Proving Determinacy of the PharOS Real-Time Operating System*

Selma Azaiez[1], Damien Doligez[2], Matthieu Lemerre[1],
Tomer Libal[3], and Stephan Merz[4,5]

[1] CEA, Saclay, France
[2] Inria, Paris, France
[3] Inria, Saclay, France
[4] Inria, Villers-lès-Nancy, France
[5] CNRS, Université de Lorraine, LORIA, UMR 7503, Vandoeuvre-lès-Nancy, France

Abstract. Executions in the PharOS real-time system are deterministic in the sense that the sequence of local states for every process is independent of the order in which processes are scheduled. The essential ingredient for achieving this property is that a temporal window of execution is associated with every instruction. Messages become visible to receiving processes only after the time window of the sending message has elapsed. We present a high-level model of PharOS in TLA$^+$ and formally state and prove determinacy using the TLA$^+$ Proof System.

1 Introduction

The outcome of an execution of a concurrent system depends not only on the inputs provided from the system's environment, but also on the relative order in which the system's processes are scheduled for execution. This order is largely unpredictable, especially when the system executes on parallel hardware; it introduces an element of non-determinism even when every process behaves deterministically. Testing and debugging of concurrent systems is therefore challenging and involves so-called "Heisenbugs" that are very difficult to reproduce.

For real-time systems, such as controllers of safety-critical components in planes or cars, designers are very reluctant to admit systems that exhibit non-deterministic behavior. Fortunately, it is possible to design concurrent real-time systems such that their behavior does not depend on the order of scheduling, as long as all components have access to a common time base. This hypothesis can be satisfied in local networks of embedded systems. For example, the PharOS real-time system [9, 10], commercialized[6] under the name Asterios®, has been designed to ensure that system executions do not depend on the scheduling order of processes. The core idea is to associate every instruction that some process wishes to execute with a temporal window of execution and to ensure that a

* This work was supported by the French BGLE Project ADN4SE. It was also partly funded by the Microsoft Research-Inria Joint Centre, France.

[6] `http://www.krono-safe.com`

message sent from one process to another can be received only if the execution window of the receiving instruction is strictly later than that of the sending instruction. Consider two executions that execute both the sending and the receiving instructions according to the given temporal constraints, then the message will either be received in both executions, or in neither of them. This argument is at the core of the determinacy proof for the PharOS model of execution [9].

In this work, we formally specify a high-level model of PharOS executions in the specification language TLA$^+$ [6] and use TLAPS, the TLA$^+$ Proof System [4], to formally prove determinacy of our model. Our proof is based on the paper-and-pencil proof of [9]. In contrast to that proof, TLA$^+$ proofs such as ours must be written in assertional style, i.e., based on inductive invariants, rather than making explicit references to different states of an execution. Moreover, the mere statement of determinacy is not entirely obvious in a linear-time framework such as TLA$^+$ because the property refers to the equivalence of different executions, whereas formulas of linear-time temporal logic are expressed in terms of a single, implicit execution.

Our work reinforces the confidence in the result that PharOS executions are indeed deterministic and makes explicit some hypotheses that were implicit in the original proof. It represents a significant case study for TLAPS and has also contributed several lemmas that are now included in the standard library of the TLAPS distribution.

Outline. Fundamental concepts of PharOS and of TLA$^+$ are presented in Section 2. Sections 3 and 4 are the core of the paper and describe the formal model of PharOS and the proof of the main theorem. We conclude in Section 5.

2 Background

2.1 PharOS

The PharOS model of execution [9, 10] is based on the OASIS model [2, 12] but relaxes the constraints on the precise instants of execution of individual processes. It was designed with the objectives of ensuring predictability of execution and of supporting formal reasoning. Avoiding race conditions between processes is crucial for achieving the first objective. It is assumed that the system consists of a fixed number of tasks (called *agents*), each of which executes instructions sequentially and atomically. All agents share a common time reference, and each instruction is associated with a fixed, non-empty time window of execution in the sense that it must be executed between an earliest and latest time instant (cf. Fig. 1). The only assumptions that are made about the scheduling policy for the agents are that the specified time windows are respected, and that deadlines are never missed. The latter property is in practice ensured by schedulability analysis of the implementation, different from the techniques discussed in the present paper [8].

Agents communicate with one another exclusively by asynchronous message passing. It can be seen (and follows from our proof) that determinacy of the

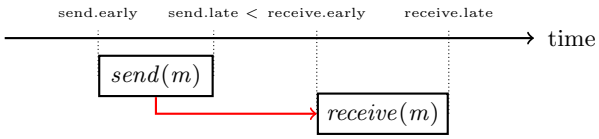

Fig. 1. Time constraints associated with instructions in PharOS.

execution model can therefore be reduced to determinacy of message reception. In order for a message m to be received by an agent, it is not enough that the corresponding send instruction was executed prior to reception: it must also be *visible* to the receiving instruction. As indicated in Fig. 1, the message can be visible only if the time window associated with the send instruction precedes the time window of the receiving instruction, i.e. the latest possible instant at which the message can be sent precedes the earliest instant at which it can be received. Hence, if there exists a schedule in which the message was not sent prior to the execution of the receiving instruction, the message cannot be received in any execution.

