

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Implementing Fault-Tolerance in Real-Time Systems by Program Transformations

Tolga Ayav — Pascal Fradet — Alain Girault

N° 5919

May 2006

Thème COM



*Rapport
de recherche*

Implementing Fault-Tolerance in Real-Time Systems by Program Transformations

Tolga Ayav , Pascal Fradet , Alain Girault

Thème COM — Systèmes communicants
Projet Pop-Art

Rapport de recherche n° 5919 — May 2006 — 42 pages

Abstract: 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 transformation highlights the benefits of separation of concerns and allows us to establish correctness properties (including the satisfaction of real-time constraints). We also present an implementation of our method, to demonstrate its feasibility and its efficiency.

Key-words: Fault-tolerance, heartbeating, checkpointing, program transformations, correctness proofs.

Mise en œuvre de la tolérance aux fautes par transformation de programme

Résumé : Nous proposons une approche formelle pour la tolérance aux fautes dans des systèmes embarqués temps réels. Le système initial consiste en un ensemble de tâches périodiques et indépendantes, ordonnancées sur un ensemble de processeurs à silence sur défaillance reliés par un réseau de communication fiable. Nous transformons les tâches de telle sorte qu'en supposant l'existence d'un processeur de secours, elles tolèrent une défaillance (permanente ou transitoire) à la fois. La détection des défaillances est réalisée en vérifiant la présence de signaux périodiques (*heartbeating*) et la reprise de l'exécution sur le processeur de secours est faite grâce à des points de reprise (*checkpointing* et *rollback*). Ces techniques sont décrites et implantées par des transformations automatiques de programmes du programme des tâches. Cette approche formelle pour la tolérance aux fautes permet de séparer les problèmes et de montrer la correction de propriétés clés comme le respect des contraintes temps réels. Nous présentons l'application de notre méthode sur un cas d'étude afin de démontrer sa faisabilité et son efficacité.

Mots-clés : Tolérance aux fautes, points de reprise, transformations de programme, preuves de correction.

1 Introduction

In most distributed embedded systems, such as automotive and avionics, fault-tolerance is a crucial issue [Cristian, 1991, Nelson, 1990, Jalote, 1994]. Fault-tolerance 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 the 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 before 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 this last 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, library-supported, etc [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 has no 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 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 con-

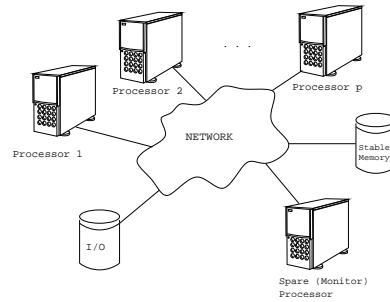


Figure 1: System architecture.

sistent 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 system after the transformations.

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). 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 non-zero transmission times as well.

Assumption 2 *All processors show omission/crash failure behavior [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 that fit 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 described in Figure 2. 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 give such an application with static schedules made of dependent tasks and deterministic and non-zero communication times, and we solve it with our method.

```

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

```

Figure 2: Program model of periodic real-time tasks.

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 multi-rate 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 keep 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.

When the monitor detects a processor failure, it rolls back to the latest local state of the faulty processor stored in the stable memory. Then, it 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 amounts 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 in 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 due to the 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.

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 inputs (which are stored in a local variable), executes some statements, and writes its outputs (see Figure 2). 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 programming language:

$S ::=$	<code>x:=A</code>	<i>assignment</i>
	<code>skip</code>	<i>no operation</i>
	<code>read(i)</code>	<i>input read</i>
	<code>write(o)</code>	<i>output write</i>
	<code>S₁;S₂</code>	<i>sequencing</i>
	<code>if B then S₁ else S₂</code>	<i>conditional</i>
	<code>for i = n₁ to n₂ do S</code>	<i>iteration</i>

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 inputs and outputs

are i and o . These instructions could be generalized to multiple reads and writes or to IO operations parameterized with a port. This language is well-known, simple and expressive enough. 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 present concisely 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) S can be evaluated. The `WCET` analysis is the topic of much work (see [Puschner and Burns, 1999, Li et al., 2005] for instance); we shall not dwell upon this issue any further.

For the remaining of the article, we fix the execution times of statements to be (in time units) those of Table 1.

<code>EXET(skip)</code>	<code>BCET(skip)</code>	<code>WCET(skip)</code>	= 1
<code>EXET(read)</code>	<code>BCET(read)</code>	<code>WCET(read)</code>	= 3
<code>EXET(write)</code>	<code>BCET(write)</code>	<code>WCET(write)</code>	= 3
<code>EXET(x := e)</code>	<code>BCET(x := e)</code>	<code>WCET(x := e)</code>	= 3
<code>EXET(S₁;S₂)</code>			= <code>EXET(S₁) + EXET(S₂)</code>
<code>BCET(S₁;S₂)</code>			= <code>BCET(S₁) + BCET(S₂)</code>
<code>WCET(S₁;S₂)</code>			= <code>WCET(S₁) + WCET(S₂)</code>
<code>WCET(if B then S₁ else S₂)</code>			= <code>1 + max(WCET(S₁), WCET(S₂))</code>
<code>BCET(if B then S₁ else S₂)</code>			= <code>1 + min(BCET(S₁), BCET(S₂))</code>
<code>EXET(for i = n₁ to n₂ do S)</code>			= <code>(n₂ - n₁ + 1) × (3 + EXET(S))</code>
<code>BCET(for i = n₁ to n₂ do S)</code>			= <code>(n₂ - n₁ + 1) × (3 + BCET(S))</code>
<code>WCET(for i = n₁ to n₂ do S)</code>			= <code>(n₂ - n₁ + 1) × (3 + WCET(S))</code>

Table 1: Exact, worst case, and best-case execution times of our programming language's statements.

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 EXET by WCET in the right-hand part (resp. BCET); 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 $\{(S_1, T_1), \dots, (S_n, T_n)\}$ is that each task must meet its deadline. Since each processor runs a single task, it amounts to:

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

The semantics of a statement S is given by the function $\llbracket S \rrbracket : \mathbf{State} \rightarrow \mathbf{State}$. A state $s \in \mathbf{State}$ 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: $\llbracket S \rrbracket_{s_0} = s_f$. Several equivalent formal definitions of $\llbracket \cdot \rrbracket$ (operational, denotational, axiomatic) can be found in [Nielson and Nielson, 1992].

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

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

where i_k is the input provided by the environment during the k th period and o_k is the last output written during the k th period. So, if several $\text{write}(o)$ 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 $\text{read}(i)$ during the same period will result in the same readings.

