \text{with } 0 \leq q \text{ and } 0 \leq r < T - \text{EXET}(c) & (\textit{End}) \end{array}$$
   Any execution trace τ of $\mathcal{I}_c^T(S_1;S_2, t)$ is made of an execution trace τ_1 of $\mathcal{I}_c^T(S_1, t)$ followed by an execution trace τ_2 of $\mathcal{I}_c^T(S_2, t - \text{EXET}(S_1))$. In other words, $\tau \doteq \tau_1;S_{2,1};c;S_{2,2} \ldots c;S_{2,n}$. The property is satisfied if, $t \leq \text{EXET}(\tau_1;S_{2,1}) < t + \varepsilon$, which follows from the fact that the *Traces* function satisfies the Property (33), i.e., $\text{EXET}(\tau_1) = \text{EXET}(S_1)$, and the hypothesis $\text{EXET}(S_1) < t$.
2. $t \leq \text{EXET}(S_1)$: In this case, there will be at least one insertion of c in S_1, after t time units, and possibly other insertions every T time units:
   $$\begin{array}{l} \mathcal{I}_c^T(S_1;S_2, t) = \mathcal{I}_c^T(S_1, t);\mathcal{I}_c^T(S_2, t_1) \\ \qquad \text{with } \text{EXET}(S_1) = t + q(T - \text{EXET}(c)) + r,\ 0 \leq q,\ 0 \leq r < T - \text{EXET}(c)), \\ \qquad\qquad t_1 = T - \text{EXET}(c) - r \end{array}$$

Since $t \leq \text{EXET}(S_1)$, S_1 satisfies the induction hypothesis and $\forall \tau_1 \in \mathit{Traces}(\mathcal{I}_c^T(S_1, t))$, $\tau_1 \doteq S_{1,1};c;S_{1,2} \ldots c;S_{1,m}$ $(1 \leq m)$ and verifies:

$$\begin{aligned} &t \leq \text{EXET}(S_{1,1}) < t + \varepsilon && (\mathit{Init}_1)\\ &T - \varepsilon < \text{EXET}(c;S_{1,i}) \leq T + \varepsilon \ \ (1 < i < m) && (\mathit{Period}_1)\\ &r - \varepsilon < \text{EXET}(S_{1,m}) \leq r && (\mathit{End}_1)\end{aligned}$$

The transformation is then applied on S_2 with the time residual $t_1 = T - \text{EXET}(c;S_{1,m})$. There are two sub-cases depending on the execution time of S_2.

a. $T - \text{EXET}(c) - r \leq \text{EXET}(S_2)$:
This is condition (2) to apply the induction hypothesis on S_2. Condition (3) is $-\varepsilon < T - \text{EXET}(c) - r \leq T$, which follows from the fact that $\text{EXET}(c)$ and r are positive and $r < T - \text{EXET}(c)$). By induction hypothesis, $\forall \tau_2 \in \mathit{Traces}(\mathcal{I}_c^T(S_2, T - (\text{EXET}(c) + r)))$, $\tau_2 \doteq S_{2,1};c;S_{2,2} \ldots c;S_{2,n}$ $(1 \leq n)$ and verifies:

$$\begin{aligned} &T - \text{EXET}(c) - r \leq \text{EXET}(S_{2,1}) < T - \text{EXET}(c) - r + \varepsilon && (\mathit{Init}_2)\\ &T - \varepsilon < \text{EXET}(c;S_{2,i}) \leq T + \varepsilon \ \ (1 < i < m) && (\mathit{Period}_2)\\ &r_2 - \varepsilon < \text{EXET}(S_{2,n}) \leq r_2 &&\\ &\qquad \text{If } \text{EXET}(S, 2) = T - \text{EXET}(c) - r + q_2(T - \text{EXET}(c)) + r_2 &&\\ &\qquad \text{with } 0 \leq q_2 \text{ and } 0 \leq r_2 < T - \text{EXET}(c) && (\mathit{End}_2)\end{aligned}$$

Any execution trace $\tau \in \mathit{Traces}(\mathcal{I}_c^T(S_1;S_2, t))$ is of the form:

$$\tau \doteq S_{1,1};c;S_{1,2} \ldots c;S_{1,m};S_{2,1};c;S_{2,2} \ldots c;S_{2,n}$$

We just have to check that $T - \varepsilon < \text{EXET}(c;S_{1,m};S_{2,1}) \leq T + \varepsilon$ which follows from:

$$\begin{aligned} &(\mathit{End}_1) && r - \varepsilon < \text{EXET}(S_{1,m}) \leq r\\ &(\mathit{Init}_2) && T - \text{EXET}(c) - r \leq \text{EXET}(S_{2,1}) < T - \text{EXET}(c) - r + \varepsilon\end{aligned}$$

