A Model of Parallel Deterministic Real-Time Computation

Matthieu Lemerre Emmanuel Ohayon
CEA, LIST, Embedded Real Time Systems Laboratory
PC 172 - F91191 Gif-sur-Yvette CEDEX, FRANCE
Email: firstname.lastname@cea.fr

***Abstract*—This paper presents a model of computation based on *real-time constraints* and *asynchronous message passing*, and proves a sufficient and necessary condition for this model to be deterministic. The model is then extended with *deterministic error handling*, meaning that the same error yields the same consequences on the system. We consider two different error occurrence models: at a specific time, or at a specific instruction, and conclude that the "error at a specific time" model is more suitable for practical use. We proceed by presenting a concrete implementation of this model in the PharOS real-time system.**

I. Introduction

A system is *deterministic* if its external behavior and output are the same when given the same input[1]. Using a deterministic system has many advantages; in particular, because it allows reproducible computation and limits the number of possible behaviors, it decreases the time needed for debugging and certifying applications.

These arguments have convinced industrial embedded real-time systems developers to use PharOS [1], a deterministic toolchain for real-time systems development. The PharOS model is original in that it combines highly flexible real-time constraints with deterministic parallel computation. It is based on the original OASIS model [2] for deterministic real-time systems running safety-critical applications. While implementing a new feature in PharOS that affected determinism, that allows to shut down a group of tasks in case an error occurs in one of them, it appeared necessary to generalize the computation model of PharOS, in order to provide a formal definition of determinism and prove that determinism could still be achieved in case of errors. The paper presents the results of this work.

The contributions of this paper are:

- an abstract model of computation for parallel real-time tasks that communicate only using asynchronous messages, that generalizes PharOS;
- the precise definition of determinism in that model;
- a theorem that gives sufficient and necessary conditions to enforce determinism in nominal ("no-error") operation;
- sufficient and necessary conditions to enforce determinism for two "error models", that specify how errors occur; and a comparison between these error models;
- a description of the PharOS implementation with the "no error" case and the "error on time" error model.

The paper is organized as follows. Section II presents motivation for developing deterministic real-time systems. Section III introduces the model of computation, and explains how determinism is enforced in nominal operation, and when errors occur. Section IV presents a possible implementation of the model with examples from PharOS. Section V presents the proof of the theorems explained in Section III. Section VI presents related works, and Section VII concludes.

II. Why deterministic real-time systems

Real-time systems are intrinsically parallel: they must perform multiple functions, possibly with several I/Os on different hardware ports, thus requiring different timing constraints. So it is natural to model the system as a set of parallel threads with separate timing requirements. But this parallelism can lead to nondeterministic behavior.

Several authors (e.g. [3],[4]) have argued against the use of nondeterministic programming primitives in parallel programs. In a nutshell, the main arguments are reproducibility and testability, increased debuggability, and general robustness (see related works for more details). This also applies to instrumentation & control real-time embedded systems, which have evolved in size and complexity. Once, when systems were developed on small micro-controllers with a limited number of threads, it was conceivable for a small team to know and understand (almost) all the interactions in a system. Besides, should a rare nondeterministic bug occur, a simple reset watchdog was still the simplest way to handle the problem.

But as hardware performance improved, embedded systems grew in size, exponentially increasing the likelihood for complex race conditions to occur[2] – and with it grew the duration and cost of the debugging and integration steps. We believe that we have reached the point where the potential savings in development costs make it worth the shift to a development methodology that guarantees determinism.

System determinism is of course helpful for integration tests, when fixed input data allows reproducible tests. But

[1]This property should not be confused with the *bounded response time* property.

[2]Multi-core architectures especially tend to trigger these conditions with a much higher probability than single-core chips. Indeed, with a single CPU, a race can only happen when a preemption occurs at the "right" time, a relatively rare event; with a multi-core processor, the same race may potentially occur on every bus access.

determinism also helps even when regularly getting input from the nondeterministic physical world. Indeed, it considerably reduces the number of system behaviors[3], even more so when the system is time-triggered[4]. This means simplified debugging (it is easier to reproduce an error, and decrease the chance of rare race conditions), and simplified qualification and certification (exhaustive checking of system behavior is possible, and determinism provides isolation of behaviors between unrelated tasks that allow composable certification).

Determinism also applies to fault tolerance. For instance many systems detect hardware malfunctions using lockstep, i.e. two processors synchronously executing and comparing the results of the same instruction. With a deterministic system, this redundant execution can be performed by software, with a standard asynchronous multi-core processor. This also applies to other safety configurations, like triple modular redundancy.

How to deal with errors must be chosen according to these applications of determinism. Like inputs, actual errors are unpredictable; what is needed is a way to limit the number of new behaviors created by taking errors into account, simplify reasoning about the consequences of errors, allow reproducible tests and redundant execution. Our choice, in the next section, to make the system deterministic according to the "error on time" error model, fulfills all these requirements.

III. The visibility principle

This section explains how determinism can be preserved in a real-time system composed of parallel tasks that communicate using asynchronous messages, by controlling when the received messages are made accessible to the tasks.

A. Definitions

We assume that the system is composed of parallel tasks (called *agents*), that execute instructions sequentially and atomically. Each instruction must be executed between a release time and a deadline, in a time interval called the *window of execution* (or window for short). In other words, the *actual* instant when the instruction is executed is sometime inside its window[5].

We assume that release time and deadline constraints of all instructions in the system are met[6], i.e. that the execution of the system is *temporally correct*. No assumption is made on the scheduling policy, hence the global order in which instructions are executed in the system is nondeterministic. Note however, that release times and deadlines provide a partial order.

[3]Because tasks typically have small internal states, the system behavior corresponds to the combination of the control flow graphs of the tasks.

[4]For instance, the possibilities of phase difference between jobs in two periodic tasks is finite, contrary to sporadic tasks.

[5]Actual implementations need not to explicitly specify the release time and deadline for each machine instruction. In PharOS (see section IV) each instruction has the same window than the previous one, and the `after`/`before` instructions allow to extend or switch the window.