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

```
read(i); o := i; write(o); read(i); o := o * i; 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:

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

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 conditional and loops to their worst case execution time. Intuitively, it amounts

to adding dummy code to conditional and loop statements. Such transformations suppose to be able to evaluate the WCET of programs. After this time equalization, checkpoints and heartbeats can be introduced simply using the same transformation. To ensure timing correctness in these transformations, we also assume the use of processors that do not have complex behaviors such as pipelining and caching. This assumption is realistic since the real-time embedded applications generally do not rely on these features.

A transformation may increase the WCET of programs. So, 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 less expensive branches. The dummy code added for padding is sequences of `skip` statements. We write `skipn` to represent a sequence of n `skip` statements: $\text{EXET}(\text{skip}^n) = n$. This technique is similar to the one used in “single path programming” [Puschner, 2002].

The global equalization process is defined inductively 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 rules 1

1. $\mathcal{F}[\text{if } B \text{ then } S_1 \text{ else } S_2]$ = $\text{if } B \text{ then } \mathcal{F}[S_1]; \text{skip}^{\max(0, \delta_2 - \delta_1)};$
 $\text{else } \mathcal{F}[S_2]; \text{skip}^{\max(0, \delta_1 - \delta_2)};$
with $\delta_i = \text{WCET}(\mathcal{F}[S_i])$ *for* $i = 1, 2$
2. $\mathcal{F}[\text{for } i = n_1 \text{ to } n_2 \text{ do } S]$ = $\text{for } i = n_1 \text{ to } n_2 \text{ do } \mathcal{F}[S]$
3. $\mathcal{F}[S_1; S_2]$ = $\mathcal{F}[S_1]; \mathcal{F}[S_2]$
4. $\mathcal{F}[S]$ = S *otherwise*

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 “less 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 $\forall S, \text{BCET}(\mathcal{F}[S]) = \text{WCET}(\mathcal{F}[S]) = \text{EXET}(\mathcal{F}[S])$.

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

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

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 $\text{wcet}(\mathcal{F}[S]) = 1 + \max(\delta_1, \delta_1) = 1 + \delta_1$.

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

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

- Let $S = \text{for } i = n_1 \text{ to } n_2 \text{ do } S_1$. The induction hypothesis is that $\text{bcet}(\mathcal{F}[S_1]) = \text{wcet}(\mathcal{F}[S_1]) = \text{exet}(\mathcal{F}[S_1]) = \delta_1$. According to Rule 2 and Table 1, we thus have $\text{exet}(\mathcal{F}[S]) = (n_2 - n_1 + 1) \times (3 + \delta_1)$. Since n_1 and n_2 are constant and by induction hypothesis, this is also equal to $\text{bcet}(\mathcal{F}[S])$ and $\text{wcet}(\mathcal{F}[S])$.
- Let $S = S_1; S_2$. The induction hypothesis is that $\text{bcet}(\mathcal{F}[S_1]) = \text{wcet}(\mathcal{F}[S_1]) = \text{exet}(\mathcal{F}[S_1]) = \delta_1$ and $\text{bcet}(\mathcal{F}[S_2]) = \text{wcet}(\mathcal{F}[S_2]) = \text{exet}(\mathcal{F}[S_2]) = \delta_2$. According to Rule 3 and Table 1, we thus have $\text{exet}(\mathcal{F}[S]) = \delta_1 + \delta_2$. By induction hypothesis, this is also equal to $\text{bcet}(\mathcal{F}[S])$ and $\text{wcet}(\mathcal{F}[S])$.

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

Furthermore, 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 $S = \text{if } B \text{ then } S_1 \text{ else } S_2$. 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 1, we thus have:

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

Moreover, according to Table 1, 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 $S = \text{for } i = n_1 \text{ to } n_2 \text{ do } S_1$. The induction hypothesis is that $\text{WCET}(S_1) = \text{WCET}(\mathcal{F}[S_1])$. According to Rule 2 and Table 1, 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 1, 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 $S = S_1; S_2$. 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 1, we thus have $\text{WCET}(\mathcal{F}[S]) = \text{WCET}(\mathcal{F}[S_1]) + \text{WCET}(\mathcal{F}[S_2])$. Moreover, according to Table 1, 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 , $\text{WCET}(S) = \text{WCET}(\mathcal{F}[S])$. \square

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

```

Fac1 =  $\mathcal{F}[Fac]$  = read( $i$ );
           if  $i > 10$  then  $i := 10$ ;  $o := 1$ ; else  $o := 1$ ; skip3;
           for  $l = 1$  to 10 do
               if  $l \leq i$  then  $o := o * l$ ; else skip3;
           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..n]` gathers the heartbeat variables of the n tasks. The command `hbeat` in task i is implemented as `HBT[i] := 1`.
- `ckpt` saves the current state in the stable memory. It is sufficient to save only the live variables and only those which have been modified since the last checkpoint. This information can be inferred by static analysis techniques. Here, we simply assume that `ckpt` 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., $\text{WCET}(\text{read}) = 3$), 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 re-apply the transformation \mathcal{F} after checkpointing. Again, the rules below must be understood like a case expression in ML.

Transformation rules 2

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

Rule 1 states that, when the statement S finishes before the next insertion time t (i.e., $\text{EXET}(S) < t$), the transformation terminates and nothing is inserted. In all the other cases (rules 2 to 6), the WCET of S is greater than t and at least one insertion must be performed.

Rule 2 applies when the time counter t is negative. This case may arise when the ideal point for inserting the command c is “in the middle” of the boolean expression of a conditional statement `if`. When t is negative, the command must be inserted right away. 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 clock drift.

Rule 3 states that, when the program is an atomic command a (whose EXET is greater than or equal to t), the command c is inserted right after a , that is $(\text{EXET}(a) - t)$ units of time later than the ideal point.

Rule 4 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 5 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 6 applies to loop statements. It unrolls the loop completely (thanks to the *Unfold* operator), performs the insertion in the unrolled resulting program (hence the $\langle S'|t' \rangle = \mathcal{I}_c^T(S'', t)$), and then factorizes code by folding code in `for` loops as much as possible (thanks to the *Fold* operator). The *Unfold* and *Fold* operators are defined by the following transformation rules:

Transformation rules 3

1. *Unfold*(`for` $l = n_1$ `to` n_2 `do` S) = $l := n_1; S; l := n_1 + 1; \dots l := n_2; S$
2. *Fold*((`for` $l = n_1$ `to` n_2 `do` S); $l := n_2 + 1; S$;) = `for` $l = n_1$ `to` $n_2 + 1$ `do` S

Actually, 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 chose instead 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.

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.

Checkpointing and heartbeating are performed using the transformation \mathcal{I} . Checkpoints are inserted first and heartbeats last. The period between two checkpoints must take into account the overhead that will be added by heartbeats afterward. The overhead added by heartbeating during X units of time is $\frac{X\bar{h}}{T_{HB}-\bar{h}}$ with $\bar{h} = \text{WCET}(\text{hbeat})$. So, if T_{CP} is the desired period of checkpoints, we must use the period T'_{CP} defined by the equation:

$$T'_{CP} = T_{CP} - \frac{T'_{CP}\bar{h}}{T_{HB} - \bar{h}} \iff T'_{CP} \left(1 + \frac{\bar{h}}{T_{HB} - \bar{h}}\right) = T_{CP} \iff T'_{CP} = \frac{T_{CP}}{1 + \frac{\bar{h}}{T_{HB} - \bar{h}}} \quad (2)$$

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

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

The first heartbeat is added right at the beginning of S' , the others are inserted by \mathcal{I} , then the last heartbeat is replaced by $\text{hbeat}(k)$. We can always ensure that S' finishes with a heartbeat by padding dummy code at the end. The command $\text{hbeat}(k)$ is a special heartbeat that sets the variable to k instead of 1, i.e., $\text{HBT}[i] := k$. Following this last heartbeat, the monitor will therefore decrease the shared variable and will resume error 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:

$$k = \left\lceil \frac{T - \text{WCET}(S''; \text{hbeat})}{T_{HB}} \right\rceil \quad (3)$$

After the introduction of heartbeats, the period between checkpoints will be $T'_{CP} \left(1 + \frac{\bar{h}}{T_{HB} - \bar{h}}\right)$, 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:*

$$T_{CP} - \varepsilon \leq \Delta_{CP} < T_{CP} + \varepsilon + \bar{h} \quad \text{and} \quad T_{HB} - \varepsilon \leq \Delta_{HB} < T_{HB} + \varepsilon$$

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

$$T'_{CP} - \varepsilon \leq \Delta'_{CP} < T'_{CP} + \varepsilon \quad (4)$$

Assuming the WCET of the most expensive atomic command of `checkpt` is less than or equal to ε , after the second transformation, $\mathcal{I}_{\text{hbeat}}^{T_{HB}}(\text{hbeat}; S', T_{HB})$, Property 3 satisfies the condition $T_{HB} - \varepsilon \leq \Delta_{HB} < T_{HB} + \varepsilon$. The second transformation, however, changes Δ'_{CP} given in Equation (4) to Δ_{CP} such that each portion with the time interval T'_{CP} in the final program will be inserted $\frac{T'_{CP}}{T_{HB} - \bar{h}}$ `hbeat` commands. Therefore, by following Equation (2), $T'_{CP} + \frac{T'_{CP}}{T_{HB} - \bar{h}} \cdot \bar{h}$ 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 (2) when a heartbeat is inserted just before a checkpoint command. In this case, T_{CP} is shifted upwards by \bar{h} . In the best case, this shift is zero. Therefore, by shifting up the lower and upper bounds of Δ_{CP} with $[0, \bar{h}]$, we finally derive $T_{CP} - \varepsilon \leq \Delta_{CP} < T_{CP} + \varepsilon + \bar{h}$. \square