2.2 The TLA$^+$ Specification Language

TLA$^+$ [6] is a formal specification language that is mainly intended for modeling concurrent and distributed algorithms and systems, and that has successfully been used in academic and industrial environments [3, 7, 13]. It is based on untyped Zermelo-Fraenkel set theory for modeling the data manipulated by the system, and on the Temporal Logic of Actions, a variant of linear-time temporal logic, for describing executions. Formulas of temporal logic serve for specifying system behavior as well as properties of systems. Systems are modeled as state machines. In particular, the system state is represented by a tuple of variables. State predicates (i.e., first-order formulas containing state variables) represent sets of system states, such as the initial condition or system invariants. Transition predicates, also called *actions*, are first-order formulas that contain both ordinary (unprimed) and primed occurrences of state variables; they describe state transitions where unprimed variables denote the value in the first state and primed variables denote the value in the second state. For example, $x' = x - y'$ is true of any pair $\langle s, t\rangle$ of states such that the value of x in state t equals the difference between the values of x in state s and y in state t. The canonical form of the safety part of a system specification in TLA$^+$ is a temporal formula of the form

$$\mathit{Init} \wedge \Box[\mathit{Next}]_{\mathit{vars}}$$

where *Init* is a state predicate constraining the initial states, *Next* is an action that describes all possible system steps, and *vars* is a tuple of all variables used to represent the state of the system. Temporal formulas are evaluated over infinite sequences of states, and the above formula is true iff *Init* is true of the first state

and all pairs of subsequent states either leave *vars* unchanged or satisfy *Next*. Fairness conditions can be added to the above formula in order to ensure liveness properties, but they play no role in this paper.

If P is a state predicate then P' denotes a copy of P in which all state variables x have been replaced by their primed counterparts x'. It is a transition formula that asserts that P is true of the second state of the pair of states. For example, the familiar proof obligation requiring that an invariant *Inv* be preserved by the system's next-state relation is written as the formula

$$Inv \wedge [Next]_{vars} \Rightarrow Inv'.$$

Sets and functions are central in TLA$^+$ for modeling systems and their data. Semantically, every TLA$^+$ value is a set. The notations for standard set-theoretic constructions are familiar. In particular, $\{\, e(x) : x \in S \,\}$ and $\{\, x \in S : p(x) \,\}$ are two forms of set comprehension: the first one denotes the set of values $e(x)$ for all elements x of set S, and the second one the subset of elements of S that satisfy predicate p. Also, SUBSET S and UNION S denote the powerset (set of all subsets) of S and the union of the elements of the family S of sets. A function f is a total mapping over its domain DOMAIN f, function application is written $f[e]$, and $[x \in S \mapsto e(x)]$ denotes the function with domain S such that $f[x] = e(x)$ for all $x \in S$. The set $[S \to T]$ denotes the set of functions f with domain S such that $f[x] \in T$ for all $x \in S$.

An n-tuple (or sequence of length n) $d = \langle d_1, \ldots, d_n \rangle$ is a function with domain $1\,..\,n$ such that $d[i] = d_i$ for all $i \in 1\,..\,n$. The set $Seq(S)$ denotes the set of all finite sequences whose elements are contained in the set S. Standard operations on sequences include *Append* (adding an element at the end of a sequence) and *Head* and *Tail* for accessing the first element and the remainder of a non-empty sequence. The predicate $IsPrefix(s, t)$ holds if s is a prefix of the finite sequence t.

Records are represented as functions whose domains are finite sets of strings. For example, $[id : String,\ bal : Int]$ denotes the set of records with two fields *id* and *bal* whose values are respectively a string and an integer. For such a record *accnt*, the fields are accessed as *accnt.id* and *accnt.bal* (short-hand forms for *accnt*["id"] and *accnt*["bal"]). The expression $[id \mapsto$ "xyz", $bal \mapsto 123]$ denotes a record in the above set.

For writing long formulas, TLA$^+$ adopts the convention of writing multi-line conjunctions and disjunctions as lists "bulleted" with $\wedge$ and $\vee$, and where indentation is used instead of parentheses to indicate precedence. For example,

$$\begin{array}{l} \wedge\ A \vee B \\ \wedge\ \vee\ C \\ \quad\ \vee\ D \\ \wedge\ E \Rightarrow F \end{array}$$

is a conjunction of three formulas, the first and second of which are disjunctions.

2.3 TLA$^+$ Modules

TLA$^+$ specifications are structured as modules. A module declares parameters (for example, the set of processes of a multi-process algorithm), defines operators, and may state assumptions and theorems about the parameters and operators. In fact, standard integer and real arithmetic, as well as the operations on sequences mentioned above, are not part of the language itself, but are defined in modules of the standard library. Modules can be imported using the EXTENDS keyword, which corresponds to copying the contents of the imported module into the current module. A more elaborate form of import is provided through module instantiation, which allows substituting expressions for module parameters.