[6]Failure to comply with a deadline can be considered as an error, see section III-C about error handling.

Agents can communicate using instructions to *send or receive messages*, that are handled by a *communication layer*. When an agent A executes a `receive` instruction, the communication layer can choose to hide a sent message: we say that the message is *invisible* to this instruction. A `receive` instruction *succeeds* if the message was sent and is visible, and fails otherwise. A message can be received by several agents.

In our system, *determinism* is the property that for each agent, the sequence of executed instructions is always the same, i.e. is independent of the global order in which all instructions are executed in the system (i.e. of the agents schedule). The informal definition of determinism is that given the same input, the system produces the same output. Here the input is the initial state of the system, and the output is the sequence of state and instructions executed by each agent.

We assume that for each agent, the outcome of the execution of an instruction is deterministic, and depends only on its local state, and on the messages received so far[7] As messages are the only mean of communication between agents, proving that the whole system is deterministic is reduced to proving that message reception is deterministic.

To summarize, given individual agents that execute deterministically, the paper gives the sufficient and necessary conditions on message passing to ensure that the composition of these agents is deterministic. These conditions are all variations of the same *visibility principle*: *an agent A shall be able to see a received message at time t only if the message would have been received at t in every possible execution*. In other words, if there exists one temporally correct schedule of the agents in which A could not receive the message, then the message should never be visible to A, for any schedule.

The complete mathematical model, and proof of the results of this section, is given in section V. In this section we focus on the practical applications of these results.

B. The no-error case

When agents are *live* (i.e. there is no error and the agents never stop executing), the visibility principle translates to: *when the window of the instruction that sends a message and the window of an instruction that tries to receive it overlaps, the message remains invisible, i.e. this receive instruction always fails*.

In this case, *determinism is achieved if, and only if, this principle is applied*.

This theorem can be understood informally (see Figure 1). When the windows do not overlap, success in reception of the message is independent of the actual sending and receiving times within the windows. So nondeterminism arise only when the windows overlap. To preserve determinism in this latter case, the receive instruction must always fail, i.e. the communication layer must make the message invisible.

[7]Thus if an instruction reads an input from the outside world, the outcome of this input instruction must not depend on the actual instant when the instruction is executed within the window.

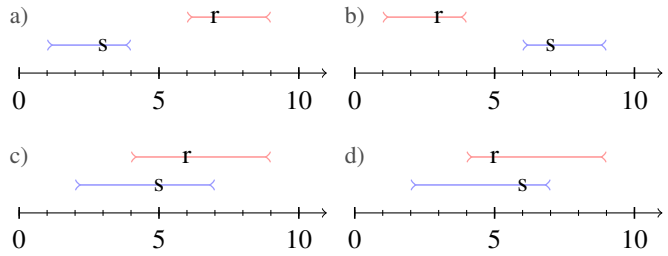

Figure 1. A `send` (s) and `receive` (r) instructions with their execution windows in two concurrent agents. In the upper cases, the windows do not overlap, and the message will be deterministically always received (upper left) or never received (upper right). In the lower cases, these windows do overlap, and reception nondeterministically depends on the actual instants when the `send` and `receive` instructions are executed. The visibility principle states that determinism is restored if, and only if, the message is invisible when these windows overlap (e.g. the system does as if no message was received in the lower left case).

Note that this is a sufficient and necessary condition: there is no other way to ensure determinism when agents communicate by asynchronous sending of messages.

C. Dealing with errors

This first result relied on the hypothesis that agents were live (i.e. never stop executing). We now relax this hypothesis to take into account errors scenarios that can terminate an agent or a set of agents. Errors may affect determinism: if an agent may be nondeterministically stopped before having sent a message, this affects nondeterministically agents that try to receive this message.

The definition of determinism states that given identical inputs, the system should have the exact same behavior. As we now consider errors as admissible inputs of the system, we want to ensure that the system behaves identically when errors occur in "identical conditions".

However, there are several ways to define "identical" error conditions. We consider two of them, which we believe are the most suitable for practical use:

- either the error always occurs at a specific time (*Error on Time* - ***EoT***);
- or the error always occurs when executing a specific instruction of an agent (*Error on Instruction* - ***EoI***).

We show that with both of these *error models*, there is a way to ensure determinism.

1) Determinism with the EoT model: In this first error model we assume that an error occurs at a given time, that will terminate some of the agents. This is a source of nondeterminism, because the sending of a message depends on whether the message could be sent *before* the error occurred, or not. Therefore this issue raises only when the error occurs in the middle of a `send` window. To restore deterministic execution for the other agents, messages sent in such case must be made invisible.

With this error model, the visibility principle states that *to achieve determinism, if the error occurs during a* `send` *window, the corresponding message must be invisible*. This is a necessary condition. In addition, if the visibility principle of section III-B is met, then the system is deterministic (sufficient condition). Figure 2 illustrates this principle.

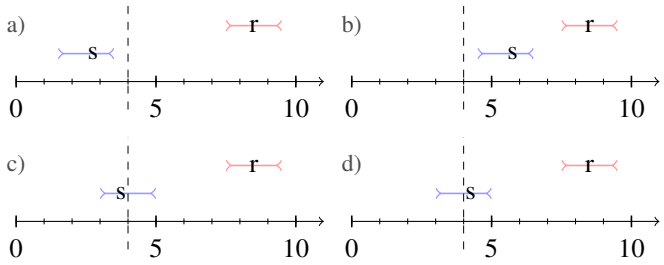

Figure 2. Sending and reception in two concurrent agents, in case there is an error at time 4 that terminates the sending agent. In the upper cases, the message will deterministically be either always sent (upper-left figure) or never sent (upper-right). If the time of error is in the interval when sending can happen (lower cases), then the sending of the message is nondeterministic. Determinism is restored if, and only if, the message is invisible when such crossing happens (e.g. the system does as if no message was received in the lower left case).