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:

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

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

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

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

```

Fac2 = read(i);
      if i > 10 then i := 10; o := 1; else o := 1; skip3;
      for l = 1 to 6 do
        if l <= i then o := o * l; else skip3;
      l := 7; if l <= i then checkpt; o := o * l; else checkpt; skip3;
      for l = 8 to 10 do
        if l <= i then o := o * l; else skip3;
      write(o);

```

A single `checkpt` is inserted after 56 time units, which happens within 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, 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 = checkpt1;checkpt2` where `checkpt1` and `checkpt2` take respectively 7 and 3 time units exactly. In other words, the largest `WCET` of an atomic instruction remains 3 (it would be 10 if `checkpt` was atomic). We add a `hbeat` as a first instruction and, in order to finish with a `hbeat`, we must add 5 `skip` at the end. The transformation $\mathcal{I}_{\text{hbeat}}^{10}(\text{Fac}_2, 10)$ inserts a `hbeat` every 10 time units and yields:

```

Fac3 = hbeat; read(i);
      if i > 10 then i := 10; hbeat; o := 1; else o := 1; hbeat; skip3;
      for l = 1 to 6 do
        if l <= i then hbeat; o := o * l; else hbeat; skip3;
      l := 7; if l <= i then hbeat; checkpt1; hbeat; checkpt2; o := o * l;
        else hbeat; checkpt1; hbeat; checkpt2; skip3;
      for l = 8 to 10 do
        hbeat; if l <= i then o := o * l; else skip3;
      write(o); hbeat; skip5; hbeat;

```

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

- Between the `hbeat` located between `checkpt1` and `checkpt2`, and the `hbeat` located inside the second `for` loop, the interval is 12 time units. This is due to 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 $8 - \text{EXET}(\text{hbeat}) = 8 - 3 = 5$ 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 (3) gives $\lceil \frac{200-143}{10} \rceil = 6$, so the last `hbeat` must be changed into `hbeat(6)`. Figure 3 illustrates the form of a general program (*i.e.*, not Fac_3) after all the transformations:

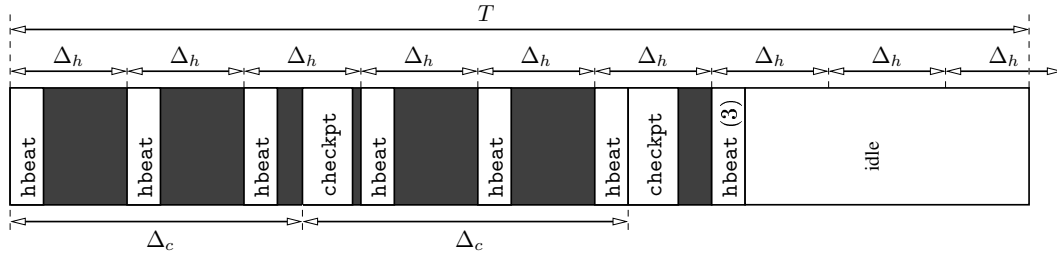


Figure 3: Program with checkpointing and heartbeating.

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 $\text{HBT}[i]$ 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 $\text{HBT}[i]$ at least at the period T_{HB_i} . Since each processor (or each task) has a potentially different heartbeat period by construction, 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. The period of the task 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:

$$Det_i = \text{HBT}[i] := \text{HBT}[i] - 1;$$

$$\text{if } \text{HBT}[i] = -2 \text{ then run } Rec(i);$$

When positive, $\text{HBT}[i]$ contains the number of T_{HB_i} periods before the next heartbeat of τ_i , hence the next update of $\text{HBT}[i]$. When it is equal to -2 , the monitor decides that the processor i is faulty, so it must launch the failure recovery program Rec . When $\text{HBT}[i]$ is equal to -1 , the processor i is suspected but not yet declared faulty. Indeed, it might just be late, or $\text{HBT}[i]$ might not have been updated yet due to 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 ($\xi_i \ll T_{HB_i}$), then the worst case detection time α_i of the failure of task τ_i satisfies:

$$\alpha_i < 3(T_{HB_i} + \varepsilon + \xi_i) + L_r + L_w \quad (5)$$

Finally, the problem of the clock drift between the task τ_i that writes $\text{HBT}[i]$, and the task Det_i that reads $\text{HBT}[i]$, 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 and starts executing the failed task instead. The following program represents the recovery operation:

```
Rec (x) = FAILED := x;
        restart ( $\tau_x$ , CONTEXT $_x$ );
```

where `restart` (τ_x , CONTEXT $_x$) is a macro that stops the monitor application and instead restarts τ_x from its latest safe point 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` $x \in \{1, 2, \dots, n\}$ 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 re-execute the part of the code after the last checkpoint. If we denote the time for context reading by L_C , then the worst case recovery time β is:

$$\beta = 3 \left(T_{HB} + \varepsilon + \max_{1 \leq i \leq n} \xi_i \right) + T_{CP} + L_r + L_w + L_C + \text{WCET}(Det) + \text{WCET}(Rec) \quad (6)$$

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 (7) below:

$$\text{WCET}(S'') = \text{WCET}(S) + \left\lfloor \frac{\text{WCET}(S)}{T_{CP}} \right\rfloor \times \text{WCET}(\text{ckpt}) + \left(\frac{\text{WCET}(S)}{T_{HB}} + 1 \right) \times \text{WCET}(\text{hbeat}) + \beta \quad (7)$$

Note that this `WCET` includes both the error detection time and recovery time. One may also be interested in the *optimum* values, T_{CP}^* and T_{HB}^* , i.e., the values that offer the best trade-off between fast failure detection, fast failure recovery, and least overhead due to the code insertion. If we combine Equations (7) and (6), we obtain a two-value function f of

the form:

$$f(T_{CP}, T_{HB}) = \frac{\text{WCET}(S) \times \text{WCET}(\text{checkpt})}{T_{CP}} + \frac{\text{WCET}(S) \times \text{WCET}(\text{hbeat})}{T_{HB}} + 3T_{HB} + T_{CP} + K \quad (8)$$

where K is a constant (see Figure 4 for a 3 dimension plot of f with the numerical values of the *Fac* example). Note that neglecting the floor function in Equation (7) is for the purpose of an explicit computation and yields approximate results. Since the least overhead due

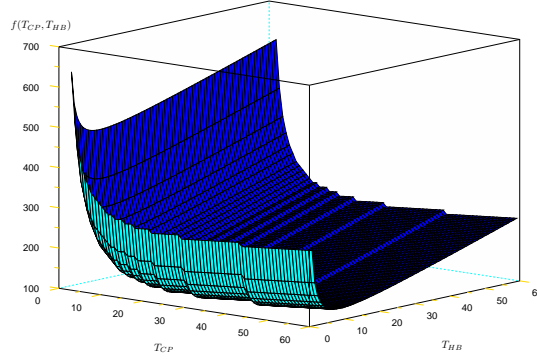


Figure 4: $f(T_{CP}, T_{HB})$.

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

$$\frac{\partial f}{\partial T_{CP}} = 1 - \frac{\text{WCET}(S) \times \text{WCET}(\text{checkpt})}{T_{CP}^2} \quad \text{and} \quad \frac{\partial f}{\partial T_{HB}} = 3 - \frac{\text{WCET}(S) \times \text{WCET}(\text{hbeat})}{T_{HB}^2} \quad (9)$$

Since the two second partial derivatives are positive in the $(0, +\infty) \times (0, +\infty)$ portion of the space, the function f is *convex* and the optimal values T_{CP}^* and T_{HB}^* are those that nullify the two first order partial derivatives. Hence:

$$\left. \frac{\partial f}{\partial T_{CP}} \right|_{T_{CP}=T_{CP}^*} = 0 \implies T_{CP}^* = \sqrt{\text{WCET}(S) \times \text{WCET}(\text{checkpt})} \quad (10)$$

$$\left. \frac{\partial f}{\partial T_{HB}} \right|_{T_{HB}=T_{HB}^*} = 0 \implies T_{HB}^* = \sqrt{\frac{1}{3} \text{WCET}(S) \times \text{WCET}(\text{hbeat})} \quad (11)$$

With our *Fac* example, we get $T_{CP}^* = \sqrt{84 \times 10} \simeq 28.98$ and $T_{HB}^* = \sqrt{\frac{84 \times 3}{3}} \simeq 9.16$. This means that the values we have chosen, respectively 80 and 10, were not the optimal values.