We define several modules that provide operations used in our specification. The module *Streams* defines an infinite sequence over a set S as a function from positive integers to S and provides several theorems about streams. In order to give a flavor of TLA$^+$, an excerpt of the module is shown here, omitting the proofs.[7]

MODULE *Streams*

EXTENDS *NaturalsInduction*, *Functions*, *SequenceTheorems*

$Natp \triangleq Nat \setminus \{0\}$

$Stream(S) \triangleq [Natp \to S]$

$take(w, n) \triangleq Restrict(w, 1 \,..\, n)$

LEMMA *takeStream* $\triangleq$

ASSUME NEW S, NEW $w \in Stream(S)$, NEW $n \in Nat$

PROVE $take(w, n) \in Seq(S) \land Len(take(w, n)) = n$

LEMMA *takeStreamMonotonic* $\triangleq$

ASSUME NEW S, NEW $w \in Stream(S)$,
NEW $m \in Nat$, NEW $n \in Nat$, $m \leq n$

PROVE $IsPrefix(take(w, m), take(w, n))$

Another module introduces the operation $filter(s, a)$. It takes as its first argument s a finite sequence of system states (cf. set *SystemState* introduced in section 3 below); each system state is a record whose *st* field is an array containing the local states of all agents. The second argument of *filter* is an agent a. The operation projects s to the sequence of local states of agent a, and then removes finite repetitions of states: when s corresponds to a prefix of a system execution, repeated agent states typically correspond to steps where some other agent than a was executed. Its formal definition is thus given as the composition of a projection operator and another operator for removing finite stuttering:

$$project(s, a) \triangleq [\, i \in 1 \,..\, Len(s) \mapsto s[i].st[a] \,]$$

$$unstutter[s \in Seq(State)] \triangleq$$

[7] The standard module *Functions* defines the domain restriction of a function as $Restrict(f, S) \triangleq [x \in S \mapsto f[x]]$.

$$
\begin{array}{l}
\quad \text{IF } \mathit{Len}(s) \leq 1 \text{ THEN } s \\
\quad \text{ELSE IF } \mathit{Last}(s) = s[\mathit{Len}(s) - 1] \text{ THEN } \mathit{unstutter}[\mathit{Front}(s)] \\
\quad \text{ELSE } \mathit{Append}(\mathit{unstutter}[\mathit{Front}(s)],\ \mathit{Last}(s)) \\
\mathit{filter}(s, a) \ \triangleq \ \mathit{unstutter}[\mathit{project}(s, a)]
\end{array}
$$

The definition of *unstutter* illustrates the definition of recursive functions in TLA$^+$. We prove many facts about these operations that are used in our main proof, including those listed below.

$$
\begin{array}{l}
\text{LEMMA } \mathit{filter\_range} \ \triangleq \\
\quad \text{ASSUME NEW } H \in \mathit{Seq}(\mathit{SystemState}),\ \text{NEW } a \in \mathit{Agent} \\
\quad \text{PROVE } \ \mathit{Range}(\mathit{filter}(H, a)) = \{H[i].st[a] : i \in 1 \mathinner{..} \mathit{Len}(H)\} \\
\text{LEMMA } \mathit{filter\_IsPrefix} \ \triangleq \\
\quad \text{ASSUME NEW } H1 \in \mathit{Seq}(\mathit{SystemState}),\ \text{NEW } H2 \in \mathit{Seq}(\mathit{SystemState}), \\
\qquad \text{NEW } a \in \mathit{Agent},\ \mathit{IsPrefix}(H1, H2) \\
\quad \text{PROVE } \ \mathit{IsPrefix}(\mathit{filter}(H1, a), \mathit{filter}(H2, a)) \\
\text{LEMMA } \mathit{IsPrefix\_filter} \ \triangleq \\
\quad \text{ASSUME NEW } H1 \in \mathit{Seq}(\mathit{SystemState}),\ \text{NEW } H2 \in \mathit{Seq}(\mathit{SystemState}), \\
\qquad \text{NEW } a \in \mathit{Agent},\ \mathit{IsPrefix}(\mathit{filter}(H1, a), \mathit{filter}(H2, a)) \\
\quad \text{PROVE } \ \exists H \in \mathit{Seq}(\mathit{SystemState}) : \wedge\ \mathit{IsPrefix}(H, H2) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \mathit{filter}(H, a) = \mathit{filter}(H1, a)
\end{array}
$$

2.4 Tool Support for TLA$^+$

The formal verification of TLA$^+$ specifications is supported by the TLC model checker and by TLAPS, the TLA$^+$ Proof System. TLC [16] is an explicit-state model checker that can verify properties of finite instances of TLA$^+$ specifications. Similar to the ProB model checker [11], TLC is notable for its capability to evaluate a highly expressive, set-based expression language.