2) Determinism with the EoI model: We now assume that the error is caused by the failure of an agent A executing a faulty instruction. The corresponding window is called "error window" in the following. If A just stops executing, then the system is still deterministic. However the system becomes nondeterministic if A belongs to a "group" ([1]), and its failure causes the termination of all other agents in the group. Indeed, sending of a message by another agent A' in the group would then depend on the actual instants when the error happens in A and the sending happens in A'.

As before, determinism for the other agents can be restored if messages sent in such nondeterministic cases are always made invisible. The new visibility principle states that *the system is deterministic if, and only if, when a* `send` *window and an error window overlap, the message remains invisible. In addition, the visibility principle of section III-B must be met*. Figure 3 illustrates this principle.

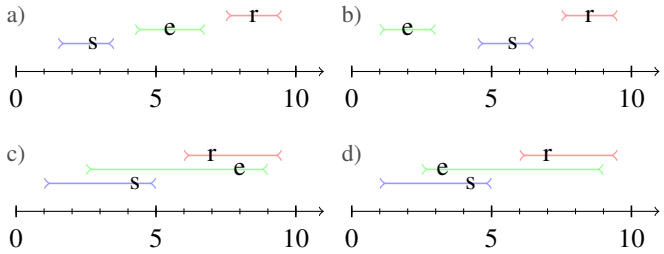

Figure 3. Sending and reception in three concurrent agents, in case there is an error in one agent that terminates the sender. In the upper cases, the message will deterministically be either always sent (upper-left figure) or never sent (upper-right). If the error window overlaps the `send` window (lower cases), then reception of the message is nondeterministic. Determinism for the receiving agent is restored if, and only if, the message is ignored during the overlapping windows, meaning that the `receive` instruction fails in both lower cases.

3) Practical comparison of error models: Errors do not happen by accident: they are generally caused by a mistake during the execution of an agent. This could lead to think that the "*Error on Instruction (EoI)*" model is best suited. For instance, if the programmer knowingly introduces an instruction that causes an error (e.g. for testing resilience to errors), this error model describes exactly what happens, whereas the "*Error on Time (EoT)*" does not, because we do not know the exact instant when the error can happen.

But in practical use, it is impossible to predict which instruction will cause an error and when (or else the error could be corrected). Furthermore, the EoI model is ill-suited to simulate errors other than software bugs, e.g. hardware errors. What the model should provide is simplified reasoning about the consequences of an error, by minimizing the number of cases to consider.

Flaws of the "Error on Instruction (EoI)" model: With this model, one has to consider that any instruction may cause an error, a fact that is not known until the instruction has been executed. An instruction can formally be considered as executed only once its deadline has passed, so it is not safe to make a message visible as long as the window of the instruction that sends this message overlaps the window of any instruction of any other agent belonging to the same group.

For instance in Figure 3.c, when r is executed the system does not yet know if the e instruction will cause an error. Thus in both the 3.c and 3.d cases, the system must do as if e would cause an error (by making the message invisible), even if no error happens.

Thus this error model leads to changes in the visibility rules, even when there is no error. These new visibility rules cause three main problems:

1) it is much more difficult for the programmer to foresee whether a receive instruction will succeed, since that depends on all the agents of the group;
2) the behavior of the system differs from the "error-free" model when no error occurs;
3) it may increase the message sending latency. For instance if an instruction in a group has a very long window, no message can be received from that group for the whole duration of the window.

These problems make the EoI model unsuitable for practical use.

Practical use of the "Error on Time (EoT)" model: The EoT model does not suffer from these problems. In practical execution, the "instant of error" is just the current time when an error is detected. Thus when an agent tries to receive a message, it is easy to know if there was an error in the group of the sending agent, and when (Section IV presents the concrete implementation in PharOS). If there was no error, the message will be visible (provided of course that the corresponding `send` window has passed), which is compatible with the "error-free" model.

The EoT model reduces the number of cases that must be considered on error, as shows the example on Figure 4.

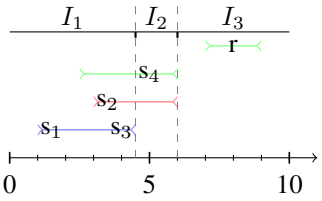

Figure 4. Sending of three agents in the same group, and a receiving agent in a different group, in the "error at a specific time" model. Here the agent on the lowest line sends two messages with the same window, and the receiving agent tries to receive all four messages. There are only three cases to consider for r, depending on the time interval in which the error happens (I_1: no message, I_2: s_1 and s_3, I_3: all four messages). By comparison if the visibility principle was not applied, there would be 12 different combinations of messages that could be received by r, depending on the execution order.

All the practical uses of determinism we listed in section II are covered by the EoT model. We already said that it reduces the number of behaviors. It allows easy exhaustive and reproducible tests, for instance by setting up an agent to fail on purpose in each interval of interest. It also allows duplication of an error in case of redundant execution (by communicating the recorded instant of error between the execution sites).

IV. Implementation of determinism with examples from PharOS

A. Introduction to PharOS

PharOS [1] is a framework to design, implement and execute safety-oriented embedded real-time applications. It relies on the time-triggered paradigm [5], and implements a multi-task model where each task may use a different, dynamically varying time scale [6].

1) The ΨC language: PharOS applications are written in ΨC, an extension of the C language that provides keywords to define real-time agents, timing constraints, and inter-agents communications. In ΨC, release times and deadlines are given explicitly using the `after` and `before` statements, with the following semantics:

- when an `after(t)` statement is encountered, execution of the agent cannot continue until time t is reached;
- when a `before(t)` statement is encountered, the time t must not have been reached yet – or it is a deadline error.

Both `before` and `after` statements are translated by the ΨC compiler into system calls instructions that ask the PharOS kernel to change the execution windows. The kernel then schedules and monitors the agents based on the windows.

The execution flow of each agent is data-dependent: for instance the next release time or deadline can be chosen according to a condition on a hardware input. [6] gives more details about timing constraints specification in PharOS.