Equations (10) and (11) 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:

$$f(T_{CP_i}, T_{HB_i}) < T_i, \quad \forall i \in \{1, 2, \dots, n\} \quad (12)$$

Removing the assumption of zero communication time just involves adding a worst case communication delay parameter in Equations (5) and (6), which does not have an effect on the optimum values, T_{CP}^* and T_{HB}^* .

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

Property 5 *The real-time distributed system with the specifications drawn in this work can always tolerate one failure respecting its real-time constraints.*

Proof: The recovery time β given in Equation (6) relies on fixed heartbeating and checkpointing intervals (given in Property 4). Therefore, according to Condition (12), there exist T_{CP} and T_{HB} such that the algorithm completes before its deadline against one failure. \square

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 $\Gamma = \{(Det_1, T_{HB_1}), (Det_2, T_{HB_2}), \dots, (Det_n, T_{HB_n})\}$ that must satisfy:

$$\forall i \in \{1, 2, \dots, n\}, \text{WCET}(Det_i) \leq T_{HB_i}. \quad (13)$$

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 the tasks' deadlines (in other words, as a task becomes 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:

$$\sum_{i=1}^n \frac{\text{WCET}(Det_i)}{T_{HB_i}} \leq 2(2^{1/n} - 1) \quad (14)$$

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

$$\sum_{i=1}^n \frac{\text{WCET}(Det_i)}{T_{HB_i}} \leq 1 \quad (15)$$

The above schedulability conditions highlight that EDF allows a better processor utilization while both are appropriate and sufficient for scheduling the monitoring tasks with deadline guarantee.

6 Tolerating transient and multiple failures

We propose two extensions to our approach. The first one concerns the duration of 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 to address 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 restarts too, 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. The following code implements this action:

```
Remi = if FAILED = i and SPARE ≠ This Processor then
        SPARE := This Processor; FAILED := 0; restart_monitor ;
```

where *This Processor* is the ID of the processor executing that code and `restart_monitor` is a macro that terminates the task and restarts the monitoring application. The shared variable SPARE is initially set to the identification number of the spare processor. 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 restarting after a transient failure on the faulty processor will detect that it must stop and will restart the monitor. The code *Rem_i* must be added to the program of τ_i before the output update:

```
write(o)    ⇒    Remi; write(o);
```

In order to detect any processor failure and to guarantee the real-time constraints, the duration of the transient failure must be larger than the max of the failure detection times α_i (c.f. Equation 5 in Section 5.1).

The second extension is to tolerate *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 would 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. 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, results 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.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 made through an electrical jack mechanically linked to the wheels. Each wheel motor block has its own power amplifier, driven by an MPC555 micro-controller. 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.s^{-1} . In normal operation, the architecture consists of two MPC555s and a PC board which drives the screen and the hard disk. In the remaining of this article, we call these nodes F555, R555, and ROOT respectively. In order to implement our program transformations, we have added one more node, named MONITOR. The architecture graph of the CYCAB is therefore given in Figure 6.

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 multicomponent architecture. The implementation is formalized in terms of graphs transformations [Grandpierre et al., 1999, Grandpierre and Sorel, 2003]. The algorithm graph of this manual-driving application is given in Figure 7.

Task execution times and communication times are given in Tables 2 and 4 respectively.

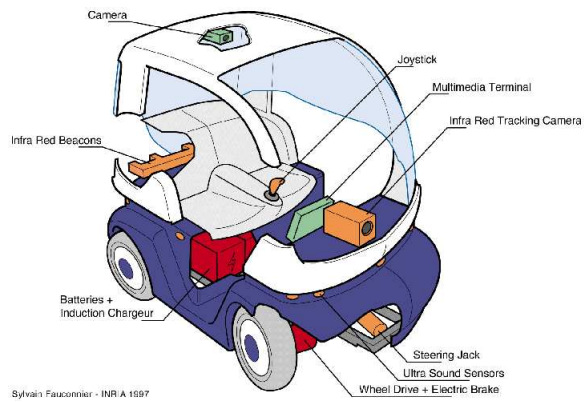


Figure 5: The CyCAB vehicle.

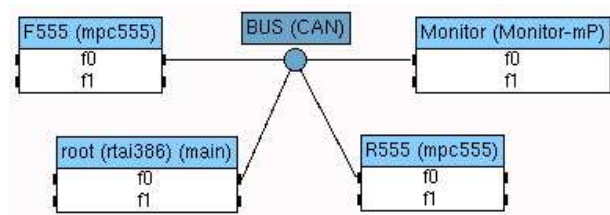


Figure 6: Architecture graph of the CyCAB application.

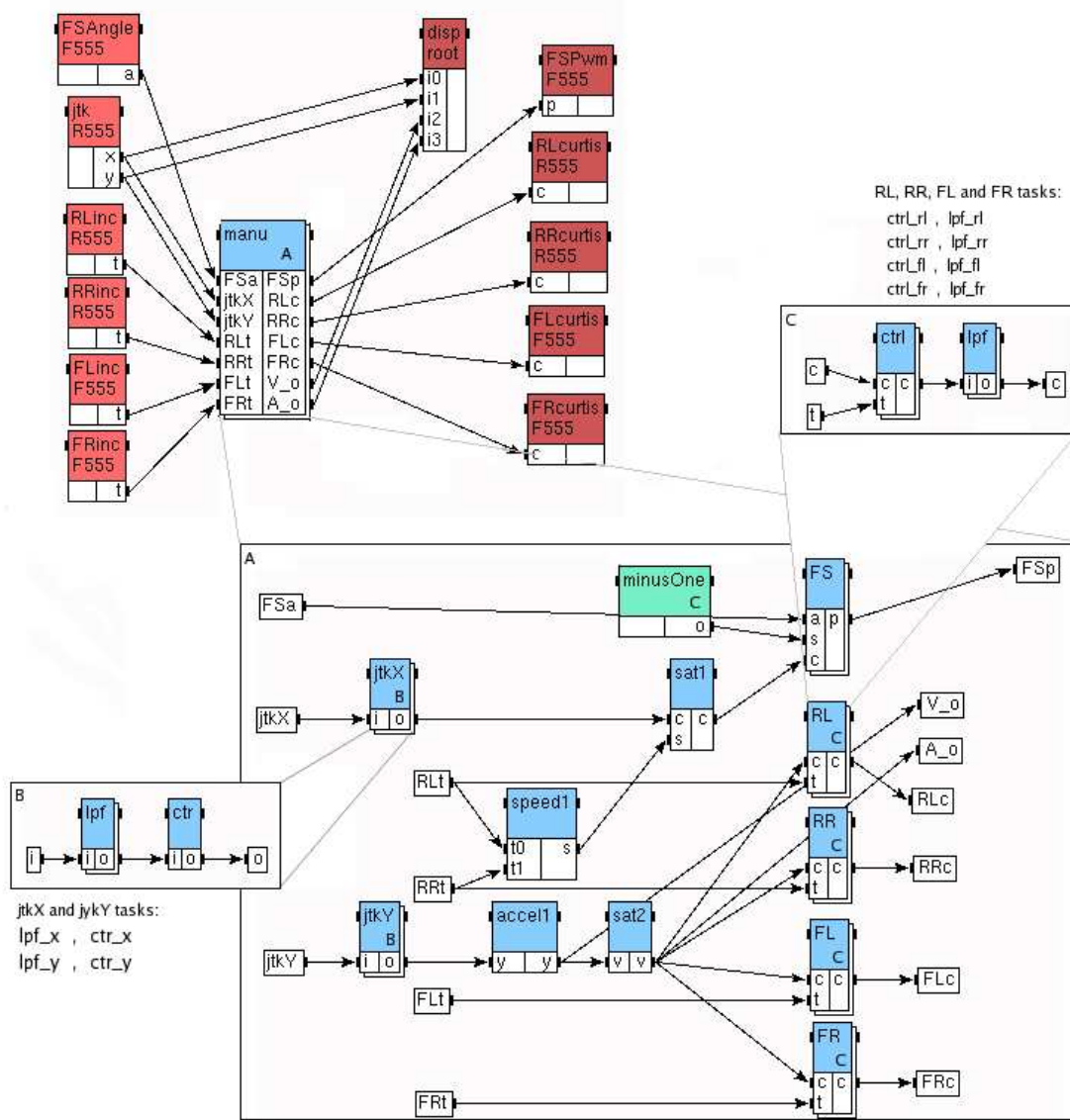


Figure 7: Application graph of Cycab. A processor name written inside a task indicates a processor constraint, i.e., that task must be scheduled onto that processor.