For analysis with TLC, parameters of TLA$^+$ modules must be instantiated by fixed values (such as instantiating the set of processes to the set $\{1, 2, 3\}$). Additionally, it must be ensured that all values that variables take during any execution of the specified system belong to some finite set. This is not always possible: for example, a finite instance of a system may have an unbounded state space when communication channels are represented by unbounded sequences and messages may be resent. In such cases, analysis may be restricted to states satisfying a user-defined constraint. While this implies an under-approximation of the analyzed state space, any counter-example produced by TLC within the restricted search space is an actual system execution. Going further, the user may also override definitions that TLC either cannot evaluate or that lead to unbounded state spaces. In these cases, the semantics can be changed arbitrarily, and the significance of the results of verification must be ensured by the user, but skilled use of these features helps building confidence in the specification. In this project, we have mainly used TLC for validating definitions of complex operators such as the *filter* operation shown in Section 2.3, where it helped us to catch off-by-one and similar errors.

TLAPS [4] is an interactive proof assistant for TLA$^+$. It allows users to develop proofs for lemmas and theorems asserted in a TLA$^+$ module, using a hierarchical proof language. These proofs are interpreted by the core of TLAPS, called the *proof manager*. Obligations corresponding to the steps in the proof are sent to automatic proof backends, including first-order provers, SMT solvers, and a decision procedure for propositional temporal logic. If no backend is able to prove the step, the user can write a more detailed, lower-level proof of the step. Proofs for the different steps can be developed in any order, which lets users concentrate on the most difficult or interesting part of a proof first and fill in details later. All proof obligations whose proof was already attempted during the current project are stored in a data base, and TLAPS allows users to quickly check the status of the proof and assess the impact of changes in definitions or assertions.

Both TLC and TLAPS are accessed from the TLA$^+$ Toolbox, an Eclipse-based GUI for editing TLA$^+$ specifications. The ability to use the same specifications for model checking and for proof is very valuable for validation. In particular, TLC can be used to check if an assertion appearing in a proof can be invalidated in a finite instance, before making a futile proof attempt.

3 A High-Level Model of PharOS in TLA$^+$

Our objective in specifying PharOS in TLA$^+$ is to provide a high-level model that abstracts from choices made in particular implementations. In particular, we do not wish to commit to any scheduling policy, nor fix the time taken by individual instructions, or indeed the set of instructions that an instance of PharOS executes. We thus obtain a highly non-deterministic specification that is intended to encompass all possible system executions.

A TLA$^+$ module representing the static model of PharOS appears in Fig. 2. PharOS coordinates executions of processes, called agents, each of which has local states. Correspondingly, the constants *Agent* and *State* are declared as parameters in the module. We assume that every state s identifies the instruction $instrOf(s)$ that the agent will execute next. There are three kinds of instructions: local instructions simply modify the local state of an agent by applying an update function. Send instructions similarly update the local state but are also tagged with a message identifier and inserted into the message pool. Receive instructions attempt to retrieve the message[8] with the given identifier from the message pool and apply an update to the local state whose effect depends on whether the message could be received.

PharOS is a real-time system, and time is discrete, represented by natural numbers. In particular, the instructions manipulated by PharOS are equipped with a temporal execution window that indicates the earliest and latest points in time when the instruction can be executed. The functions updating the local states are assumed to be monotonic in the sense that the execution window of

[8] For simplicity, messages are identified with the sending instruction.

MODULE *Types*

EXTENDS $Sequences, Streams, TLAPS$

CONSTANT $Agent, State, initState, instrOf(\_), MsgId, Update, visible(\_,\_)$

$Time \triangleq Nat$

$Instruction \triangleq [type : \{\text{“local”}\}, upd : Update]$
$\cup [type : \{\text{“send”}\}, msg : MsgId, upd : Update]$
$\cup [type : \{\text{“receive”}\}, msg : MsgId, bupd : [\text{BOOLEAN} \to Update]\ ]$

$DatedInstruction \triangleq \{\ x \in [ins : Instruction, early : Time, late : Time] :$
$x.early < x.late\ \}$

$Message \triangleq \{\ di \in DatedInstruction : di.type = \text{“send”}\ \}$

ASSUME $instrOf\_type \triangleq \forall s \in State : instrOf(s) \in DatedInstruction$

$SystemState \triangleq [st : [Agent \to State], msgs : \text{SUBSET } Message, t : Time]$

$GoodNextTime(s, a, t) \triangleq \land\ s.t \leq t$
$\land\ instrOf(s.st[a]).early \leq t$
$\land\ \forall b \in Agent : t \leq instrOf(s.st[b]).late$

$GoodSystemState \triangleq \{\ s \in SystemState :$
$\exists a \in Agent,\ t \in Time : GoodNextTime(s, a, t)\ \}$

ASSUME $goodInitState \triangleq$
$[\ st \mapsto initState, msgs \mapsto \{\}, t \mapsto 1\ ] \in GoodSystemState$

ASSUME $Update\_type \triangleq Update \in \text{SUBSET } [State \to State]$

ASSUME $Update\_monotonic \triangleq \forall state \in State, upd \in Update :$
$\land\ instrOf(state).early \leq instrOf(upd[state]).early$
$\land\ instrOf(state).late \leq instrOf(upd[state]).late$