We get $T - \varepsilon < \text{EXET}(c) + \text{EXET}(S_{1,m}) + \text{EXET}(S_{2,1}) < T + \varepsilon$, and the combined trace τ satisfies the property.

b. $\text{EXET}(S_2) < T - \text{EXET}(c) - r$:
Any execution trace $\tau \in \mathit{Traces}(\mathcal{I}_c^T(S_1;S_2, t))$ is of the form:

$$\tau \doteq S_{1,1};c;S_{1,2} \ldots c;S_{1,m};\tau_2$$

Since $\text{EXET}(S_1) = t + q(T - \text{EXET}(c)) + r$, $\mathit{exet}(S_1;S_2) = t + q(T - \text{EXET}(c)) + r + \text{EXET}(S_2)$ and $0 \leq r + \text{EXET}(S_2) < T - \mathit{exect}(c)$. We have to check that

$$r + \text{EXET}(S_2) - \varepsilon \leq \text{EXET}(S_{1,m};\tau_2) < r + \text{EXET}(S_2)$$

which follows directly from (End_1) and the fact that the *Traces* function satisfies the Property (33), i.e., $\text{EXET}(\tau_2) = \text{EXET}(S_2)$.

CASE $S = \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2$. Recall that:

$$\mathcal{I}_c^T(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, t) = \texttt{if } b \texttt{ then } \mathcal{I}_c^T(S_1, t-1) \texttt{ else } \mathcal{I}_c^T(S_2, t-1)$$

Hence, traces are of the form $\tau = \texttt{skip};S_{1,1};c;S_{1,2} \ldots c;S_{1,m}$ or $\tau = \texttt{skip};S_{2,1};c;S_{2,2} \ldots c;S_{2,m}$. Since $t > 0$ and $\varepsilon \leq 1$, we have $t - 1 > -\varepsilon$, so the induction hypothesis applies on S_1 (resp. S_2). Therefore, $\forall \tau_1 \in \mathit{Traces}(\mathcal{I}_c^T(S_1, t))$,

$\tau_1 \doteq S_{1,1};c;S_{1,2} \ldots c;S_{1,m}$ $(1 \le m)$ and verifies:

$$\begin{array}{lll} t-1 \le \text{EXET}(S_{1,1}) < t-1+\varepsilon & & (\mathit{Init}_1) \\ T-\varepsilon < \text{EXET}(c;S_{1,i}) \le T+\varepsilon \quad (1 < i < m) & & (\mathit{Period}_1) \\ r-\varepsilon < \text{EXET}(S_{1,m}) \le r \qquad \text{If } \text{EXET}(S) = t-1+q(T-\text{EXET}(c))+r & & (\mathit{End}) \\ \qquad\qquad\qquad\qquad\qquad \text{with } 0 \le q \text{ and } 0 \le r < T-\text{EXET}(c) & & \end{array}$$

It follows that $t \le \text{EXET}(\texttt{skip};S_{1,1}) < t+\varepsilon$ and the combined trace satisfies the property. The reasoning is the same with S_2.

CASE $S = \texttt{for}\ l = n_1\ \texttt{to}\ n_2\ \texttt{do}\ S$: Recall that:

$$\mathcal{I}_c^T(\texttt{for}\ l = n_1\ \texttt{to}\ n_2\ \texttt{do}\ S, t) = \mathit{Fold}\,(\mathcal{I}_c^T(\mathit{Unfold}\,(\texttt{for}\ l = n_1\ \texttt{to}\ n_2\ \texttt{do}\ S), t))$$

It follows that:

$$\begin{array}{l} \mathit{Traces}\,(\mathit{Fold}\,(\mathcal{I}_c^T(\mathit{Unfold}\,(\texttt{for}\ l = n_1\ \texttt{to}\ n_2\ \texttt{do}\ S), t))) \\ \quad = \mathit{Traces}\,(\mathcal{I}_c^T(\mathit{Unfold}\,(\texttt{for}\ l = n_1\ \texttt{to}\ n_2\ \texttt{do}\ S), t)) \\ \quad = \mathit{Traces}\,(\mathcal{I}_c^T(l := n_1;\ S; \ldots, t)) \end{array}$$

The operator *Unfold* replaces for-loop by sequences of commands. This case boils down to the already treated case $S = S_1;S_2$. □