Table 2: Task execution times (*ms*) of the Cycab application algorithm.

Task name	WCET on F555	on R555	on ROOT	on Monitor
FSAngle, FSPwm	0.3	∞	∞	∞
jtk	∞	0.3	∞	∞
RLinc, RRinc	∞	0.2	∞	∞
FLinc, FRinc	0.2	∞	∞	∞
ctr_x, ctr_y, FS	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	∞
accel1	0.3	0.3	0.3	∞
sat1	0.3	0.3	0.3	∞
RLcurtis, RRcurtis	∞	0.5	∞	∞
FLcurtis, FRcurtis	0.5	∞	∞	∞
disp	∞	∞	0.5	∞

Table 3: Task execution times (*ms*) of heartbeating and checkpointing.

Task name	WCET on F555	on R555	on ROOT	on Monitor
hbeat1, cp1	0.06	∞	∞	∞
hbeat2, cp2	∞	0.06	∞	∞
hbeat3, cp3	∞	∞	0.06	∞
monitor1, monitor2, monitor3 cpsave1, cpsave2, cpsave3	∞	∞	∞	0.06

Table 4: Communication times.

Communication	Duration (<i>ms</i>)
hbeat \rightarrow monitor	$\Delta_{\text{hbeat}} = 0.12$
checkpt \rightarrow cpsave	$\Delta_{\text{checkpt}} = 0.15$
Other messages	$\Delta = 0.15$

The AAA algorithm produces the static schedule shown in Figure 8. The real-time constraint is the completion time of the whole algorithm. Let \bar{S}_1 , \bar{S}_2 , and \bar{S}_3 be the completion times of three processors, F555, ROOT, and R555, *i.e.*, $\text{WCET}(S_1)$, $\text{WCET}(S_2)$, and $\text{WCET}(S_3)$ respectively. The completion time of the whole algorithm is therefore given by Equation (16) below:

$$\bar{S} = \max(\bar{S}_1, \bar{S}_2, \bar{S}_3) \quad (16)$$

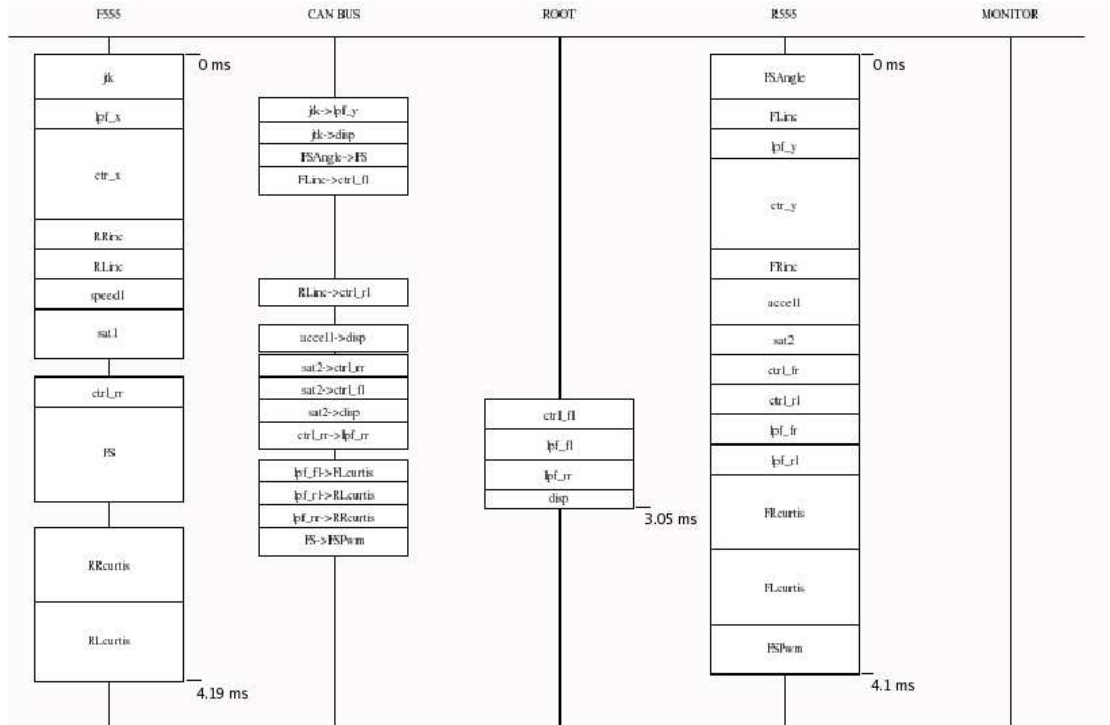


Figure 8: Static schedule created by the SYNDEX tool (completion time=4.19 *ms*).

According to Figure 8, $\bar{S}_1 = 4.19$, $\bar{S}_2 = 3.05$, $\bar{S}_3 = 4.10$, hence $\bar{S} = 4.19$. The period of the algorithm, *i.e.*, the deadline, is set to 10 *ms* in this case study.

7.2 Applying program transformations

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 into the tasks, instead they can be placed between the tasks. For example, according to Table 2, 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, this approach simplifies the transformation while still satisfying the properties. Moreover, checkpoint data to be stored will be much less since

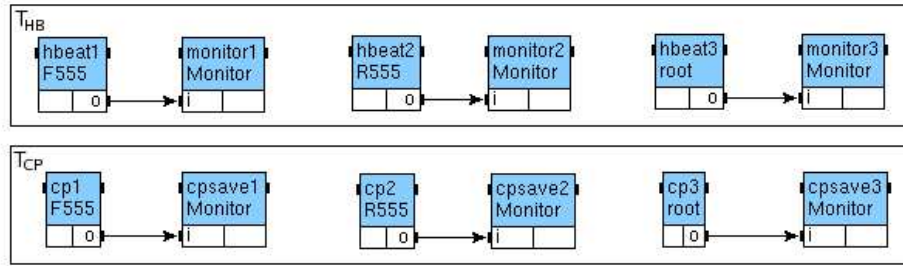


Figure 9: Application graphs for heartbeating and checkpointing. The two algorithms are executed periodically with the periods T_{HB} and T_{CP} respectively.

checkpoints are taken only between the tasks, *i.e.*, internal variables of tasks are not included in the checkpoint data.

For a 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. Let S_1 , S_2 , and S_3 be the programs of processors F555, ROOT, and R555 respectively. For instance, the program of ROOT processor is as follows:

$S_2 = \text{idle time}; \text{ctrl_fl}; \text{lpf_fl}; \text{lpf_rr}; \text{disp};$

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 (5) and (6) can be expressed by Equations (17) and (18) below:

$$\alpha_i < T_{HB_i} + \varepsilon + \xi_i + L_r + L_w \quad (17)$$

$$\beta = T_{HB} + \varepsilon + \max_{1 \leq i \leq n} \xi_i + T_{CP} + L_r + L_w + L_C + \text{WCET}(Det) + \text{WCET}(Rec) \quad (18)$$

where n is the number of processors, Δ_{max} is the transmission time of the longest message and Δ_{hbeat} is the transmission time of `hbeat` message. The reason why the “3” factor in Equation (5) has been removed in Equation (17) is that the tasks `monitori` are not scheduled anymore with a Rate Monotonic policy (implying a complete lack of synchronization with

the tasks `hbeati`), 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 `hbeati` and its corresponding task `monitori`). The reasoning is the same between Equations (6) and (18).