ASSUME $visible\_cond \triangleq \forall di, dj \in DatedInstruction :$
$\land\ visible(di, dj) \in \text{BOOLEAN}$
$\land\ visible(di, dj) \Rightarrow di.late < dj.early$

$msgReceived(msgs, di) \triangleq \exists i \in msgs : di.ins.msg = i.ins.msg \land visible(i, di)$

$exec(state, msgs) \triangleq$
LET $i \triangleq instrOf(state)$
IN IF $i.ins.type = \text{“update”}$ THEN $\langle i.ins.upd[state], msgs\rangle$
ELSE IF $i.ins.type = \text{“send”}$ THEN $\langle i.ins.upd[state], msgs \cup \{i\}\rangle$
ELSE $\langle i.ins.bupd[msgReceived(msgs, i)][state], msgs\rangle$

$executes(a, pre, post) \triangleq$
LET $pstate \triangleq pre.st[a]$
$res \triangleq exec(pstate, pre.msgs)$
IN $\land\ GoodNextTime(pre, a, post.t)$
$\land\ post.st = [\ pre.st \text{ EXCEPT } ![a] = res[1]\ ]$
$\land\ post.msgs = res[2]$

ASSUME $TimeProgress \triangleq$
$\forall H \in Stream(SystemState) :$
$(\forall n \in Natp : \exists a \in Agent : executes(a, H[n], H[n+1]))$
$\Rightarrow \forall T \in Time : \exists k \in Natp : H[k].t > T$

Fig. 2. Static model of PharOS.

the instruction associated with the updated state cannot precede the execution window of the instruction for the original state. As specified in the definition of operator *msgReceived*, a message can be received if it is present in the message pool and if it is *visible*, which implies that the execution window of the sending instruction precedes the execution window of the receiving instruction.

The operator *exec* corresponds to the result of executing the pending instruction of the local state in the context of the message pool passed as the arguments. It returns a pair consisting of the updated state and the new message pool. Note that in our model, nothing is assumed about the delivery order of messages, and that a message may be received multiple times.

A *system state* is represented as a record with three fields corresponding to the array of local states per agent, the messages that have been sent, and the current time. Given system state s and agent a, the predicate $GoodNextTime(s, a, t)$ identifies time instants t at which a can take a step without any deadline being missed. More precisely, it holds if t is at least as big as both the time recorded in system state s and the earliest execution time for the instruction that a is about to execute, but does not exceed the time window of the pending instruction of any agent b (including a itself). A *good system state* is one for which some such t exists, for some agent a. The initial states of every agent are given by parameter *initState*, and we assume that the system state formed by this state assignment, an empty message pool, and initial time 1 is good.

The predicate $executes(a, pre, post)$ is true if the system state *post* can be obtained from the system state *pre* by an execution of agent a at a good next time. It is easy to prove that whenever *pre* is a good system state then $executes(a, pre, post)$ implies that *post* is also good. As expressed by predicate *TimeProgress*, we assume that in any infinite sequence of system states such that every transition corresponds to the execution of some agent, the recorded time progresses beyond any bound. This corresponds to the familiar non-Zenoness assumption in the analysis of real-time systems [1].

Figure 3 presents the system specification of PharOS. The state of the system is represented by the variables *state*, *messages*, and *time*.[9] Additionally, variable *history* is used for the specification of determinacy; it records the sequence of all previous states of the system. The overall system behavior is specified in standard form as formula *Spec*. The definition of the initial condition is obvious. The next-state relation requires that time advances, without missing any deadline, to some value within the execution window for the instruction of some agent a, and that the state and message pool are updated according to the execution of that instruction. Moreover, the new system state is appended to the sequence *history*.

We start by proving that the predicate *TypeOK* is indeed an invariant of specification *Spec*, i.e. that $Spec \Rightarrow \Box TypeOK$ is valid. Moreover, we prove the following invariants.

[9] Alternatively, we could have used a single variable and represented the system state as a record in set *SystemState*.

```
MODULE System_Spec
EXTENDS Types, Filters
VARIABLES state, messages, time, history
vars ≜ ⟨state, messages, time, history⟩
TypeOK ≜ ∧ state ∈ [Agent → State]
         ∧ messages ∈ SUBSET Message
         ∧ time ∈ Time
         ∧ history ∈ Seq(SystemState) \ {⟨⟩}
Init ≜ ∧ state = initState ∧ messages = {} ∧ time = 1
       ∧ history = ⟨[st ↦ state, msgs ↦ messages, t ↦ time]⟩
Next ≜
  ∧ time′ ∈ {t ∈ Time : t ≥ time}
  ∧ ∀a ∈ Agent : time′ ≤ instrOf(state[a]).late
  ∧ ∃a ∈ Agent :
      ∧ instrOf(state[a]).early ≤ time′
      ∧ LET res ≜ exec(state[a], messages)
        IN  ∧ state′ = [state EXCEPT ![a] = res[1]]
            ∧ messages′ = res[2]
  ∧ history′ = Append(history, [st ↦ state′, msgs ↦ messages′, t ↦ time′])
Spec ≜ Init ∧ □[Next]_vars
```