PROOF THAT PROPERTY 6 HOLDS FOR NEGATIVE TIME RESIDUALS. For $-\varepsilon < t \le 0$ we have $\mathcal{I}_c^T(S, t) = c;\mathcal{I}_c^T(S, T - \text{EXET}(c) + t)$. Since Property 6 holds for positive time residuals, it follows from $\text{EXET}(c) + \varepsilon < T$ and $-\varepsilon < t$ that $T - \text{EXET}(c) + t$ is positive and therefore $\forall \tau \in \mathit{Traces}\,(\mathcal{I}_c^T(S, T - \text{EXET}(c) + t))$, $\tau \doteq S_1;c;S_2 \ldots c;S_n$ $(1 \le n)$ and verifies:

$$\begin{array}{lll} T-\text{EXET}(c)+t \le \text{EXET}(S_1) < T-\text{EXET}(c)+t+\varepsilon & & (\mathit{Init}) \\ T-\varepsilon < \text{EXET}(c;S_i) \le T+\varepsilon \quad (1 < i < n) & & (\mathit{Period}) \\ r-\varepsilon < \text{EXET}(S_n) \le r & & (\mathit{End}) \\ \qquad\qquad \text{If } \text{EXET}(S) = T-\text{EXET}(c)+t+q(T-\text{EXET}(c))+r & & \\ \qquad\qquad\quad \text{with } 0 \le q \text{ and } 0 \le r < T-\text{EXET}(c) & & \end{array}$$

The traces in $\mathit{Traces}\,(\mathcal{I}_c^T(S, T - \text{EXET}(c) + t))$ are of the form:

$$c;S_1;c;S_2 \ldots c;S_n \doteq \texttt{void};c;S_1;c;S_2 \ldots c;S_n$$

Since $\text{EXET}(\texttt{void}) = 0$ and, by hypothesis, $-\varepsilon < t \le 0$, we have:

$$t \le \text{EXET}(\texttt{void}) < t + \varepsilon \quad (\mathit{Init})$$

It remains to show that the (*Period*) condition holds, i.e., $T - \varepsilon < \text{EXET}(c;S_1) < T + \varepsilon$. We have:

$$T - \text{EXET}(c) + t \le \text{EXET}(S_1) < T - \text{EXET}(c) + t + \varepsilon$$

Since $\text{EXET}(c;S_1) = \text{EXET}(c) + \text{EXET}(S_1)$ and $-\varepsilon < t \le 0$, we conclude:

$$T - \varepsilon < T + t \le \text{EXET}(c;S_1) < T + t + \varepsilon \le T + \varepsilon$$ □

REFERENCES

Aggarwal, A. and Gupta, D. 2002. Failure detectors for distributed systems. Tech. rep., Indian Institute of Technology, Kanpur, India. http://resolute.ucsd.edu/diwaker/publications/ds.pdf.

Aguilera, M., Chen, W., and Toueg, S. 1997. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *Proceedings of the 11th International Workshop on Distributed Algorithms*. Saarbrucken, Germany. Springer-Verlag, Berlin, 126–140.

Arora, A. and Kulkarni, S. 1998. Detectors and correctors: A theory of fault-tolerance components. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'98)*. Amsterdam, The Netherlands. IEEE, Los Alamitos, CA. 436–443.

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Depend. Secure Comput. 1*, 1, 11–33.

Aydin, H., Melhem, R., and Mossé, D. 2000. Optimal scheduling of imprecise computation tasks in the presence of multiple faults. In *Proceedings of Real-Time Computing Systems and Applications (RTCSA'00)*. Cheju Island, South Korea. IEEE, Los Alamitos, CA. 289–296.

Baille, G., Garnier, P., Mathieu, H., and Pissard-Gibollet, R. 1999. Le CyCab de l'Inria Rhne-Alpes. Tech. rep. 0229, Inria, Rocquencourt, France.

Beck, M., Plank, J., and Kingsley, G. 1994. Compiler-assisted checkpointing. Tech. rep., University of Tennessee.

Buttazzo, G. 2005. Rate monotonic vs EDF: Judgment day. *Real-Time Syst. 29*, 1, 5–26.

Caspi, P., Mazuet, C., Salem, R., and Weber, D. 1999. Formal design of distributed control systems with Lustre. In *Proceedings of International Conference on Computer Safety, Reliabilitiy, and Security (SAFECOMP'99)*. Lecture Notes in Computer Science, vol. 1698. Springer-Verlag, Berlin. 396–409.

Chandra, T. and Toueg, S. 1996. Unreliable failure detectors for reliable distributed systems. *J. ACM 43*, 2, 225–267.

Colin, A. and Puaut, I. 2000. Worst case execution time analysis for a processor with branch prediction. *Real Time Syst. 18*, 2/3, 249–274.

Cristian, F. 1991. Understanding fault-tolerant distributed systems. *Comm. ACM 34*, 2, 56–78.

Dean, A. and Shen, J. 1998. Hardware to software migration with real-time thread integration. In *Proceedings of the Euromicro Conference*. Västeras, Sweden. IEEE, Los Alamitos, CA. 10243–10252.

Dumitrescu, E., Girault, A., Marchand, H., and Rutten, E. 2007. Optimal discrete controller synthesis for modeling fault-tolerant distributed systems. In *Workshop on Dependable Control of Discrete Systems (DCDS'07)*. Cachan, France. IFAC, New York. 23–28.