The costs of one `checkpt` and one `hbeat` to the completion time are respectively:

$$\begin{aligned}\bar{c} &= \text{WCET}(\text{checkpt}) + \Delta_{\text{checkpt}} \\ \bar{h} &= \text{WCET}(\text{hbeat}) + \Delta_{\text{hbeat}}\end{aligned}$$

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. Therefore, \bar{h} and \bar{c} are the maximum costs of one `hbeat` and `checkpt`. We compute the completion time of the algorithm in the presence of one failure by Equation (19) below:

$$f(T_{CP}, T_{HB}) = \bar{S} + \frac{\bar{S}_1}{T_{CP}}\bar{c} + \frac{\bar{S}_2}{T_{CP}}\bar{c} + \frac{\bar{S}_3}{T_{CP}}\bar{c} + \frac{\bar{S}_1}{T_{HB}}\bar{h} + \frac{\bar{S}_2}{T_{HB}}\bar{h} + \frac{\bar{S}_3}{T_{HB}}\bar{h} + T_{HB} + T_{CP} + K' \quad (19)$$

Similarly, f is the worst case completion time that may occur only if the initial schedule given in Figure 8 has fully utilized processors and communication bus. 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 completion time is expected to be less than the one given in Equation (19). 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.

Using the partial derivatives, we obtain the optimum values for heartbeating and check-pointing as follows:

$$T_{CP}^* = \sqrt{(\bar{S}_1 + \bar{S}_2 + \bar{S}_3) \times \bar{c}} \quad (20)$$

$$T_{HB}^* = \sqrt{(\bar{S}_1 + \bar{S}_2 + \bar{S}_3) \times \bar{h}} \quad (21)$$

Taking into account the values given in Tables 3 and 4, we find that $\bar{c} = 0.18 \text{ ms}$ and $\bar{h} = 0.21 \text{ ms}$. Therefore,

$$\begin{aligned}T_{CP}^* &= \sqrt{(4.19 + 3.05 + 4.1) \times 0.21} = 1.54 \text{ ms} \\ T_{HB}^* &= \sqrt{(4.19 + 3.05 + 4.1) \times 0.18} = 1.42 \text{ ms}\end{aligned}$$

7.3 Results and discussion

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

```
ckpt ; hbeat ; nop26; hbeat ; nop ; ckpt ; nop25; ctrl_fl; ckpt ; lpf_fl; lpf_rr;
disp; hbeat ;
```

In failure-free operation, the completion time of the new algorithm is 6.21 *ms* as shown in Figure 10. The overhead of the fault-tolerance properties is therefore $6.21 - 4.1 = 2.11$ *ms*.

Thanks to Equation (19), 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.

We finally perform some tests to demonstrate the completion times, when failure recovery is performed by the transformed application algorithm. Figures 12, 13, and 14 show the completion times of the algorithm for 50 iterations, *i.e.*, for 50 different failure times of the processors. The first figure, for instance, presents the completion times against the failures of F555, the second figure for R555 and so on. Processor failures are injected by software at relative failure times that are sampled from the uniform distribution $uniform(0, 70)$ *ms*.

8 Related work

Related work on failure detectors is abundant. On the theoretical side, Fisher et al. 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 communication) 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 which can guarantee consensus on a binary value in finite time [Fisher et al., 1985]. 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 have formalized unreliable failure detectors in terms of completeness and accuracy [Chandra and Toueg, 1996]. 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 present in [Aggarwal and Gupta, 2002] a short survey on failure detectors. They explain the push and pull methods in detail and introduces 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).

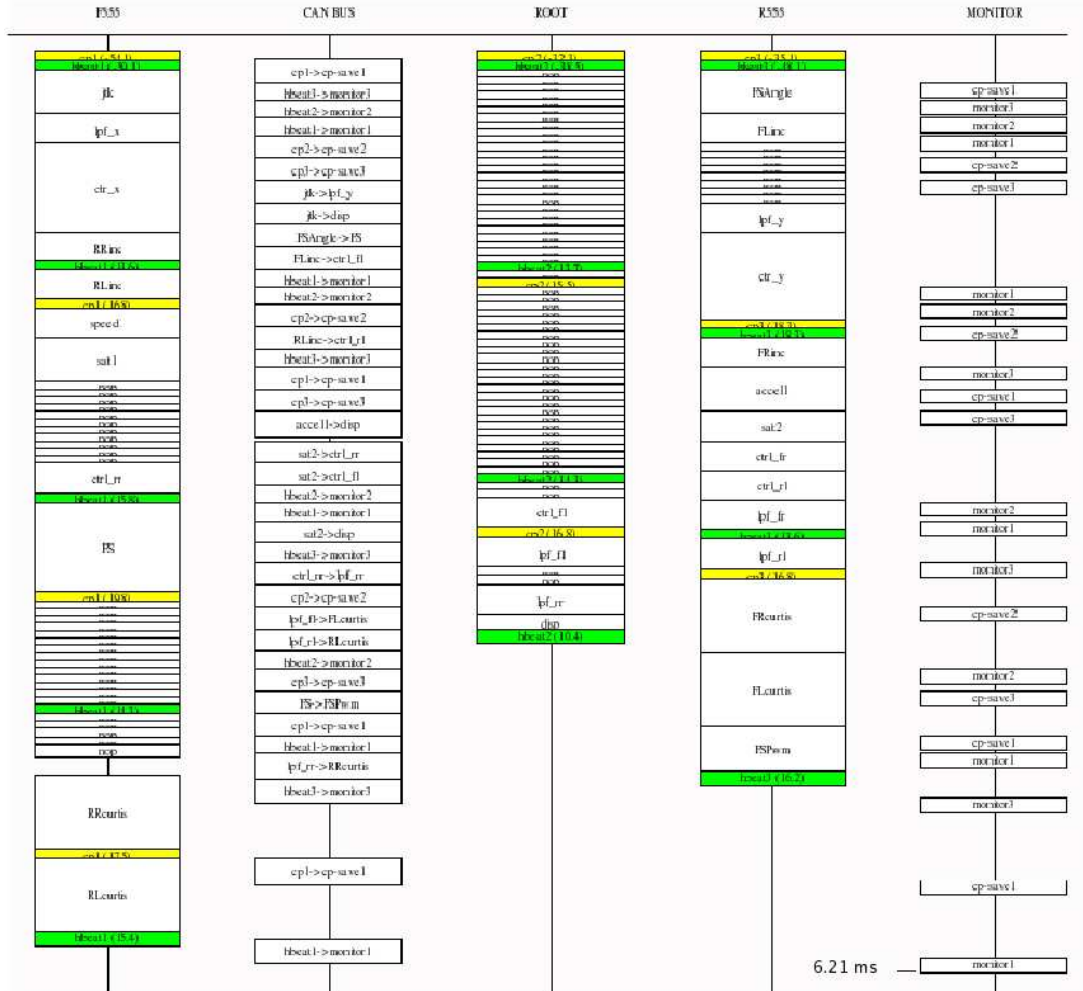


Figure 10: Fault-tolerant static schedule with heartbeating and checkpointing (completion time is 6.21 ms).