Fig. 3. Dynamic model of PharOS executions.

```
Inv ≜ ∧ Last(history) = [st ↦ state, msgs ↦ messages, t ↦ time]
      ∧ ∀i ∈ 1 .. Len(history) : history[i].t ≤ time
      ∧ ∀a ∈ Agent : time ≤ instrOf(state[a]).late

MsgInv ≜ LET instr(a) ≜ {instrOf(history[i].st[a]) : i ∈ DOMAIN history}
             sends(a) ≜ {di ∈ instr(a) : di.ins.type = "send"}
             toSend ≜ UNION {sends(a) : a ∈ Agent}
             expired ≜ {m ∈ toSend : m.late < time}
         IN  ∧ messages ⊆ toSend
             ∧ expired ⊆ messages
```

Predicate *Inv* asserts that the last element of the *history* sequence records the current system state, that the time recorded at any history entry cannot exceed the current time, and that no agent attempts to execute an instruction whose deadline has passed. Predicate *MsgInv* states bounds on the contents of the message pool. First, any message that was sent corresponds to some send instruction of some agent in the execution history. Second, send instructions whose deadline has expired must indeed have been executed, and hence be in the message pool. The proofs of these invariants are straightforward.

4 Stating and Proving Determinacy

Our main result about the formal model of PharOS is that its executions are deterministic in the sense that the sequence of local states of every agent is the same in any execution, independently of the order of scheduling.

4.1 Witness Executions

Since we cannot directly refer to different executions in a linear-time formalism such as TLA$^+$, we will state and prove determinacy of executions in PharOS by relating any execution as specified by formula *Spec* to a fixed, statically chosen "witness" execution. The witness execution is represented as a stream of system states. We will choose the witness such that every agent executes infinitely often. Formally, we introduce the predicate

$$
\begin{array}{l}
\mathit{IsWitness}(H) \triangleq \\
\quad \land\ H \in \mathit{Stream}(\mathit{GoodSystemState}) \\
\quad \land\ H[1] = [\, \mathit{st} \mapsto \mathit{initState}, \mathit{msgs} \mapsto \{\}, t \mapsto 1\,] \\
\quad \land\ \forall n \in \mathit{Natp} : \exists a \in \mathit{Agent} : \mathit{executes}(a, H[n], H[n+1]) \\
\quad \land\ \forall a \in \mathit{Agent} : \forall n \in \mathit{Natp} : \\
\qquad\qquad \exists m \in \mathit{Natp} : m \geq n \land \mathit{executes}(a, H[m], H[m+1])
\end{array}
$$

In words, a witness is a stream of good system states that starts in the initial system state, where all transitions correspond to the execution of some agent, and where every agent executes infinitely often. Observe that the witness execution is represented as a TLA$^+$ value, independent of the actual system execution, and that the predicate *IsWitness* does not refer to any state variable. In particular, system states that appear in the witness are explicitly numbered.

In order to ensure that our subsequent results are not vacuous, we prove the existence of a witness execution.

$$\text{THEOREM } \mathit{witnessExistence} \triangleq \exists w \in \mathit{Stream}(\mathit{SystemState}) : \mathit{IsWitness}(w)$$

The proof relies on the inductive definition of an execution according to a specific scheduling strategy, for which we then prove that the resulting execution is a witness. The time progress assumption stated in Fig. 2 ensures that every agent must be scheduled infinitely often. We also prove some properties of witness executions similar to the invariants stated in Section 3.

4.2 Determinacy of Executions

In order to express determinacy, we define a predicate $\mathit{Det}(w)$ that relates the current execution and the witness w. More precisely, the predicate holds if for every agent a, the sequence of local states recorded in the history of the current execution agrees with the local states predicted by w.

$$
\begin{array}{l}
\mathit{Det}(w) \triangleq \exists U \in \mathit{Natp} : \forall a \in \mathit{Agent} : \\
\qquad\qquad \mathit{IsPrefix}(\mathit{filter}(\mathit{history}, a), \mathit{filter}(\mathit{take}(w, U), a))
\end{array}
$$

Since w is a stream of system states, the formal definition requires the existence of a sufficiently long initial subsequence of w such that the sequence of local states in the history is a prefix of the local states in that prefix of w. The determinacy theorem states that given any witness execution w, the predicate $Det(w)$ holds throughout the execution of the actual system.