We represent the execution trace of an agent on a timeline, where `after` constraints are symbolized by ▶, and `before` constraints by ◀.

2) Explicit visibility date: We call *visibility date of a message* the minimum among the release times of all instructions that could successfully receive the message. The visibility principles requires that this visibility date is at least the deadline of the instruction that sends the message, but it can be any later date. In PharOS, the visibility date of the messages is explicit and independent of the timing constraints of the sending and receiving agents; various mechanisms ensure that the visibility date is always later than the deadline of the sending agent. Explicit visibility dates provides a *temporal interface*, that decouples the sending of a message from the time it can be received.

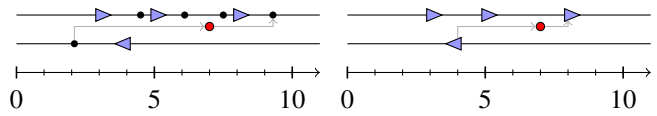

Figure 5. Sending a message in the PharOS model. Here the message is sent at time 2, with a deadline of 4 and a visibility date of 7. The message is received only by receive instructions with a release time later or equal to the visibility date.

On the timeline representation (Figure 5), a message is represented by • at the visibility date. We could represent sending and receiving instructions at their actual instant of execution (left figure), but these instants play no role in the execution. Instead (right figure), one can view the execution as *formally sending* the message at the deadline of the send instruction, and *formally receiving* it at the first release time encountered after the visibility date.

3) PharOS communication primitives: PharOS implements different variants of the general communication mechanism described before. The `message` communication primitive is a direct implementation of this mechanism: the visibility date is given as an argument to the `send` instruction that sends the message. The only difference is that the send instruction also acts as a `before`, whose deadline is the visibility date: this ensures that the message is always sent before the visibility date. This primitive is well suited to handle sporadic communications.

The *temporal variable* communication primitive (Figure 6) implements a periodic data flow: a new message is periodically emitted by the producer of the temporal variable. By default, if the sending agent does not explicitly modify the flow, the same value is re-emitted.

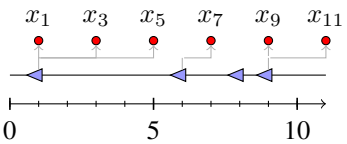

Figure 6. A temporal variable $(x_{2t+1})_{t\in\mathbb{N}}$, of phase 1 and period 2. The messages x_1, x_3 and x_5 are identical and carry the value written by the sender at time 1; the message x_7 carries the value written at time 6; the messages x_9 and x_{11} carry the value written at time 9.

Although these communication primitives have fairly different implementations, their underlying principles remain the same as those extensively described in section III. Therefore for the rest of this paper we will only consider the sporadic `message` mechanism used so far, as it is the most general.

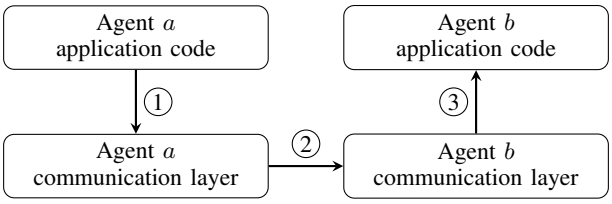

Figure 7. The three steps of message transit. The communication layer can be seen as a protected library to the application code, while transfer between the communication layers of two agents happens concurrently.

B. Implementation of the communication mechanism

The implementation of PharOS communication primitives are variations of the following generic algorithm:

1) The message is copied from application buffers to the communication layer. The visibility date is either given by the sender, or computed by the communication layer. The communication layer ensures that the visibility date of the message is always later than the current deadline of the sending instruction, and stores the message and its visibility date in a "sending area".
2) The message is copied from the sending area to a "receiving area", in the communication layer of the receiving agent. This copy occurs when the receiving agent is awakened from a `after(t)` instruction (i.e. when time t was reached). Only messages whose visibility date is earlier than t are retrieved.
   Note that steps 1) and 2) may imply concurrent accesses to the sending area[8].
3) The agent retrieves the message from its receiving area.

Special care has to be taken in case a recipient receives messages from multiple senders. The order in which the communication layer returns messages to the agent must be deterministic, i.e. independent of the actual time when the messages were sent. Nondeterminism would arise if, for instance, a simple shared FIFO was used to store the messages.

In PharOS, the "message" communication primitive returns messages according to the following lexical order:

- Messages with the earliest visibility date are shown first;
- Messages with equal visibility dates are sorted by IDs of the sender agent;
- Messages with equal visibility dates sent by the same agent are sorted with a "last in first out" policy, meaning the last message sent is the first shown to the receiver.

There are many other total orders that would also provide determinism.

[8]The various implementations of this mechanism in PharOS are all lock-free, to avoid tampering with scheduling.

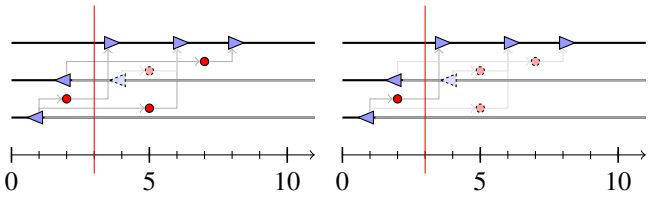

Figure 8. Comparison on the execution with the two choices. On the left, all messages sent by instruction whose deadline is sooner than the error date are visible. As a result, some surprising situations can happen: for instance at time 5 the receiving agent tries to receive 2 messages with the same visibility date, but only one is received. At time 7 it receives a message from an agent, while it did not receive a previous message from that same agent. On the right, only messages whose visibility is below the error date of the group are visible, and these problems are avoided.

C. Taking errors into account

1) Messages made invisible by an error: We saw earlier that PharOS allows the visibility dates of messages to be later than the deadline of the corresponding `send` window. What should happen when an agent sends a message visible at time t_v, with a deadline of $t_d < t_v$, and there is an error at time t_e with $t_d < t_e < t_v$?