Dumitrescu, E., Girault, A., and Rutten, E. 2004. Validating fault-tolerant behaviors of synchronous system specifications by discrete controller synthesis. In *Workshop on Discrete Event Systems (WODES'04)*. Reims. France. IFAC, New York.

Fisher, M., Lynch, N., and Paterson, M. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2, 374–382.

Girault, A. and Rutten, E. 2004. Discrete controller synthesis for fault-tolerant distributed systems. In *Proceedings of the International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04)*. Electronic Notes in Theoretical Computer Science, vol. 133, Elsevier Science, New York. 81–100.

Girault, A. and Yu, H. 2006. A flexible method to tolerate value sensor failures. In *Proceedings of the International Conference on Emerging Technologies and Factory Automation (ETFA'06)*. Prague, Czech Republic. IEEE, New York. 86–93.

Grandpierre, T., Lavarenne, C., and Sorel, Y. 1999. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of the 7th International Workshop on Hardware/Software Co-Design (CODES'99)*. Rome, Italy. ACM, New York.

Grandpierre, T. and Sorel, Y. 2003. From algorithm and architecture specifications to automatic generation of distributed real-time executives: A seamless flow of graphs transformations. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE'03)*. Mont Saint-Michel, France. IEEE, Los Alamitos, CA.

Jalote, P. 1994. *Fault-Tolerance in Distributed Systems*. Prentice-Hall, Englewood Cliffs, NJ.

KALAISELVI, S. AND RAJARAMAN, V. 2000. A survey of checkpointing algorithms for parallel and distributed computers. *Sadhana 25*, 5, 489–510.

KOPETZ, H. 1997. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishing, Novell, MA.

KULKARNI, S. AND ARORA, A. 2000. Automating the addition of fault-tolerance. In *Proceedings of the International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'00)*. M. Joseph, Ed. Lecture Notes in Computer Science, vol. 1926, Springer-Verlag, Berlin, 82–93.

LISPER, B. 2006. Trends in timing analysis. In *Proceedings of the IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES'06)*. Braga, Portugal. Springer, Berlin, 85–94.

LIU, C. AND LAYLAND, J. 1973. Scheduling algorithms for multiprogramming in hard real-time environnement. *J. ACM 20*, 1, 46–61.

LIU, Z. AND JOSEPH, M. 1992. Transformation of programs for fault-tolerance. *Formal Aspects Comput. 4*, 5, 442–469.

MILNER, R., TOFTE, M., AND HARPER, R. 1990. *The Definition of Standard ML*. MIT Press, Cambridge, MA.

MOSSÉ, D., MELHEM, R., AND GHOSH, S. 2003. A nonpreemptive real-time scheduler with recovery from transient faults and its implementation. *IEEE Trans. Software Engin. 29*, 8, 752–767.

NELSON, V. 1990. Fault-tolerant computing: Fundamental concepts. *IEEE Comput. 23*, 7, 19–25.

NIELSON, H. AND NIELSON, F. 1992. *Semantics with Applications—A Formal Introduction*. Wiley, New York, NY.

PUSCHNER, P. 2002. Transforming execution-time boundable code into temporally predictable code. In *Design and Analysis of Distributed Embedded Systems (DIPES'02)*, B. Kleinjohann, K. Kim, L. Kleinjohann, and A. Rettberg, Eds. Kluwer Academic Publishing.

PUSCHNER, P. AND BURNS, A. 2000. A review of worst-case execution-time analysis. *Real-Time Syst. 18*, 2/3, 115–128.

RAMADGE, P. AND WONHAM, W. 1987. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim. 25*, 1, 206–230.

RUSHBY, J. 2001. Bus architectures for safety-critical embedded systems. In *Proceedings of the International Workshop on Embedded Systems (EMSOFT'01)*. Lecture Notes in Computer Science, vol. 2211, Springer-Verlag, Berlin.

SEKHAVAT, S. AND HERMOSILLO, J. 2000. The Cycab robot: A differentially flat system. In *Proceedings of the IEEE Conference on Intelligent Robots and Systems (IROS'00)*. Takamatsu, Japan. IEEE, Los Alamitos, CA.

SILVA, L. AND SILVA, J. 1998. System-level versus user-defined checkpointing. In *Proceedings of the Symposium on Reliable Distributed Systems (SRDS'98)*. West Lafayette, IN. IEEE, Los Alamitos, CA. 68–74.

THEILING, H., FERDINAND, C., AND WILHELM, R. 2000. Fast and precise WCET prediction by separate cache and path analyses. *Real-Time Sys. 18*. 2/3, 157–179.

ZIV, A. AND BRUCK, J. 1997. An on-line algorithm for checkpoint placement. *IEEE Trans. Comput. 46*, 9, 976–985.

Received June 2006; revised February 2007; accepted May 2007