THEOREM $Determinacy \triangleq$
  ASSUME NEW $w \in Seq(SystemState),\ IsWitness(w)$
  PROVE $Spec \Rightarrow \Box Det(w)$

The theorem is proved by induction, relying on the previously proved invariants of the specification and the witness. From the definition of the witness predicate and the initial condition $Init$, the two executions start in the same system state, hence the predicate Det holds trivially, choosing $U = 1$. Inductively, assume that the predicate holds for some prefix of w up to U, and that the system takes a non-stuttering step, with agent a executing its current instruction. Since $filter(history, a)$ is a prefix of $filter(take(w, U), a)$ and a takes infinitely many steps in w, there is some N such that

$$filter(history, a) = filter(take(w, N), a) \tag{1}$$

and a performs a step in w from system state $w[N]$ to $w[N + 1]$. Defining $nU \triangleq Max(U, N + 1)$, it suffices to show that

$$IsPrefix(filter(history', b), filter(take(w, nU), b))$$

holds for all agents b. This is easy to see for $b \neq a$, since $nU \geq U$ and the local state of b does not change in the transition of the system. Now assume $b = a$. Because $N + 1 \leq nU$, it is enough to show that

$$filter(history', a) = filter(take(w, N + 1), a).$$

From (1) it follows that $state[a] = w[N].st[a]$, i.e., the local states of a in the current execution and in the N-th configuration of the witness execution are the same. In particular, the same instruction is performed in both executions. For local updates and send instructions, this is enough for proving that $state'[a] = w[N + 1].st[a]$, since the same update is applied to the same state, and this implies the conclusion. For receive instructions, we must moreover show

$$msgReceived(w[N].msgs, instrOf(w[N].st[a]))$$
$$\Leftrightarrow\ msgReceived(messages, instrOf(state[a]))$$

in order to ensure that the updates are the same. For "$\Rightarrow$", if the message is received in the witness execution it must be in the message pool $w[N].msgs$, and it is visible. In particular, it was sent by some agent c, and the execution window of the send instruction strictly precedes the execution window of the current (receive) instruction. Therefore, the state at which c performed the send instruction is contained in the prefix $1\,..\,U$ of w and therefore also in the history of the system execution. Since no deadline is missed, the message must have

Module	Definitions (lines)	Proofs (lines)	# Theorems
Streams	2	50	5
Filters	9	624	27
System Model	78	226	10
Witness	22	832	11
Main proof	6	241	1
Total	117	1973	54

Table 1. Sizes of the different modules.

been sent in the system execution (recall invariant *MsgInv*), and will therefore be received.

Conversely, if the message is received in the system execution, it must have been sent by some agent c, and the deadline of the send instruction strictly precedes the execution window of the receive instruction. By invariant *MsgInv* and theorem *filter_Range*, the message appears in $sent(Range(filter(history, c)))$, hence by induction hypothesis also in $sent(Range(filter(take(w, U), c)))$. Because the deadline for sending the message has expired when the witness executes the receive instruction in the step from $w[N]$ to $w[N + 1]$, and using the analogue of *MsgInv* for the witness execution, the message must be contained in $witness[N].msgs$, and is therefore received.

4.3 Evaluation

Table 1 summarizes the sizes of the definitions and proofs (measured as the numbers of lines of TLA$^+$ code) that make up the different modules that we developed for the proof of determinacy, as well as the number of theorems proved in each module. The overall development required almost 2000 lines of proof, of which the main effort was devoted to proving auxiliary results about filters and the witness execution. The system specification (static and dynamic model) contains the bulk of the definitions; some basic invariants are also proved there.

The case study presented here also fed back into parts of the standard library of TLAPS. In particular, many lemmas about finite sequences were informed by the development of the proof of determinacy; these lemmas are being reused in other developments and do not appear in Table 1.

Compared to the paper-and-pencil proof of [9], although the main arguments are the same, the styles of proof differ in several respects.

- The original proof compares two possible system executions; determinism then means that for every agent a, the projection of one execution to the states of a must be a prefix of the other. This gives rise to two symmetric situations and would likely have led to duplications of proofs, which we avoided by choosing a fixed, infinite witness execution, of which the actual execution is always a prefix. We also prove the existence of a witness execution by exhibiting an actual scheduling strategy.

- The original proof is behavioral and refers to past and future instants of an execution, whereas the TLA$^+$ proof is assertional, relying on a central inductive invariant whose statement is distributed among the lemmas of the paper proof. This style is imposed by the formalism, but also follows established good practice on formalizing proofs of safety properties.
- The need for writing formal definitions in TLA$^+$ led us to introduce certain abstractions that were not explicit in the paper proof. For example, the notion of "good" execution times or system states proved to be very helpful in the machine-checked proof but are implicit in [9], where the assumption that deadlines are not missed is inferred from the assumption that agents execute infinitely often. Observe that our model does not impose a liveness condition on the specification of the system execution, and only requires non-Zenoness for the witness execution.
- Many of our auxiliary lemmas on filters and witness do not have a counterpart in the original proof, and in fact they may appear obvious to a mathematician. However, several mistakes that we made in the initial formal statements of these lemmas convinced us of the added value of a fully formal proof.