Both choices (the message is received, or the message is not received) are deterministic. But we believe that not showing the messages is the best solution for a variety of reasons:

- The visibility date provides a temporal interface between several agents. If the visibility of a message depends on the deadline of the sending agent, rather than the visibility date, it breaks this temporal interface, as it forces the recipient to know temporal details of the sender.
- The implementation is simpler, because we already store the visibility date of the messages.
- It is simpler to send messages "atomically", where messages from several agents are either all received or none. In the first case, the messages must be sent with the same deadline, in the second they must be sent with the same visibility date, and the latter is simpler to achieve.
- The behavior is simpler to understand. Figure 8 presents various situations that can be surprising for the programmer. Not showing the messages fulfills the following principle: if a receiver can receive a message of visibility date d, then other messages with visibility earlier or equal to d can also be received. This simplifies reasoning about the consequences of errors.

Thus, in PharOS we chose to implement the second choice.

2) Implementation: The implementation requires only a small modification to step 2 of the implementation of the communication mechanisms. The algorithm is:

- Let a_r be the receiving agent, t_r its current release time
- For all agents a_s that can send messages to a_r:
  - If a_r is in a group that was not terminated:
    - Retrieve all messages with visibility t_v such that $t_v \leq t_r$.
  - Else:
    - Let t_e be the error date of the group.
    - Retrieve all messages with visibility t_v such that $t_v < t_e$ and $t_v \leq t_r$.

In addition, if an agent is restarted after an error of date t_e, all messages with visibility $t_v \geq t_e$ must be removed from the sending area before the agent can report that it is active again. Else invalid messages could be retrieved after the agent has restarted.

Note: for communication primitives such as the temporal variable, that always provide a value periodically, "removing a message", or not retrieving it from the sending area, is implemented by marking the temporal variable value as invalid; thus a receiving agent can know that a temporal variable value was produced while the sending agent was down.

D. Enforced determinism

To enforce determinism, providing deterministic communication mechanisms does not suffice; they must be the only possible communication channels between agents. This is achieved using memory protection.

In PharOS, every agent is protected using static generation of hardware memory tables (depending on the target hardware, this can be a MMU or MPU). These tables allow agent applicative code to access only their private data, and a small memory mapping shared with the communication layer of this agent (this mapping is used to send and receive messages, i.e. corresponds to steps 1 and 3 of the message transit).

In particular, agents do not have direct access to other agents memory (even read-only), or to any system data that may change during their execution. The only observable data that changes during an agent execution is changed by the agent itself; this enforces determinism.

Thus, even if the implementation of an agent has a bug and reads random pointers, this error will be deterministic, and thus reproducible. In particular, this avoids spending time debugging errors that may be particularly hard to reproduce (or impossible to reproduce when the debugger is active).

V. Sufficient and necessary conditions on message visibility for system determinism

Defining determinism requires a model of computation that represents the parallel execution of the agents in the system. This model, presented first, is general enough to let agents represent any sequential computation, and limits only the nature of the interactions between agents. Then we proceed with the full proof of determinism in the "no-error" case, and sketch the proof (which is similar) when errors may occur.

A. Presentation of the computation model

1) Execution of a single agent: The model represents concurrently executing *agents*. An agent has a current *state*, and sequentially executes *instructions* that update that state.

There are three kinds of instruction:

- The *update* instruction changes the current state of the agent s to a new state $s' = f(s)$, for some function f.

- The $send(m)$ instruction sends a message m and updates the state to $f(s)$, for some function f;
- The $receive(m)$ instruction updates the state to $s' = f(s, has\_received?(m))$ conditionally, depending on whether the message has been received, for some function f. The $has\_received?$ function returns true iff. the message was previously sent (using a $send(m)$ instruction with the same m) and is *visible*. We do not give a definition for *visible*[9], and instead give the minimum conditions on *visible* for the system to be deterministic.

In each case, f is a deterministic function; hence the only source of nondeterminism in the execution of an agent comes from the reception of a message.

Given a state s, $next(s)$ returns the next instruction to be executed, and $exec(next(s), s)$ computes the next state. The *execution trace of an agent* is a sequence of states, whose first element is the initial state s_0, and each further element represents one step in execution: $s_{n+1} = exec(next(s_n), s_n)$.

2) System execution and determinism: The *system execution trace* represents the concurrent execution of several agents. The instructions executed by each agent are interleaved, but each instruction is executed atomically.

A system execution trace X is a sequence, whose elements are vectors of state representing the current state of each agent. X_0 contains the initial state $<s_0^{a_1}, s_0^{a_2}, ..., s_0^{a_k}>$ of the k agents of the system, and the next element represent one step in the execution of one agent: $X_{n+1} = X_n$ except for one agent a, whose state s^a_{n+1} is $exec(next(s^a_n), s^a_n)$.

We denote by $X_{0..n}$ the $n+1$ first elements of a system execution trace X. $X_{0..n}$ is also a system execution trace.

If a is an agent, we denote by X^a the execution trace of the agent a in the system execution trace X. X^a is extracted from X by keeping only the elements in X for which the state of a changes, and by having the elements in X^a contain only the state of a instead of the vector of states of all agents.

3) Timing constraints: To each instruction i is associated a *release time* and a *deadline*. Informally, this means that i must be executed between its release time and deadline. But in fact, there is no need to introduce a notion of real-time in the model. The only needed rule is that *if i and j are two instructions in an execution trace X, (possibly executed by different agents), if the deadline of i is earlier or equal to the release time of j, then i must be executed before j in X.*

This timing constraints rule provides a partial order for the execution of instructions in a system execution trace, and thus informally "introduces some determinism" to the system (i.e. allows to guarantee that some instructions will always be executed before some others).

It is easy to see that this rule implies that if i is executed before j by an agent, and j has a deadline $d_j < d_i$, then any instruction k that must be executed after j will also be executed after i. Thus the model would be identical if we considered that i had for deadline d_j. In general, the deadlines of an agent can be put in increasing order without changing the expressivity of the model[10].