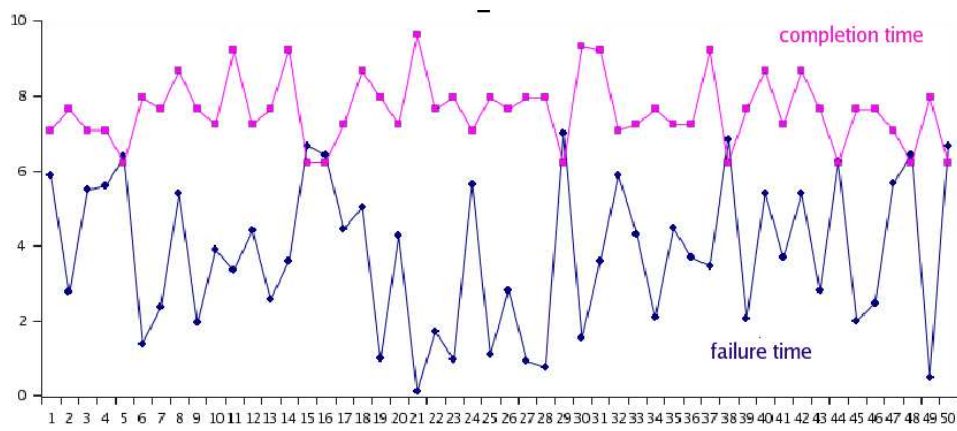


Figure 12: Completion times when processor F555 fails (repeated 50 times).

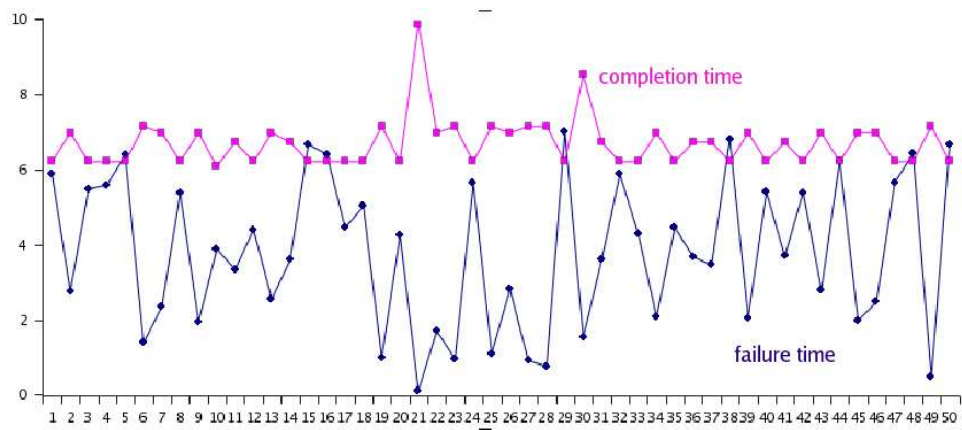


Figure 13: Completion times when processor ROOT fails (repeated 50 times).

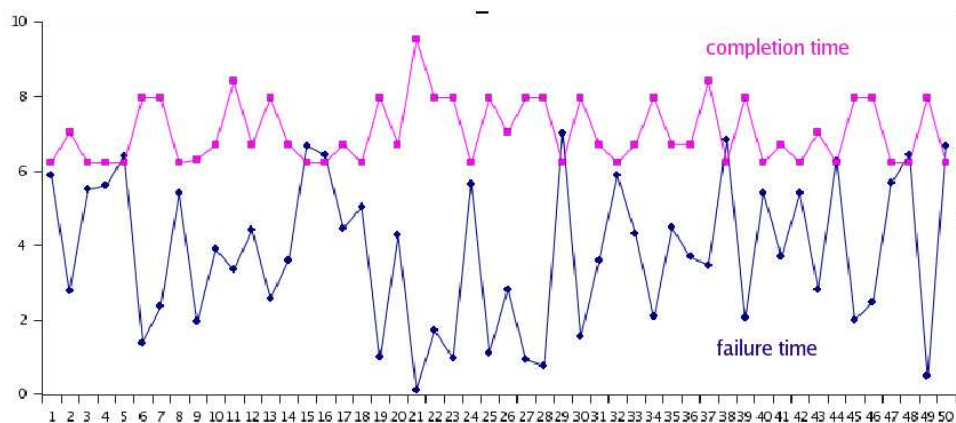


Figure 14: Completion times when processor R555 fails (repeated 50 times).

Other works on failure recovery include the efforts of reserving sufficient slack in dynamic schedule, *i.e.*, gaps between tasks due to the precedence, resources or timing constraints, so that the scheduler can re-execute faulty tasks without jeopardizing the deadline guarantees [Mossé et al., 2003]. Further studies proposed different heuristics for re-execution of faulty tasks in imprecise computation models such that faulty mandatory sub-tasks may supersede optional sub-tasks [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]. Their technique involves synthesizing a fault-tolerant program starting from a fault-intolerant program. A program is a set of states, each state being a valuation of the program's variables, and a set of transitions. Two execution models are considered: high atomicity (the program can read and 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. A fault is a subset of the set of transitions. 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 non-determinism). Such an automatic program transformation is very similar to discrete controller synthesis [Ramadge and Wonham, 1987], a technique that has also been successfully applied to fault-tolerance [Dumitrescu et al., 2004, Girault and Rutten, 2004].

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.

9 Conclusion

In this paper, 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 the 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 due to 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 the same as 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, data upsetting etc. These transformations are seemingly more user dependent, which may lead to the design of aspect-oriented based tools.

References

- [Aggarwal and Gupta, 2002] Aggarwal, A. and Gupta, D. (2002). Failure detectors for distributed systems. Technical report, Indian Institute of Technology, Kanpur, India. <http://resolute.ucsd.edu/diwaker/publications/ds.pdf>.
- [Aguilera et al., 1997] 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*, pages 126–140. Springer-Verlag.
- [Arora and Kulkarni, 1998] Arora, A. and Kulkarni, S. (1998). Detectors and correctors: A theory of fault-tolerance components. In *International Conference on Distributed Computing Systems, ICDCS'98*, pages 436–443, Amsterdam, The Netherlands. IEEE.
- [Avizienis et al., 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. on Dependable and Secure Computing*, 1(1):11–33.
- [Aydin et al., 2000] Aydin, H., Melhem, R., and Mossé, D. (2000). Optimal scheduling of imprecise computation tasks in the presence of multiple faults. In *Real-Time Computing Systems and Applications, RTCSA'00*, pages 289–296.
- [Baille et al., 1999] Baille, G., Garnier, P., Mathieu, H., and Pissard-Gibollet, R. (1999). Le CYCAB de l'Inria Rhône-Alpes. Technical report 0229, Inria, Rocquencourt, France.
- [Beck et al., 1994] Beck, M., Plank, J., and Kingsley, G. (1994). Compiler-assisted checkpointing. Technical report, University of Tennessee.
- [Buttazzo, 2005] Buttazzo, G. (2005). Rate monotonic vs EDF: Judgment day. *Real-Time Systems Journal*, 29(1):5–26.
- [Caspi et al., 1999] Caspi, P., Mazuet, C., Salem, R., and Weber, D. (1999). Formal design of distributed control systems with Lustre. In *International Conference on Computer Safety, Reliability, and Security, SAFECOMP'99*, number 1698 in LNCS, pages 396–409, Toulouse, France. Crisys Esprit Project 25.514.
- [Chandra and Toueg, 1996] Chandra, T. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267.
- [Cristian, 1991] Cristian, F. (1991). Understanding fault-tolerant distributed systems. *Communication of the ACM*, 34(2):56–78.
- [Dean and Shen, 1998] Dean, A. and Shen, J. (1998). Hardware to software migration with real-time thread integration. In *Euromicro Conference*, pages 10243–10252, Västerås, Sweden.