Our TLA$^+$ specification is written at the same level of abstraction as the model of computation of PharOS that was considered in [9]. In future work, it would be interesting to extend the specification and proof in two directions. First, we assume that no deadlines is ever missed. In typical instances of real-time systems, schedulability analysis ensures that this assumption is met. However, as discussed in [9], the result can be generalized to executions in which deadlines may actually be missed, or abrupt termination occurs for another reason. Second, it would be challenging to show that an actual implementation, as in the Asterios® system, refines our high-level model, and thus formally establish determinacy for an implementation. While TLA$^+$ includes a natural notion of refinement as trace inclusion, such a project would constitute a significant effort given the complexity of an actual implementation.

5 Conclusion

We have formally proved determinacy of the model of execution underlying PharOS, a real-time system that is now being commercialized as Asterios®. Based on an existing paper-and-pencil proof, our machine-checked proof reinforces the confidence in the result and clarifies some of the underlying assumptions. Moreover, our proof represents a sizable case study for the TLA$^+$ Proof System, which is still actively being developed. The overall proof effort appears to be reasonable. The final TLA$^+$ model and proof was obtained in several iterations, mainly focusing on the introduction of auxiliary abstractions. The hierarchical style of TLA$^+$ proofs helped us focus on the main argument and let us fill in proofs of auxiliary lemmas only when we knew that they were actually needed. Access to the TLC model checker was helpful for validating intermediate definitions and lemmas.

Several proposals exist in the literature for making computations of real-time systems deterministic, and we refer to [9] for a detailed discussion. The time-triggered architecture (TTA) [5] is probably the most widely known one. In comparison, PharOS imposes fewer static constraints on when tasks are scheduled, and it is designed to run on off-the-shelf hardware without a specific, deterministic communication substrate. TTA has also been the object of formal verification using the PVS proof assistant [14, 15]. In contrast to our work, which focuses on a high-level property of the execution model, these proofs focus on algorithms that underly the implementations of mechanisms such as clock synchronization or group membership. Relating our high-level model of execution with actual implementations of PharOS, down to actual code written in PsyC (a real-time extension of the C language) is left as a challenge for future work.

Acknowledgements. Jael Kriener contributed to this work by writing initial specifications and proofs of PharOS executions.

References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
2. C. Aussaguès and V. David. A method and a technique to model and ensure timeliness in safety critical real-time systems. In *4th Intl. Conf. Engineering of Complex Computer Systems (ICECCS '98)*, pages 2–12, Monterey, CA, U.S.A., 1998. IEEE Comp. Soc.
3. N. Azmy, S. Merz, and C. Weidenbach. A rigorous correctness proof for Pastry. In M. Butler and K.-D. Schewe, editors, *5th Intl. Conf. Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2016)*, LNCS, Linz, Austria, 2016. Springer. This volume.
4. D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto. TLA$^+$ proofs. In D. Giannakopoulou and D. Méry, editors, *18th Intl. Symp. Formal Methods (FM 2012)*, volume 7436 of *LNCS*, pages 147–154, Paris, France, 2012. Springer.
5. H. Kopetz and G. Bauer. The time-triggered architecture. *Proc. of the IEEE*, 91(1):112–126, 2003.
6. L. Lamport. *Specifying Systems.* Addison-Wesley, Boston, Mass., 2002.
7. L. Lamport. Byzantizing Paxos by refinement. In D. Peleg, editor, *25th Intl. Symp. Distributed Algorithms (DISC 2011)*, volume 6950 of *LNCS*, pages 211–224, Rome, Italy, 2011. Springer.
8. M. Lemerre, V. David, C. Aussagus, and G. Vidal-Naquet. Equivalence between schedule representations: Theory and applications. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pages 237–247, April 2008.
9. M. Lemerre and E. Ohayon. A model of parallel deterministic real-time computation. In *Proc. 33rd IEEE Real-Time Systems Symp. (RTSS 2012)*, pages 273–282, San Juan, PR, U.S.A., 2012. IEEE Comp. Soc.
10. M. Lemerre, E. Ohayon, D. Chabrol, M. Jan, and M.-B. Jacques. Method and tools for mixed-criticality real-time applications within PharOS. In *14th IEEE Intl. Symp. Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pages 41–48, Newport Beach, CA, U.S.A., 2011. IEEE Comp. Soc.

11. M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *Software Tools for Technology Transfer*, 10(2):185–203, 2008.
12. S. Louise, M. Lemerre, C. Aussaguès, and V. David. The OASIS kernel: A framework for high dependability real-time systems. In *13th IEEE Intl. Symp. High-Assurance Systems Engineering (HASE 2011)*, pages 95–103, Boca Raton, FL, U.S.A., 2011. IEEE Comp. Soc.
13. C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How amazon web services uses formal methods. *CACM*, 58(4):66–73, 2015.
14. H. Pfeifer and F. W. von Henke. Modular formal analysis of the central guardian in the time-triggered architecture. *Reliability Engineering & System Safety*, 92(11):1538–1550, 2007.
15. J. Rushby. An overview of formal verification for the time-triggered architecture. In W. Damm and E.-R. Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *LNCS*, pages 83–105, Oldenburg, Germany, 2002. Springer.
16. Y. Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *LNCS*, pages 54–66, Bad Herrenalb, Germany, 1999. Springer Verlag.