In the following, we only consider execution traces that fulfill the timing constraint rule, and whose agents execute instructions with increasing deadlines.

4) Determinism: We say that the system is *deterministic for an agent a*, when for all execution trace X that have the same initial system state X_0, the execution trace for a X^a is unique (it is independent of the execution trace X).

We says that the system is *deterministic for a until n*, when given an initial system state X_0, the first $n+1$ elements of every execution trace Y whose first element is X_0 are a prefix of a common execution trace for a X^a:

$$\forall (X_0, a),\ \exists X^a,\ \forall Y : Y_0 = X_0, \quad (Y_{0..n})^a \sqsubseteq X^a$$

Intuitively this means that the beginning of all executions are the same for a, but the number of executed instructions of a vary (depending on the order of execution of the agents).

We say that the system is *deterministic* (resp. *until n*) when it is deterministic (resp. *until n*) for all agents.

The proof that if a system is deterministic until n for all n, then it is deterministic, is trivial.

B. Determinism when agents are live

In this section we assume that agents are live, i.e. for every execution trace X and agent a, X^a is infinite.

In this case, determinism comes from the combination of two separate properties, used to structure the proof. Lemma 5.1 shows that timing constraints provide a partial order, that guarantees deterministic reception in some cases. Lemma 5.2 introduces conditions on visibility that remove the remaining nondeterministic behaviors. Theorem 5.3 completes the proof, by induction on the length of execution traces.

1) Deterministic subset:

For any time t, integer n and execution trace X, we define by $Before(t, X_{0..n})$ the set of instructions executed in the first $n+1$ steps of X whose deadlines are earlier than t.

Lemma 5.1: Let X be an execution trace, i an instruction in X, n the index in X in which i is executed, and t_i the release time of i. We assume that the system is deterministic until n. Then for all Y such that i has been executed in $Y_{0..(n+1)}$, $Before(t, Y_{0..n})$ is constant (i.e. independent of Y) and equal to $Before(t, X_{0..n})$.

Proof: Let Y be an execution trace such that $Y_{0..(n+1)}$ executes i, and a an agent. As the system is deterministic until n, one of $(X_{0..n})^a$ and $(Y_{0..n})^a$ is a prefix of the other.

Either $(X_{0..n})^a \sqsubseteq (Y_{0..n})^a$, and all instructions executed by a in $Before(t, X_{0..n})$ are in $(Y_{0..n})^a$.

Or $(Y_{0..n})^a \sqsubset (X_{0..n})^a$. Let j be the first instruction in $(X_{0..n})^a$ which is not in $(Y_{0..n})^a$ (i.e. it is the next instruction to be executed by a in Y).

[9] Indeed several definitions are suitable; for instance the visibility date in Section IV-A, used to provide a temporal interface, makes messages invisible for longer than is necessary for the system to be deterministic. Another suitable definition for *visible* is to never make the messages visible.

[10] This is also true for the release times; see [6, Theorem 2].

Either j is never executed, but this is impossible because a is live. As j is executed after i, following the timing constraint rule (Section V-A3), the deadline of j is strictly later than t_i. As deadlines are increasing, any later instruction executed by a in X will have a deadline later than t_i. Thus, each instruction executed in X^a whose deadline is earlier than t_i has also been executed in $(Y_{0..n})^a$.

Thus: $Before(t, (X_{0..n})^a) \subseteq (Y_{0..n})^a$. As instructions of deadline earlier than t in $(X_{0..n})^a$ have the same deadline in $(Y_{0..n})^a$, we have:

$$Before(t, (X_{0..n})^a) \subseteq Before(t, (Y_{0..n})^a).$$

Because of the symetric role of X and Y, we also have $Before(t, (Y_{0..n})^a) \subseteq Before(t, (X_{0..n})^a)$, and finally: $Before(t, (Y_{0..n})^a) = Before(t, (X_{0..n})^a)$.

As this is true for every agent a, this proves that:

$$Before(t, Y_{0..n}) = Before(t, X_{0..n})$$

■

2) Visibility:

Lemma 5.2: We assume that the system is deterministic until n for some n. Let X be an execution trace that executes i, a $receive(m)$ instruction, in the nth step.

If messages sent with a deadline later or equal to the release time of i are not visible to i, then reception of m is independent of the execution trace.

Proof: Let t be the release time of the i instruction. The message is visible only if it has been sent in an instruction with a deadline earlier than t, i.e. if it is in $Before(t, X_{0..n})$. But thanks to Lemma 5.1, we know that $Before(t, Y_{0..n})$ is constant for all execution traces Y that have executed i in $Y_{0..(n+1)}$. Thus there are only two possible cases:

- Either there is a $send(m)$ instruction in $Before(t, X_{0..n})$, and thus in all execution traces Y that have executed i in $Y_{0..(n+1)}$, and the message will always be received by i;
- Else there is no such instruction in $Before(t, X_{0..n})$, and the message will never be received by i.

In any case, the reception of m is independent of the execution trace. ■

Note the role of *visible* in this proof, which is to rule out $send(m)$ instructions in $X_{0..n}$ that are not in $Before(t, X_{0..n})$, and thus are not present in all execution traces.

3) Proof of determinism:

Theorem 5.3: If messages sent with a deadline later or equal to the release time of a receiving instruction i are not visible to i, then the system is deterministic.

Proof: The proof is by induction. The induction hypothesis is that the system is deterministic until n.

For $n = 0$, this is true: the initial state X_0, given as input, is the same for all execution traces by hypothesis.

Now we suppose that the system is deterministic until n. Let X be an execution trace, a the agent being executed on the nth step, s_a the current state of a and $i = next(s_a)$ the next instruction to be executed.

It suffices to prove that the state of a after i executes, is always the same. Indeed the induction hypothesis implies that the system is deterministic until n for all agents. For all agents $a' \neq a$, $(X_{0..n+1})^{a'}$ will not change, and stay prefixes of a common sequence. For a we know that $(X_{0..n})^a$ is a prefix of a common sequence, as will be $(X_{0..n+1})^{a'}$ if the outcome of executing i is always the same.