- [Dumitrescu et al., 2004] Dumitrescu, E., Girault, A., and Rutten, E. (2004). Validating fault-tolerant behaviors of synchronous system specifications by discrete controller synthesis. In *IFAC Workshop on Discrete Event Systems, WODES'04*, Reims, France.
- [Fisher et al., 1985] Fisher, M., Lynch, N., and Paterson, M. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.
- [Girault and Rutten, 2004] Girault, A. and Rutten, E. (2004). Discrete controller synthesis for fault-tolerant distributed systems. In *International Workshop on Formal Methods for Industrial Critical Systems, FMICS'04*, ENTCS, Linz, Austria. Elsevier Science.
- [Grandpierre et al., 1999] Grandpierre, T., Lavarenne, C., and Sorel, Y. (1999). Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy. ACM.
- [Grandpierre and Sorel, 2003] 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 *Formal Methods and Models for Codesign Conference, MEMOCODE'03*, Mont Saint-Michel, France.
- [Jalote, 1994] Jalote, P. (1994). *Fault-Tolerance in Distributed Systems*. Prentice-Hall, Englewood Cliffs, New Jersey.
- [Kalaiselvi and Rajaraman, 2000] Kalaiselvi, S. and Rajaraman, V. (2000). A survey of checkpointing algorithms for parallel and distributed computers. *Sadhana*, 25(5):489–510.
- [Kopetz, 1997] Kopetz, H. (1997). *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers.
- [Kulkarni and Arora, 2000] Kulkarni, S. and Arora, A. (2000). Automating the addition of fault-tolerance. In Joseph, M., editor, *6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT'00*, volume 1926 of *LNCS*, pages 82–93, Pune, India. Springer-Verlag.
- [Li et al., 2005] Li, X., Mitra, T., and Roychoudhury, A. (2005). Modeling control speculation for timing analysis. *Real-Time Systems Journal*, 29(1).
- [Liu and Layland, 1973] Liu, C. and Layland, J. (1973). Scheduling algorithms for multiprogramming in hard real-time environment. *Journal of the ACM*, 20(1):46–61.
- [Milner et al., 1990] Milner, R., Tofte, M., and Harper, R. (1990). *The Definition of Standard ML*. MIT Press.
- [Mossé et al., 2003] Mossé, D., Melhem, R., and Ghosh, S. (2003). A nonpreemptive real-time scheduler with recovery from transient faults and its implementation. *IEEE Trans. on Software Engineering*, 29(8):752–767.

- [Nelson, 1990] Nelson, V. (1990). Fault-tolerant computing: Fundamental concepts. *IEEE Computer*, 23(7):19–25.
- [Nielson and Nielson, 1992] Nielson, H. and Nielson, F. (1992). *Semantics with Applications — A Formal Introduction*. John Wiley & Sons.
- [Puschner, 2002] Puschner, P. (2002). Transforming execution-time boundable code into temporally predictable code. In Kleinjohann, B., Kim, K., Kleinjohann, L., and Rettberg, A., editors, *Design and Analysis of Distributed Embedded Systems, DIPES'02*, pages 163–172. Kluwer Academic Publishers.
- [Puschner and Burns, 1999] Puschner, P. and Burns, A. (1999). A review of worst-case execution-time analysis. *Real-Time Systems Journal*, 18(2/3):115–128.
- [Ramadge and Wonham, 1987] Ramadge, P. and Wonham, W. (1987). Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230.
- [Rushby, 2001] Rushby, J. (2001). Bus architectures for safety-critical embedded systems. In *International Workshop on Embedded Systems, EMSOFT'01*, volume 2211 of *LNCS*, Tahoe City, USA. Springer-Verlag.
- [Sekhavat and Hermosillo, 2000] Sekhavat, S. and Hermosillo, J. (2000). The Cycab robot: A differentially flat system. In *IEEE Intelligent Robots and Systems, IROS'00*, Takamatsu, Japan. IEEE.
- [Silva and Silva, 1998] Silva, L. and Silva, J. (1998). System-level versus user-defined checkpointing. In *Symposium on Reliable Distributed Systems*, pages 68–74.
- [Ziv and Bruck, 1997] Ziv, A. and Bruck, J. (1997). An on-line algorithm for checkpoint placement. *IEEE Trans. on Computers*, 46(9):976–985.

10 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 $a_1; \dots a_n$. 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 rules 4

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

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., $BCET(\tau) = WCET(\tau) = EXET(\tau)$), and moreover we have for any τ :

$$\tau \in Traces(S) \implies \begin{cases} BCET(S) \leq EXET(\tau) \leq WCET(S) \text{ and} \\ BCET(S) = WCET(S) \implies EXET(\tau) = EXET(S) \end{cases} \quad (22)$$

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 $EXET(\text{void}) = 0$. The relation \doteq is such that:

$$(\tau_1; \tau_2); \tau_3 \doteq \tau_1; (\tau_2; \tau_3) \quad \tau \doteq (\text{void}; \tau) \doteq (\tau; \text{void})$$

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

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

Property 6 states that 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 $a; c \doteq a; c; \text{void}$, 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:

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

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

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

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 $\forall \tau_2 \in \text{Traces}(\mathcal{I}_c^T(S_2, t - \text{EXET}(S_1)))$, $\tau_2 \doteq S_{2,1};c;S_{2,2} \dots c;S_{2,n}$ ($1 \leq n$) and verifies:

$$\begin{aligned} t - \text{EXET}(S_1) &\leq \text{EXET}(S_{2,1}) < t - \text{EXET}(S_1) + \varepsilon && (\text{Init}) \\ T - \varepsilon < \text{EXET}(c;S_{2,i}) &\leq T + \varepsilon \quad (1 < i < n) && (\text{Period}) \\ r - \varepsilon < \text{EXET}(S_{2,n}) &\leq r \quad \text{If } \text{EXET}(S_2) = t - \text{EXET}(S_1) + q(T - \text{EXET}(c)) + r \\ &&& \text{with } 0 \leq q \text{ and } 0 \leq r < T - \text{EXET}(c) && (\text{End}) \end{aligned}$$

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, $\tau \doteq \tau_1;S_{2,1};c;S_{2,2} \dots c;S_{2,n}$. 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 (22), 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:

$$\begin{aligned} \mathcal{I}_c^T(S_1;S_2, t) &= \mathcal{I}_c^T(S_1, t); \mathcal{I}_c^T(S_2, t_1) \\ \text{with} \quad \text{EXET}(S_1) &= t + q(T - \text{EXET}(c)) + r, \quad 0 \leq q, \quad 0 \leq r < T - \text{EXET}(c) \\ t_1 &= T - \text{EXET}(c) - r \end{aligned}$$

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

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

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, $\forall \tau_2 \in \text{Traces}(\mathcal{I}_c^T(S_2, T - (\text{EXET}(c) + r)))$, $\tau_2 \doteq S_{2,1};c;S_{2,2} \dots c;S_{2,n}$ ($1 \leq n$) and verifies:

$$\begin{aligned} T - \text{EXET}(c) - r &\leq \text{EXET}(S_{2,1}) < T - \text{EXET}(c) - r + \varepsilon && (\text{Init}_2) \\ T - \varepsilon < \text{EXET}(c;S_{2,i}) &\leq T + \varepsilon \quad (1 < i < m) && (\text{Period}_2) \\ r_2 - \varepsilon < \text{EXET}(S_{2,n}) &\leq r_2 \quad \text{If } \text{EXET}(S_2) = T - \text{EXET}(c) - r + q_2(T - \text{EXET}(c)) + r_2 \\ &&& \text{with } 0 \leq q_2 \text{ and } 0 \leq r_2 < T - \text{EXET}(c) && (\text{End}_2) \end{aligned}$$

It follows that:

$$\begin{aligned} & \text{Traces}(\text{Fold}(\mathcal{I}_c^T(\text{Unfold}(\text{for } l = n_1 \text{ to } n_2 \text{ do } S), t))) \\ &= \text{Traces}(\mathcal{I}_c^T(\text{Unfold}(\text{for } l = n_1 \text{ to } n_2 \text{ do } S), t)) \\ &= \text{Traces}(\mathcal{I}_c^T(l := n_1; S; \dots, t)) \end{aligned}$$

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

Proof that Property 6 holds for negative time residuals. For $-\varepsilon < t \leq 0$ 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 $\forall \tau \in \text{Traces}(\mathcal{I}_c^T(S, T - \text{EXET}(c) + t))$, $\tau \doteq S_1; c; S_2 \dots c; S_n$ ($1 \leq n$) and verifies:

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

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

$$c; S_1; c; S_2 \dots c; S_n \doteq \text{void}; c; S_1; c; S_2 \dots c; S_n$$

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

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

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

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

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

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

\square



Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
<http://www.inria.fr>
ISSN 0249-6399