Now i is either an $update$, $send$ or $receive$ instruction. If i is an $update$ or $send$ instruction, the next state only depends on s_a, the current state of a, which is unique by the induction hypothesis, so it will also be unique. If i is a $receive$ instruction, its execution depends on s_a and the fact that a has received some message m. But this has been proved independent of the execution trace in Lemma 5.2.

Thus in every case, the state resulting of i execution will always be the same. Thus the system is deterministic until $n + 1$. ■

4) Necessary condition for determinism: What we proved so far was that the visibility rule was sufficient for the system to be deterministic. Now we prove that this rule is necessary.

Theorem 5.4: Determinism implies that messages sent with a deadline later or equal to the release time of a receiving instruction i are not visible to i.

Proof: Let X be an execution trace for which an instruction i_r successfully receives a message m, that has been sent by an instruction i_s in another agent with deadline later than the release time of i_r.

Then it is easy to construct a new execution trace X' for which i_s is after i_r, and that still fulfill the timing constraints. In that case the execution in X of the agent that executes i_r would diverge[11] from its execution in X. ■

C. Determinism when there are errors

In this section we study an "error in a time interval" (EiTI) model, where the actual time of error is constrained between a release time and a deadline. The error is inserted in a system execution trace as a regular instruction; in particular the timing constraints rules of Section V-A3 apply.

We choose this model because it generalizes the "error on time" model (that constrain the interval of errors to be a point). It is also close, but less general, than the "error on instruction" model (it can be seen as a special case where errors are caused only by special agents with only one error instruction), while avoiding some of the difficulties of that model[12].

For brevity and readability sake, instead of writing the complete proof in the case of EiTI errors, we will only explain the difference with the case without errors.

[11] Actually this requires that the state after execution of i_r depends on the reception of the message, i.e. determinism is a necessary condition only if the receive instructions actually "care" about the message. The model and/or definition of determinism could be changed to require that.

[12] In the true "error on instruction" model the $error$ instruction is undistinguished from the others. Two main problems arise: first, the list of errors is not known before the beginning of execution, which complexifies the proof. Second, stopping an agent on error causes the execution of some instructions to be nondeterministic, including $error$ instructions.

1) Changes to the model: There is now a fourth special instruction, $error(\mathcal{A})$, which takes as a parameter a set of agents. A new constraint on the system execution traces requires that once an $error(\mathcal{A})$ instruction has been executed, the states of the agents in $\mathcal{A}$ cannot be updated.

Agents are no longer all live, but either live, or stopped by an $error$ instruction. We say that an agent a is *known to be live at time* t when no $error$ instruction of release time earlier than t stops a.

The definition of *determinism for an agent until* n is still the same, but the definition of *determinism for an agent* must be changed, because we cannot avoid the fact that the exact instruction where an agent is stopped may vary between executions. The new definition states that for all agents, the execution trace is a prefix of a common X^a:

$$\forall(X_0, a),\ \exists X^a,\ \forall Y : Y_0 = X_0, \quad Y^a \sqsubseteq X^a$$

This new definition is thus weaker than the old definition (where the execution trace was equal to X^a, and X^a was unique), and closer to the definition of determinism for an agent until n. The definition of *system determinism* as determinism for all agents is still the same.

2) Changes to the visibility conditions and proof: The visibility conditions change to also remove messages sent by instruction overlapping an error, and the new theorem for sufficient condition of determinism becomes:

Theorem 5.5: If for a receiving instruction i of release time t, neither the messages sent with a deadline later or equal to t, nor the instructions sent by agents not known to be live at t, are visible to i, then the system is deterministic.

Proof: The proof structure is very similar to the case when the agents are live.

The new version of Lemma 5.1 no longer states that $Before(t, X_{0..n})$ is constant; but instead that for all agents a known to be live at t, that $Before(t, (X_{0..n})^a)$ is constant (and the proof is similar);

The new version of Lemma 5.2 is similar, but is updated to take into account the new visibility conditions (the proof is not changed);

And the proof of Theorem 5.3 is also not changed. ■

Proving that these visibility conditions are also necessary is also similar: starting from an execution trace that does not meet the visibility conditions, reorder the instructions so that the two executions of the receiving agent diverge.

VI. Related works

The recent increase of the use of multicore hardware has increased the interest in determinism and its benefits: testability, avoiding rare bugs[4], replay-based debugging [7][8], fault-tolerance, security, and intrusion analysis [9][10].

Lee has proved[4] that the the usual shared-memory thread model is harmful for determinism, as it actually implements an intractable computation model. Subsequently, several authors ([4],[3]) promote the idea that deterministic models should be the default approach for parallel programming, be it through dedicated languages, software runtime, or hardware support.

The hardware-oriented DMP approach [11] aims at enforcing determinism for *any* multi-threaded program by systematically serializing concurrent memory accesses, and authorize parallelism only on communication-free instructions chunks. The authors also propose a software implementation of their mechanism, but the performance tradeoff makes this solution unsuitable for low-power embedded systems. Other works such as the Determinator OS [12], the dOS Linux extension [13] or user-mode schedulers such as Kendo [14] and Tern [15] implement general-purpose deterministic execution environments. These solutions provide more or less POSIX-compliant APIs, and are well suited for adapting lock-based legacy threaded programs; they do not offer however a high-level framework to conceive and design a deterministic communicating real-time system.

Another approach is to abandon completely the legacy shared-memory thread model in favor of a less expressive but more formal one, showing "good" properties, such as offline scheduling, and allowing to guarantee the absence of deadlock. If CSP [16] or Petri Nets [17] are not deterministic, Kahn Process Networks [18] and Lee's Synchronous Data-Flow model [19] are[13]. These models have been successfully extended and implemented with dedicated languages such as StreamIt [20] or ΣC [21]. However these models are designed for data-intensive and not control-intensive applications, and lack a notion of real-time.

The Time-Triggered Architecture (TTA) [5][22] is a components framework for designing real-time dependable applications, that relies on the time-triggered paradigm like PharOS. It includes however a complete hardware architecture and a Network-on-Chip deterministic communication protocol between components, whereas PharOS aims to be executed on COTS (Components Off The Shelf). As it also aims to be platform-independent, Giotto's approach [23] is the closest to PharOS. It is a specification language made to design real-time applications composed of parallel periodic tasks, which may be turned on and off dynamically (mode switching). Giotto aims at separating concerns for *reactivity* and *schedulability*, which is also the key idea of the PharOS ΨC language. However, ΨC offers a higher temporal expressiveness [6], as it allows to define tasks whose timing behaviour is not periodic. Besides, Giotto implements only synchronous communication between tasks, which brings it closer to fully-synchronous reactive languages such as Esterel [24] or Lustre [25], whereas in the PharOS model the communications are asynchronous.

VII. Conclusion

In this paper we have formalized the model of computation implemented in PharOS, based on real-time constraints and asynchronous messages, and proved that it is deterministic. We have then extended this model of computation to allow errors

[13]The SDF model is a restriction of KPNs.

that can shut down some tasks in the system, and extended the visibility conditions on messages to maintain determinism in that case. We have compared two models of error, "error on time" and "error on instruction", and concluded that the "error on time" model is best suited for practical applications. Finally, we have presented a simple practical implementation of these theoretical concepts inside PharOS, yielding a deterministic real-time system that can handle faults deterministically.

Let us stress that the concepts presented above are independent of the scheduling algorithm. Thus they can be implemented on multicore systems. We would like to take advantage of this to implement software redundant fault-tolerant execution on a standard multicore.

Shutting down agents may be desired on other conditions than errors; for instance to implement "modes" of execution, such as "take-off", "cruise" and "landing" modes in an avionics system. We would like to extend the use of deterministic shut down of agents in PharOS to handle mode changing.

These concepts also work in distributed systems, the only requirement being that nodes use a common time reference. It would be interesting to have a distributed version of PharOS that would handle failure of an agent in a node or a failure of a complete node deterministically.

Finally, determinism ultimately depends on the correctness of the system algorithms; in particular that they implement the visibility conditions correctly, and are independent of the order in which messages are sent. We have started a long-run work to obtain a machine-checked proof of the correctness of these algorithms, which are parallel lock-free system code, using the TLA specification language.

REFERENCES

[1] M. Lemerre, E. Ohayon, D. Chabrol, M. Jan, and M. Jacques, "Method and tools for mixed-criticality real-time applications within PharOS," in *Proceedings of the 14th International ISORCW Symposium (AMICS Workshop)*. IEEE, 2011, pp. 41–48.

[2] C. Aussaguès, C. Cordonnier, M. Aji, V. David, and J. Delcoigne, "OASIS: a new way to design safety critical applications," in *21st IFAC/IFIP Workshop on Real-Time Programming (WRTP'96), Gramado, Brazil*, 1996.

[3] R. Bocchino Jr, V. Adve, S. Adve, and M. Snir, "Parallel programming must be deterministic by default," in *Proceedings of the First USENIX conference on Hot topics in parallelism*. USENIX Association, 2009, pp. 4–4.

[4] E. Lee, "The problem with threads," *IEEE Transactions on Computers*, vol. 39, no. 5, pp. 33 – 42, 2006.

[5] H. Kopetz and G. Bauer, "The time-triggered architecture," *IEEE Special Issue on Modeling and Design of Embedded Software*, January 2003.

[6] M. Lemerre, V. David, C. Aussaguès, and G. Vidal-Naquet, "An introduction to time-constrained automata," *EPTCS: Proceedings of the 3rd Interaction and Concurrency Experience Workshop (ICE'10)*, 2010.

[7] T. LeBlanc and J. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Transactions on Computers*, vol. 100, no. 4, pp. 471–482, 1987.

[8] P. Montesinos, L. Ceze, and J. Torrellas, "DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently," in *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*. IEEE, 2008, pp. 289–300.

[9] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen, "Execution replay of multiprocessor virtual machines," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2008, pp. 121–130.

[10] A. Joshi, S. King, G. Dunlap, and P. Chen, "Detecting past and present intrusions through vulnerability-specific predicates," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 91–104, 2005.

[11] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "DMP: deterministic shared memory multiprocessing," in *ACM Sigplan Notices*, vol. 44, no. 3, 2009, pp. 85–96.

[12] A. Aviram, S. Weng, S. Hu, and B. Ford, "Efficient system-enforced deterministic parallelism," in *9th OSDI*. USENIX Association, 2010, pp. 1–16.

[13] T. Bergan, N. Hunt, L. Ceze, and S. Gribble, "Deterministic process groups in dOS," *9th OSDI*, 2010.

[14] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: efficient deterministic multithreading in software," in *ACM Sigplan Notices*, vol. 44, no. 3. ACM, 2009, pp. 97–108.

[15] H. Cui, J. Wu, C. Tsai, and J. Yang, "Stable deterministic multithreading through schedule memoization," *9th OSDI*, 2010.

[16] C. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[17] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981.

[18] G. Kahn, "The semantics of a simple language for parallel programming," *proceedings of IFIP Congress74*, vol. 74, pp. 471–475, 1974.

[19] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[20] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *Compiler Construction*. Springer, 2002, pp. 49–84.

[21] T. Goubier, R. Sirdey, S. Louise, and V. David, "ΣC: A programming model and language for embedded manycores." in *ICA3PP (1)*, ser. Lecture Notes in Computer Science, vol. 7016. Springer, 2011, pp. 385–394.

[22] C. Paukovits and H. Kopetz, "Building encapsulated communication channels in the time-triggered system-on-chip architecture," *Research Report*, vol. 8, 2009.

[23] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: A time-triggered language for embedded programming," in *Embedded Software*. Springer, 2001, pp. 166–184.

[24] G. Berry, "The foundations of Esterel," *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pp. 425–454, 2000.

[25] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.